Introduzione al Linguaggio C

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2016-17

La presente dispensa introduce il nucleo del linguaggio di programmazione C, prevalentemente attraverso piccoli programmi che risolvono semplici problemi. Verrà inoltre isolato un nucleo "minimale" del linguaggio C (che a volte appare in letteratura come Tiny C), sufficiente però a risolvere qualunque problema. Verranno quasi subito introdotte le funzioni, in modo da stimolare la soluzione dei problemi come composizione di soluzioni a problemi più semplici.

Per un trattamento estensivo del linguaggio C, il lettore è rimandato a un qualsiasi manuale del linguaggio.

1 Primi Programmi in Linguaggio C

Nel corso di queste esercitazioni useremo il linguaggio C per formalizzare le procedure risolutive ai problemi che via via ci porremmo. Il primo programma C, è il programma `HelloWorld`, che stampa a video la scritta `HelloWorld`.

```
#include <stdio.h>
int main(){
  /* il mio primo programma C */
     printf("HelloWorld\n");
     return 0;
}
```

Per eseguire questo programma è necessario prima *compilare* il programma, cioè tradurlo in un linguaggio meno astratto del C, eseguibile della macchina. In un sistema di tipo Unix (o Mac Os X o Linux) è sufficiente scrivere il programma con un editor di testo, salvarlo, ad esempio con il nome `helloWorld.c` e a prompt di sistema scrivere `>gcc helloWorld.c`. Questo produrrà un file di nome `a.out`.

Per eseguirlo, sarà sufficiente digitare dal prompt di sistema `>./a.out`. È possibile anche dare un nome diverso all'eseguibile, con l'opzione `-o` del `gcc`. Ad esempio, il comando `>gcc -o hw helloWorld.c` produrrà il file eseguibile `hw` e per eseguirlo sarà sufficiente digitare dal prompt di sistema `>./hw`.

Ogni programma C deve contenere una funzione `main()` che contiene il codice messo in esecuzione all'avvio del programma. La funzione `printf` stampa a video il contenuto della stringa che le viene passata come parametro. `\n` è un carattere speciale che indica il carattere "vai a capo". La funzione `printf` è una cosidetta *funzione di libreria*, ossia una funzione di uso comune che viene messa a disposizione di tutti i programmatori C. Per poterla usare, dovete informare il compilatore che intendete usare le funzioni standard di input/output, contenute nella libreria `stdio.h`. Questo è (in grande sintesi) il significato della prima riga, `#include <stdio.h>`. Le frasi scritte tra i caratteri `/*` e `*/` sono ignorate dal compilatore e servono per scrivere commenti. Noi useremo i commenti anche per descrivere *proprietà logiche* dei programmi, come *precondizioni*, *postcondizioni* e *invarianti*. In generale, i commenti servono per rendere *leggibile* il codice e facilitare future modifiche.

1.1 Linguaggio C: una Introduzione Rapida

Per scrivere in linguaggio C le procedure risolutive viste nelle sezioni precedenti, dovremmo conoscere un frammento minimo del linguaggio[1].

Un programma C consiste in una sequenza di *comandi*. I comandi servono in generale a modificare lo stato della computazione, cioè la *memoria*, che il programmatore C può gestire tramite le *variabili*. Le variabili sono astrazioni delle celle di memoria, o se preferite, nomi per celle di memoria.

Il comando base è l'*assegnazione* che permette di modificare il valore di una variabile e in C si scrive `x=E;`, dove `x` è una variabile ed `E` è una *espressione*. Un'espressione è del tutto analoga alle espressioni che vi sono note, e può contenere variabili e operatori aritmetici. Ad esempio `x+y-3` è un'espressione. Un po' per gioco, un po' per motivi più seri, in questi primi programmi useremo solo l'espressione `x+1`. Cioè immagineremo che il nostro esecutore che noi comanderemo usando il linguaggio C, sia in grado solo di sommare 1. Per aderire al galateo del linguaggio C, che prevede di risparmiare caratteri, diciamo subito che il comando `x=x+1;` può essere abbreviato con `x++;`[2].

Più comandi possono essere messi in *sequenza* separandoli con il simbolo separatore `;`. Quando una sequenza di comandi deve essere considerata come un *unico* comando va racchiusa tra graffe. Ad esempio {`i++; j++;`} può essere visto come un unico comando. Ciò serve a delimitare, ad esempio, il *corpo di un ciclo* (cioè i comandi da ripetere) o i rami di un costrutto condizionale.

[1] ricordiamo che questa è una introduzione minimale. Il lettore è invitato a consultare un manuale del linguaggio.

[2] **attenzione**: questo comando abbrevia `x=x+1` e non semplicemente l'espressione `x+1`.

Il *comando condizionale* `if (E) C1 else C2;` permette di eseguire una porzione di codice o un'altra a seconda del valore di una espressione. Diciamo che se l'espressione `E` valuta al valore `true` viene eseguito il comando C_1, viceversa viene eseguito il comando C_2. Tradizionalmente, nella maggioranza dei linguaggi di programmazione, è richiesto che `E` sia una espressione *booleana*, cioè una espressione il cui risultato sia un valore di verità, cioè `true` oppure `false`. In C non esiste il tipo booleano, ma in compenso ad ogni tipo è associato una interpretazione in valori di verità. Per esempio, per gli interi, 0 viene interpretato come `false`, mentre ogni valore diverso da 0 viene interpretato come `true`. Per cui: `if (0) printf("pippo") else printf("paperino");` stamperà il messaggio `paperino`, mentre `if (4) printf("pippo") else printf("paperino");` stamperà il messaggio `pippo`. Valutano a valori di verità le espressioni che contengono gli operatori relazionali `==` (test di uguaglianza), `<` (minore), `<=`, `>`, `>=`, `!=` (test d'uguaglianza negato). Attenzione alla differenza sintattica tra assegnazione e test d'uguaglianza! Nel seguito, ancora una volta, immagineremo di poter usare solo il test d'uguaglianza tra numeri interi e quindi gli operatori `==` e `!=`. Il simbolo `!` messo davanti ad una espressione, ne inverte il valore di verità e quindi è la sintassi C per il *not* logico. Quindi `a!=b` in C è un'abbreviazione equivalente a `!(a==b)`.

Per ripetere l'esecuzione di alcune istruzioni, vengono usati dei *comandi iterativi*. Nel comando `while (E) C` il comando `C` verrà ripetuto fino a che la condizione `E` valuta a `true`. Usualmente la condizione `E` viene detta *guardia del ciclo*, mentre il comando da ripetere `C` viene detto *corpo del ciclo*. Ad esempio il comando `while(1){}` è il comando che non fa nulla per l'eternità (beato lui!). Osservate inoltre che il controllo della guardia è fatto *prima* di eseguire il ciclo, per cui, ad esempio, in `x=0; while(x) x++;`, il comando `x++` non viene mai eseguito.

Il ciclo più usato è senz'altro il ciclo `for`. Questo ciclo tradizionalmente serve a ripetere un comando un numero *prefissato* di volte (cioè noto all'ingresso del ciclo), di solito con associata una variabile *contatore* che conta quante iterazioni sono state eseguite finora. In linguaggio C, il ciclo `for` è particolarmente espressivo ed ha la forma:

$$\texttt{for (}C_1\texttt{; E; }C_2\texttt{) }C_3$$

con la seguente semantica:

> Esegui il comando C_1, ripeti il comando C_3 finché `E` valuta a un valore `true`. Alla fine di ogni iterazione esegui C_2.

L'uso tipico, tuttavia è nella forma che lo rende simile al ciclo for di altri linguaggi di programmazione, in cui si usa il comando C_1 per inizializzare una variabile contatore, `E` per valutare se la variabile contatore sia dentro i limiti previsti, e C_2 per incrementare la variabile contatore. Il più delle volte scriverete cicli `for` nella forma:

```
for (i=0; i<n; i++) C
```

1.2 Esercizi e Spunti di Riflessione

1. ♣ Far vedere che il comando iterativo `for (`C_1`; E; `C_2`)` C_3 non è necessario, in quanto si può scrivere un comando "equivalente" che usa solo il comando `while`.

2. ♣ ★ Far vedere che il comando condizionale `if (E)` C_1 `else` C_2 non è necessario, in quanto si può scrivere un comando "equivalente" che usa solo il comando `while` (**attenzione**: questo esercizio è più insidioso di quanto non possa sembrare a prima vista!).

3. Scrivete i diagrammi di flusso equivalenti al comando condizionale `if (E)` C_1 `else` C_2 e al comando iterativo `while (E) C`.

4. ▶ Attenzione al fatto che in linguaggio C i comandi sono anche espressioni, cioè ritornano un valore. Di conseguenza, il comando `if (x=0) printf("pippo"); else printf("paperino");` è legale, anche se `x=0` è un comando di assegnazione e non una espressione condizionale. Verificate cosa stampa questo frammento di programma (indipendentemente dal valore di `x` prima dell'`if`). Verificare anche cosa viene stampato dal comando `if (x=4) printf("pippo"); else printf("paperino");`. Cosa ne deducete?

5. ♣ In linguaggio C esiste il comando `do C while (E)` che esegue il comando `C` fino a che l'espressione `E` valuta a `false`. Inoltre, a differenza del comando `while`, la condizione viene valutata *dopo* aver eseguito il comando `C` almeno una volta. Scrivete un comando "equivalente" che usa solo il comando `while`. Scrivete il diagramma di flusso equivalente.

6. ▶ Anche l'espressione (o comando?) vuoto ha un senso. Testare il significato di `if ()` C_1 `else` C_2. Provare a dare un'interpretazione.

2 Programmi C per Problemi Elementari

In Fig. 1 è mostrata la traduzione in linguaggio C del semplice algoritmo che calcola la somma tra due numeri naturali, e che suppone un esecutore in grado solo di calcolare il successore e il test di uguaglianza. Osserviamo che tutte le variabili *devono essere dichiarate*, cioè prima di usarle è necessario assegnare loro un *tipo*. Con la frase `int i, m, n;` informiamo il compilatore che le variabili `i`, `m` ed `n` sono variabili *intere*. Un tipo è caratterizzato da un *insieme di valori* che una variabile di quel tipo può assumere e dalle *operazioni* che sono lecite su quel tipo: il compilatore, traducendo il programma verificherà anche che le variabili vengano usate coerentemente con il loro tipo (questa verifica si chiama *type checking*).

Il tipo `int` del C è per la verità molto diverso dal tipo *numero naturale* a cui ci siamo astrattamente riferiti nelle sezioni precedenti. Infatti i possibili valori di

```c
#include <stdio.h>

int main(){
  int i, m, n;

  scanf("%d",&m);
  scanf("%d",&n);
  i=0;
  while (i!=n){
       i++;
       m++;
        }
  printf("La Somma vale %d\n",m);
  return 0;
}
```

Figura 1: Programma somma in C

```c
#include <stdio.h>

int main(){
  int i=1;
  int j=0;
  int n;

  scanf("%d",&n);
  while (i!=n){
       i++;
       j++;
        }
  printf("Pred di %d vale %d\n",n,i);
  return 0;
}
```

Figura 2: Predecessore in C

una variabile definita `int` sono i numeri interi compresi nell'intervallo $[-2147483648, 2147483647]$ (sul calcolatore che uso in questo periodo. Ricordo con nostalgia quando questo intervallo era $[-32768, 32767]$). Questo intervallo è un po' meno misterioso se lo scriviamo come $[-2^{31}, 2^{31} - 1]$ (e ancora meno, se osserviamo che, nel calcolatore che usavo nella mia adolescenza era $[-2^{15}, 2^{15} - 1]$). Se vogliamo solo interi positivi, possiamo dichiarare le variabili `unsigned int`. In tal caso, i possibili valori saranno compresi nell'intervallo $[0, 4294967295]$ (sempre sulla mia macchina). Anche questo intervallo può più convenientemente essere scritto come $[0, 2^{32} - 1]$.

La procedura `scanf("%d",&m);` aspetta l'inserimento di un valore da tastiera e carica la variabile `m` con tale valore. Per ora manteniamo il più stretto riserbo sul perché occorra scrivere `&m` invece che semplicemente `m`. `"%d"` sia nella `scanf` che nella `printf` indica il punto in cui l'intero verrà stampato e con che formato. Il lettore è invitato a studiare i possibili formati di stampa su un manuale. Tutto il resto dovrebbe essere immediatamente comprensibile.

Non dovrebbe richiedere molte spiegazioni neanche il codice in Fig. 2 che calcola il predecessore di un numero naturale. L'unica osservazione riguarda il fatto che possiamo dare a una variabile un valore iniziale contestualmente alla sua dichiarazione (vedere le dichiarazioni di `i` e `j`). Ricordo a tal proposito, che inizializzare le variabili contestualmente alla loro dichiarazione è una buona norma di galateo di programmazione ed evita errori imprevedibili dovuti al fatto che non possiamo, in generale, fare assunzioni sul valore iniziale delle variabili. Infine, nella `printf` stampiamo il valore di due variabili (e diligentemente mettiamo due campi con `%d`).

2.1 Esercizi e Spunti di Riflessione

1. Scoprire quali sono i possibili valori per le variabili `int` nel vostro calcolatore. Fare lo stesso per altri tipi interi come `unsigned int`, `short int`, `long int` etc.
2. ▶ Provare ad eseguire un programma che calcola il predecessore inserendo 0 come valore di input per `n`. Come vi spiegate il risultato ottenuto?
3. ▶ Forse riuscite ad avere qualche idea sull'esercizio precedente, se confrontate il risultato con quello che ottenete dal programma in cui definite tutte le variabili come `unsigned int` e usando come istruzione di stampa `printf("Il predecessore vale %u\n",i);`
4. Se avete risolto i due esercizi precedenti, forse riuscite a risolvere l'esercizio 1 semplicemente scrivendo dei programmi che calcolano il range di valori associati ai diversi tipi interi.
5. ▶ Provate a togliere le inizializzazioni a `i` e `j` nel programma predecessore. Osservate i risultati. Traetene un insegnamento per il futuro!
6. ▶ Provate a togliere il carattere `&` nelle `scanf` del programma `somma`. Osservate i risultati. Riprovate, dopo aver inizializzato le variabili `m` ed `n`. Ipotizzate cosa cambia togliendo il simbolo `&`.

3 Meccanismi per Creare Astrazioni: le Funzioni

Poniamoci ora il problema di codificare in C la procedura risolutiva che calcola la moltiplicazione partendo dall'esecutore che sa fare solo il successore. La soluzione data al Problema 4 nelle note "Problemi, Soluzioni, Algoritmi" (tradotta in C in Fig. 3), in cui il prodotto viene ottenuto iterando l'operazione di somma, andrà ulteriormente dettagliata, inserendo il codice del programma per la somma, in sostituzione dell'istruzione proibita `prod=prod+m`. Questo, oltre che noioso, va fatto con la dovuta cautela. Per esempio, volendo inserire il ciclo del programma in Fig. 1, dovremmo avere cura di evitare che tale ciclo non "sporchi" il valore delle variabili `m` e `i` usate dal programma per il prodotto. Per ottenere risultati corretti dovremmo quindi introdurre nuove variabili con nuovi nomi. Tali variabili, seguendo una tradizione della Logica Matematica, vengono di solito chiamate *fresche*. Il codice che "vorremmo scrivere" si trova in Fig. 3, mentre quello che siamo obbligati a scrivere si trova in Fig. 4.

3.1 Funzioni: Parametrizzazione

D'altra parte, è naturale pensare che una volta scritta la soluzione del problema della somma, noi possiamo risolvere i problemi riferendoci ad un esecutore più abile

```c
#include <stdio.h>

int main(){
  int p=0;
  int i=0;

  scanf("%d",&m);
  scanf("%d",&n);

  while (i!=n) {
        i++;
        prod=prod + m;
        }

  printf("%dx%d vale %d\n",m,n,p);
  return 0;
 }
```

Figura 3: Moltiplicazione, vers. 1

```c
#include <stdio.h>
int main(){
  int p=0;
  int i=0;
  int j;
  scanf("%d",&m);
  scanf("%d",&n);
  while (i!=n) {
        i++;
        j=0;
        while (j!=m) {
              prod++;
              j++;
        }
  }
  printf("%dx%d vale %d\n",m,n,p);
  return 0;
 }
```

Figura 4: Moltiplicazione, vers. 2

in grado di eseguire anche la somma. Per fortuna, i linguaggi di programmazione e il C in particolare, offrono adeguati meccanismi di astrazione, le *funzioni*, che ci permetteranno di scrivere del codice molto simile a quello in Fig. 3.

Piuttosto di scrivere il programma per la somma come in Fig. 1, definiremo una funzione per calcolare la somma (Fig. 5), che calcolerà il risultato della somma in funzione dei due parametri di input `m` ed `n`. La scrittura `int somma(int m, int n)` dice infatti che somma calcolerà un valore intero, che dipende dai parametri interi `m` ed `n`. `m` ed `n` sono detti *parametri formali* e devono obbligatoriamente essere variabili.

Una volta definita la funzione `somma`, è possibile invocarla, ad esempio nel comando `printf("La somma di 3 e 4 vale %d",somma(3,4));`. `3` e `4` sono detti *parametri attuali* e istanziano il valore dei parametri formali. I parametri attuali sono espressioni, quindi possono essere costanti, variabili, o, a loro volta, chiamate ad altre funzioni. Ad esempio, il seguente frammento di codice è perfettamente lecito:

```c
int x=3;
int y=2;
printf("%d",somma(x, somma(x, somma(y,3+x))));
```

Si ricordi, che *prima* viene valutata l'espressione, e *poi* il valore ottenuto viene copiato nella casella di memoria associata alla variabile-parametro formale. Questa sequenza di operazioni prende il nome di *passaggio di parametri per valore*. Nel nostro esempio, per valutare l'espressione `somma(x, somma(x, somma(y,3+x))`, dovremmo valutare

```
int somma(int m, int n){
  int i=0;
  while (i!=n) {
        i++;
        m++;
        }
  return m;
 }
```

Figura 5: Funzione somma

```
int molt(int m, int n){
  int p=0; int i=0;
  while (i!=n) {
        i++;
        p=somma(p,m);
        }
  return p;
 }
```

Figura 6: Funzione moltiplicazione

`x` (che ovviamente valuta a `3`) e poi valutare `somma(x, somma(y,3+x))`. Questo a sua volta richiede prima di tutto di valutare `x` (che valuta ancora a `3`, ma non date questo fatto troppo per scontato!), e valutare `somma(y,3+x)`. Qui i parametri valutano rispettivamente a `2` e `6` e finalmente verrà messa in esecuzione la funzione `somma` che restituirà `8`. Ora siamo pronti a eseguire un'altra volta la funzione `somma` con parametri attuali `3` ed `8` e otterremo (sperabilmente) `11`. A questo punto siamo finalmente pronti per eseguire la chiamata più esterna della funzione `somma` sui parametri `3` e `11` e otterremo il risultato finale da stampare, `14`. Provare per credere.

3.2 Variabili: Regole di Scoping

Osserviamo, che nella funzione `somma` sia i parametri formali (`m` ed `n`), sia le variabili definite all'interno della funzione, `i`, sono *locali* alla funzione: cioè nascono al momento in cui la funzione viene chiamata e scompaiono quando la funzione finisce di eseguire. Più importante ancora, le modifiche alle variabili `i`, `m` ed `n` non interferiscono con eventuali altre variabili che si chiamino `i`, `m` ed `n` esistenti fuori dalla funzione `somma`.

Le funzioni che abbiamo scritto (e che scriveremo in futuro) sono *chiuse*, cioè non riferiscono mai variabili che non siano locali. Occorre ricordare che in `C` è possibile definire anche variabili *globali*, cioè variabili che sono visibili da ogni funzione. Le variabili globali sono definite *fuori* da ogni funzione (quindi anche fuori dal `main`). Qualora in una funzione sia riferita una variabile globale, eventuali modifiche saranno visibili ovunque. Se in una funzione viene definita una variabile locale o un parametro con lo stesso nome di una variabile globale, la variabile locale (o il parametro) *maschera* la variabile globale. Le regole qui sintetizzate, che riassumono le regole che stabiliscono la *visibilità* delle variabili, sono dette *regole di scoping*. L'uso (e soprattutto l'abuso) di variabili globali è vivamente sconsigliato, in quanto può causare effetti difficilmente prevedibili.

Infine, una funzione trasmette un valore al chiamante attraverso l'istruzione `return E`, dove `E` è una espressione compatibile con il tipo di ritorno dichiarato per la funzione. L'effetto dell'esecuzione di `return` è sospendere l'esecuzione della funzione, e passare il controllo dell'esecuzione all'istruzione successiva a quella in cui

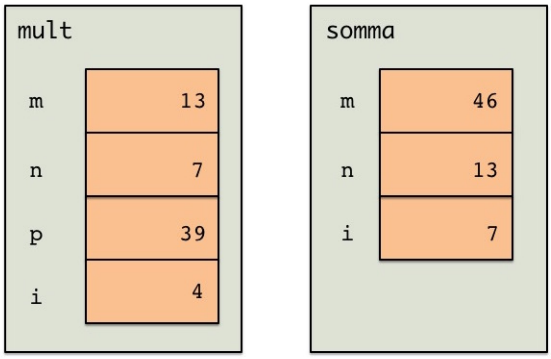

Figura 7: Stato della memoria durante il calcolo di `mult`

la funzione è stata chiamata. Il valore ottenuto dalla valutazione dell'espressione `E` è il valore calcolato dalla funzione. Il `return` sospende l'esecuzione della funzione, anche se non è l'ultima istruzione della funzione.

In Fig. 6 vediamo l'utilizzo della funzione `somma`, all'interno di una funzione che calcola la moltiplicazione. Osservate bene che ogni funzione ha il suo *stato locale*. In particolare, variabili con nomi uguali definite nello stato locale delle due funzioni (come `i` ed `m`) indicheranno celle di memoria diverse. Ad esempio, una chiamata della funzione `mult(13,7);` causerà 7 chiamate della funzione `somma`, con diversi parametri: `somma(0,13)`, `somma(13,13)`, `somma(26,13)`, `somma(39,13)` etc. Nella chiamata `somma(39,13)` i parametri `m` ed `n` di somma vengono inizializzati a `39` e `13` rispettivamente e viene sommato 1, 13 volte alla variabile `m`. Dopo ad esempio 7 iterazioni lo stato della memoria è quello mostrato in Fig. 7.

A questo punto, possiamo divertirci a scrivere tutte le funzioni che implementano tutte le operazioni aritmetiche o relazionali che ci vengono in mente, a cominciare da quelle proposte nei problemi delle note "Problemi, Soluzioni, Programmi".

```
int pred(int n){
/* PREC: n>0 */
  int i=1; int j=0;
  while (i!=n) {
  /* INV: j=i-1 */
        i++;
        j++;
        }
  return j;
}
```

Figura 8: Funzione predecessore

```
int diff(int m, int n){
/* PREC: 0<=n<=m */
  int i=0; int j=n;
  while (j!=m) {
  /* INV: j-i=n */
        i++;
        j++;
        }
  return i;
}
```

Figura 9: Funzione differenza

```
int minore(int m, int n){
/* PREC: m,n >= 0 */
  int i=0;
  if (m==n) return 0;
  while (1) {
  /* INV: i<n and i<m */
        if (i==n) return 1;
        if (i==m) return -1;
        i++;
        }
}
```

Figura 10: Funzione minore

```
int div(int m, int n){
/* PREC: m>=0,0<n<=m */
  int q=0;
  int r=n;
  while (minore(r,n)==1) {
  /* INV: q*n+r=m */
         q++;
         r=diff(r,n);
        }
  return q;
}
```

Figura 11: Funzione divisione

In Fig. 8, abbiamo il calcolo del predecessore, in versione di funzione. Osservate che, come prima cosa abbiamo scritto un commento che esplicita la *precondizione* della funzione, ossia le assunzioni che permettono a `pred` di funzionare correttamente. Inoltre, sempre come commento, abbiamo scritto la proprietà invariante del ciclo `while`. Osservate che questa proprietà invariante non è stata scelta a caso: ci sono numerosissime altre proprietà invarianti per quel ciclo (per esempio $i > 0$), ma questa proprietà invariante, è intimamente legata a quello che la funzione `pred` calcola. Infatti l'invariante e la negazione della guardia (che è $i = n$) implicano proprio che $j = n - 1$, ed infatti il valore della variabile `j` viene restituito al chiamante come risultato, coerentemente con il comportamento atteso dalla funzione `pred`.

Ricordiamo, che tutto ciò ha senso solo se il ciclo *termina*. Nelle assunzioni della precondizione $n > 0$, abbiamo chiaramente che $i \leq n$ è un'altra proprietà invariante per questo ciclo. Quindi la funzione $n - i$ è una funzione a valori positivi strettamente decrescente a ogni ciclo (infatti `i` viene incrementata nel corpo del `while`). Una tale funzione è detta *funzione di terminazione*, e l'esistenza di una tale funzione costituisce una dimostrazione che il ciclo termina. Chiaramente, lo ripetiamo, nel caso in cui siano soddisfatte le precondizioni della funzione. Viceversa, infatti ($n \leq 0$), $n - i$ potrebbe assumere valori negativi.

3.3 Esercizi e Spunti di Riflessione

1. ▶ Data la funzione `somma` vista prima, considerate il seguente programma:

```
int x=1;
int main() {int x=3; printf("%d", foo(3));}
int foo(y){return somma(x,y);}
```

   Qual è l'output? Motivate la risposta.

2. ▶ Considerate il seguente programma:

```
int x=1;
int main() {int x=3; printf("%d", somma(x,foo(3)));}
int foo(y){x++; return somma(x,y);}
```

   Qual è l'output? Motivate la risposta.

3. ▶ Considerate il seguente programma:

```
int x=1;
int main() {int x=3; printf("%d", somma(foo(3),x));}
int foo(y){x++; return somma(x,y);}
```

   Qual è l'output? Motivate la risposta.

4. Scrivere una funzione `int exp(int m, int n)`, iterando le chiamate alla funzione `mult(m,n)`. Quale deve essere il valore iniziale della variabile che accumula il risultato? ♣ Riflettete su quante operazioni vengono eseguite. È possibile fare meglio nelle tristi condizioni in cui ci troviamo (cioè con un esecutore che sa solo sommare 1)?

5. Considerate il programma di Fig. 10. In cosa differisce dal diagramma di flusso di Fig. 2 nelle note "Problemi, Soluzioni, Programmi"? Scrivere il diagramma di flusso corrispondente al programma in Fig. 10. E ovviamente il programma corrispondente al succitato diagramma di flusso.

6. ♣ Considerate le precondizioni delle funzioni `diff` (Fig. 9) e `div` (Fig. 11). Cosa accade se non sono soddisfatte? Scrivete le funzioni di terminazione, e dare argomenti informali sul fatto che le funzioni terminino (e sotto quali assunzioni) e che le proprietà scritte nei cicli siano effettivamente proprietà invarianti.

7. Scrivere una funzione per la differenza alternativa a quella in Fig. 9, che itera la funzione `pred`. Come cambia la complessità computazionale?

8. Scrivere una funzione `int fatt(int n)` che calcola il fattoriale di `n`. Corredare di precondizioni e invarianti.

9. Scrivere una funzione `C int resto(int m, int n)` che restituisce il resto della divisione intera tra `m` ed `n`. In cosa differisce questa funzione rispetto alla funzione `div`?

10. Scrivere un programma C che dato un intero n in input, stampi tutte le coppie di numeri interi che danno n come prodotto. Evitare di riscrivere coppie simmetriche.

11. Scrivere un programma C che dato un intero n in input, stampi tutte le terne di numeri interi che danno n come prodotto. ★ Evitare di riscrivere terne che differiscono solo per l'ordine dei fattori.

12. Scrivere una funzione C `int primo(int n)` che restituisce 1 se `n` è un numero primo e 0 altrimenti.

4 Esempi: Programmi C con Asserzioni Logiche

Quest'ultima sezione è dedicata a qualche esempio un po' più impegnativo dal punto di vista algoritmico.

Problema 1: *Scrivere una funzione che calcoli la moltiplicazione sfruttando le seguenti uguaglianze:*

$$\begin{aligned} m \times 2n &= (m+m) \times n \quad (n > 1) \\ m \times (2n+1) &= m \times 2n + m \\ m \times 0 &= 0 \end{aligned}$$

Questo algoritmo (che si può applicare anche al calcolo dell'esponenziale) si basa sull'idea che moltiplicare per 2 o fare somme è relativamente semplice rispetto a fare prodotti. È noto come *moltiplicazione egiziana*, in quanto sembra fosse già noto agli antichi egizi. Per la verità qualcuno lo chiama *moltiplicazione russa*. Il suo grande vantaggio, rispetto alle n somme richieste dalla funzione in Fig. 6, è che permette di fare un numero di somme molto piccolo (quante?).

Osserviamo, che la definizione di moltiplicazione è data per *induzione* sul secondo fattore. Viene definito il significato della moltiplicazione distinguendo i casi in cui il secondo fattore sia un numero pari (maggiore di zero!), un numero dispari oppure zero (caso base). In ogni caso, il significato di $m \times n$ $(n > 1)$ viene dato induttivamente in termini di una moltiplicazione con un secondo fattore minore.

Soluzione: Per comodità, indicheremo con m_0 ed n_0 i valori iniziali da moltiplicare, e con m, n e p i valori delle variabili `m`, `n` e `p`, e con m', n' e p' i loro valori dopo l'esecuzione del corpo del ciclo.

L'idea di accumulare i risultati in una variabile `p`, in modo che venga mantenuta la proprietà invariante $m_0 n_0 = mn + p$. Questa relazione è banalmente soddisfatta assegnando inizialmente 0 a `p` (e ovviamente $m = m_0$ e $n = n_0$).

Quando n è pari, raddoppiando m e dividendo n per 2, la relazione si mantiene perché $m'n' + p' = 2m\frac{n}{2} + p = mn + p$. Quando n è dispari, toglieremo 1 a n e aggiungeremo m a p, mantenendo la relazione in quanto in questo caso $m'n' + p' =$

```
int multiplyingLikeAnEgiptian(int m, int n){
/* PREC: m,n>= 0 */
 int p=0;
 while (n!=0){ /* INV: m0 * n0 = m*n+p  */
   if (resto(n,2) == 0){
       m = somma(m,m);
       n = div(n,2);
     } else {
         p = somma(p,m);
         n = pred(n);
       }
   } /* end while */
   return p;
} /* end function */
```

Figura 12: Funzione per la moltiplicazione egiziana

$m(n-1)+(p+m) = mn-m+p+m = mn+p$. Usciremo dal ciclo quando raggiungiamo il caso base delle equazioni induttive scritte sopra, cioè quando $n = 0$. In tale situazione $mn + p = m_0 n_0$ implica proprio che $p = m_0 n_0$, che è quanto volevamo ottenere. Il caso base viene raggiunto, perché la guardia ($n \neq 0$) implica che $n' < n$. Fatte queste osservazioni, il programma si scrive da solo, come in Fig. 12.

Problema 2: *Scrivere una funzione che calcola il massimo comun divisore.*

Soluzione 1: Per risolvere questo problema seguiremo diverse strategie. Cominciamo con un algoritmo "ingenuo". Siano m ed n i numeri di cui vogliamo calcolare $\mathsf{mcd}(m, n)$. Prendiamo il minore dei due (diciamo sia $p = \mathsf{min}(m, n)$) e proviamo tutti i numeri $p, p-1, p-2 \ldots 2, 1$. Il primo valore che divide sia m che n è il massimo comun divisore di m ed n. Il programma corrispondente in Fig. 13.

Soluzione 2: Il metodo precedente, in effetti, benchè intollerabile per un esecutore umano, non è poi così male per i calcolatori moderni, che riescono a fare moltissime operazioni semplici in poco tempo. Certo, dovessimo chiederci chi è il massimo comun divisore per numeri a 10 cifre primi tra loro, il costo computazionale diventerebbe proibitivo. Un algoritmo più intelligente è quello che vi è noto già dalle elementari (o medie?). La maestra recitava una filastrocca del tipo:

> *Il massimo comun divisore di due numeri è il prodotto di tutti i fattori (primi) comuni, presi con il loro minimo esponente.*

Lasciamo al lettore il piacere di dimostrare che questa filastrocca corrisponde a una verità matematica. Ma come insegnare questa filastrocca al nostro calcolatore usando il frammento di C che conosciamo? Il compito sembra a prima vista proibitivo:

```
int mcdIngenuo(int m, int n){
/* PREC: m,n> 0 */
    int p;
    if (minore(m,n)==1) p=m; else p=n;
    while (p!=0){
    /* INV:forall j<p.!(j | n & j | m */
      if (resto(n,p) == 0)
          if (resto(m,p)=0)
              return p;
      p=pred(p);
      }
     return p;
}
```

Figura 13: Calcolo dell'MCD: algoritmo ingenuo.

infatti il numero di fattori di un certo naturale n è variabile (e potenzialmente illimitato al crescere di n). Allo stato delle nostre conoscenze, avremmo bisogno di un numero non prevedibile a priori di variabili.

Risolviamo dapprima un compito più semplice. Scriviamo una funzione che scrive tutti i fattori primi di un numero n (Fig. 14). Anche qui, ci sono molte questioni interessanti da analizzare. Perché ad esempio, tale programma stampa solo i fattori primi? Cercate almeno una giustificazione informale. Facciamo inoltre conoscenza con una particolare forma di funzione, in cui il valore di ritorno è `void`. `void` è un curioso tipo, che ha un solo elemento `()`. In generale, viene dichiarato `void` il tipo di ritorno delle funzioni che non restituiscono nessun valore. Tali funzioni, tradizionalmente, vengono dette *procedure*. Idealmente, una funzione è un'astrazione di un'espressione, mentre una procedura è un'astrazione di un comando. In C, a causa della promiscuità ammessa tra comandi ed espressioni (vedi Esercizio 1.2(4)), questa separazione è meno stigmatizzata: esistono solo funzioni, alcune delle quali restituiscono un valore non significativo di tipo `void`. Le funzioni, come le procedure, hanno il diritto di modificare lo stato *globale* della memoria (comportandosi quindi come comandi, vedere esercizi 3.3(1–3)).

Torniamo al calcolo dell'MCD. Il programma in Fig. 14 genera i fattori primi di un numero. Ma per risolvere il nostro problema è veramente necessario memorizzarli? La risposta è ovviamente no. Basta generare i fattori primi di entrambi i numeri in parallelo (con la dovuta cura) e accumulare i prodotti dei fattori primi che dividono entrambi i numeri. Dobbiamo distinguere con cura i casi in cui troviamo un primo che entrambi i numeri, nessuno dei due numeri, o solo uno dei due (sono i 4 comandi `if` in Fig. 14). Per mantenere il codice ragionevolmente sintetico in Fig. 14 mi sono permesso di usare l'operatore logico `&&` che restituisce l'and logico tra due operazioni ma avrei potuto farne a meno (vedi esercizi 4.1(5),

```
void stampaDivisoriPrimi(int m){
/* PREC: m>0 */
   int p=2;

    while (p*p<=m){
      if (resto(m,p)) p++;
          else {
            printf(" %d",p);
            m = div(m,p);
          }
    }

    if (m>1) printf(" %d",m);
    printf("\n");
}
```

Figura 14: Stampa dei fattori primi

```
int mcdMaestra(int m, int n){
/* PREC: m,n>0 */
  int p=2;
  int mcd=1;
  while (minore(mult(p,p), max(m,n))
           && minore(p,min(m,n))){
  /* INV: mcd(m0,n0)=mcd*mcd(m,n) */
    if (resto(m,p) && resto(n,p)) p++;
    if (!resto(m,p) && !resto(n,p))
        mcd=mult(mcd,p);
    if (!resto(m,p)) m=div(m,p);
    if (!resto(n,p)) n=div(n,p);
  }
  if (m==n) mcd=mult(mcd,n);
  return mcd;
}
```

Figura 15: MCD (versione della maestra)

4.1(6) e 4.1(7) qui sotto). Lasciamo al lettore il piacere di verificare che la proprietà $\mathsf{mcd}(m_0, n_0) = mcd \times \mathsf{mcd}(m, n)$ è effettivamente un invariante.

Per quanto riguarda la terminazione, quando la guardia è verificata, si ha chiaramente $m, n \geq p$ e quindi anche che la quantità $m + n - p$ è positiva. Inoltre, ad ogni iterazione o viene incrementato p o almeno uno tra m ed n viene diviso per $p > 1$. Quindi, necessariamente si ha $m' + n' - p' < m + n - p$.

Soluzione 3: Già gli antichi greci avevano osservato che se p divide m e n, allora divide anche $m - n$ (o $m + n$). Questa osservazione subito suggerisce le seguenti equazioni induttive (tradizionalmente attribuite ad Euclide) per definire il massimo comun divisore $\mathsf{mcd}(m, n)$ di due numeri strettamente positivi.

$$\begin{array}{rcl} \mathsf{mcd}(n, n) & = & n \\ \mathsf{mcd}(m, n) & = & \mathsf{mcd}(m - n, n) \quad (n < m) \\ \mathsf{mcd}(m, n) & = & \mathsf{mcd}(n - m, m) \quad (m < n) \end{array}$$

Analogamente a quanto fatto per la moltiplicazione egiziana, queste equazioni si traducono abbastanza naturalmente nel programma C in Fig. 16, che rispetto ai precedenti, spicca in eleganza, semplicità ed efficienza.

Problema 3: *Scrivere un programma interattivo che gioca ad* alto-basso.

Soluzione: Ricordiamo brevemente le regole del gioco: il giocatore A sceglie un numero segreto n (in un certo intervallo finito $[a, b]$) e il giocatore B cerca di indovinarlo proponendo al giocatore A un numero t. Il giocatore A deve dire al giocatore

```
int mcdEuclide(int m, int n){
/* PREC: m,n>0 */
    while (m!=n){
      if (minore(m,n)=1)
             n=diff(n,m);
        else m=diff(m,n);
       }
  return m;
}
```

Figura 16: mcd calcolato con l'algoritmo di Euclide

B se il numero proposto è più alto del numero segreto ($t > n$), è più basso ($t < n$) o se il giocatore B ha indovinato ($t = n$). Supponiamo che il numero segreto sia in un intervallo limitato, diciamo tra 1 e 1000. Cercheremo di scrivere un programma che si comporta come il giocatore B e cerca di indovinare il numero segreto del giocatore A (umano) che interagisce da riga di comando.

Cominciamo con l'osservare che i tre possibili risultati di ogni tentativo (più basso, più alto, indovinato) possono essere facilmente codificati rispettivamente con gli interi -1,1 e 0, esattamente come fatto per la funzione `minore`.

Dovrebbe essere subito chiaro che la strategia migliore consiste nel tentare con un numero che è il punto medio dei possibili valori assunti dal numero segreto, cosicché dopo una risposta negativa possiamo scartare metà dell'intervallo in esame. Il programma C corrispondente è riportato in Fig. 17.

In questo programma, facciamo la conoscenza di una nuova istruzione del linguaggio C: il costrutto `break` che interrompe l'esecuzione di un ciclo. C'è anche una analoga istruzione `continue` che salta i successivi comandi all'interno di un ciclo e forza il controllo dell'esecuzione a rivalutare la guardia. Queste istruzioni possono essere molte comode, ma il loro abuso è da evitare, in quanto possono rendere i programmi poco leggibili. Chiudiamo osservando che in questo ciclo, posto esista un numero segreto s, tale che $inf \leq s \leq sup$, questa proprietà si mantiene invariante (a meno che l'operatore non bari ☺) e l'intervallo di ricerca ha dimensione decrescente $sup - inf$.

4.1 Esercizi e Spunti di Riflessione

1. Scrivere un algoritmo per risolvere l'esponenziale, basato sullo stesso concetto della moltiplicazione egiziana (Problema 1). Scrivere la definizione induttiva dell'esponenziale, il codice della funzione, invarianti e precondizioni etc.

2. Quante moltiplicazioni esegue la funzione dell'esercizio precedente? E se invocate la funzione `multiplyingLikeAnEgiptian(int m, int n)`, quante som-

```
#include <stdio.h>
int main(){
  int tentativo, ris;
  int inf = 1;
  int sup = 1000;

  do {
     if (minore(sup,inf)){
        printf("mi stai prendendo in giro.
                    Non gioco piu con te\n");
        break;
     }
     tentativo = div(somma(inf,sup),2);
     printf("%d\n",tentativo);
     scanf("%d",&ris);
     if (minore(0,ris)) inf = tentativo + 1;
     if (minore(ris,0)) sup = pred(tentativo);
  } while (ris!=0);
}
```

Figura 17: Implementazione del giocatore B nell'alto-basso

me? E se avete stratificato il codice a partire dal successore, quante volte si esegue un successore nei due casi?

3. Considerate la funzione `mcdIngenuo`. È possibile riscrivere il ciclo con la guardia `while(1)` e togliere il `return` alla fine della funzione. Perché? Dimostrare che la funzione continua a dare risultati corretti.

4. Considerate la funzione `mcdMaestra`. Dimostrare che all'uscita del `while` m ed n o sono uguali o sono primi tra loro.

5. ♣ Dimostrare che gli operatori logici `&&` (and logico) e `||` (or logico) non sono necessari, nel senso che ogni comando nella forma `if (`$\mathtt{E}_1$ `&&` $\mathtt{E}_2$`)` $\mathtt{C}_1$ `else` $\mathtt{C}_2$ (oppure `if (`$\mathtt{E}_1$ `||` $\mathtt{E}_2$`)` $\mathtt{C}_1$ `else` $\mathtt{C}_2$) può essere trasformato in un comando condizionale equivalente che non fa uso di tali operatori.

6. La semantica della valutazione di un'espressione nella forma $\mathtt{E}_1$ `&&` $\mathtt{E}_2$ può essere informalmente descritta come segue:

   > Se $\mathtt{E}_1$ valuta a un valore interpretato come `false`, restituisci `false`, altrimenti restituisci il valore di verità risultante dalla valutazione di $\mathtt{E}_2$.

   Dare simmetricamente, la semantica dell'operatore `||`. ▸ Considerate infine la seguente funzione `int foo(){ while(1){} return 1;}`. Cosa stampa `if`

`(1 && foo()) printf("pippo") else printf("paperino")`? E `if (1 || foo()) printf("pippo") else printf("paperino")`?

7. ♣★ Supponendo che E_1 e E_2 siano espressioni intere, e considerando la funzione `myAnd` definita sotto, perché la chiamata `myAnd(`E_1`, `E_2`)` non è sempre equivalente a E_1 `&&` E_2?

```
int myAnd(int x, int y){
    if (!x) return 0:
       else return y;
    }
```

   Trarre analoghe conclusioni per l'espressione E_1 `||` E_2.

8. Scrivere l'invariante per il ciclo della funzione `mcdEuclide`. Dimostrare che il ciclo termina.

9. Scrivere tre funzioni C che calcolano il minimo comune multiplo di due numeri. Le funzioni `mcmIngenuo`, `mcmMaestra` e `mcmEuclide` saranno ciascuna l'analogo della corrispondente versione del calcolo del massimo comun divisore visti sopra.

10. ▶ Riscrivere tutte le funzioni di questo capitolo (nel testo o negli esercizi) usando gli operatori predefiniti infissi `+` (al posto dell'invocazione della funzione `somma`) `<=` (al posto dell'invocazione della funzione `minore`), `%` (al posto dell'invocazione della funzione `resto`), `*` (al posto dell'invocazione della funzione `mult`), `/` (al posto dell'invocazione della funzione `div`), etc.

    Scoprire un modo per calcolare i tempi di esecuzione di un programma, e confrontare il tempo di esecuzione di un programma complesso (tipo l'esponenziale o il massimo comun divisore) rispetto alla versione in cui tutte le operazioni aritmetiche e logiche sono state implementate a partire dal successore e dal test di uguaglianza.

11. Scrivete una programma C che gioca il ruolo del giocatore A nel gioco dell'alto basso.

12. Organizzate il codice in modo tale da far giocare il giocatore B (implementato prima) contro il giocatore A (vedi esercizio precedente).