2

IL PRIMO IMPATTO

Lo scopo di questo capitolo è spiegare le operazioni da svolgere per costruire un programma eseguibile, anche del tutto banale. L'attenzione non è rivolta alla programmazione, ma piuttosto all'interazione col sistema operativo e a cosa succede dietro le quinte dei comandi che si devono concatenare per passare dalla scrittura delle istruzioni all'esecuzione del programma.

Va da sé che un tale argomento non si può svolgere in astratto, senza il supporto di un esempio specifico. E un esempio coinvolge necessariamente un linguaggio di programmazione ed un sistema operativo specifici. La scelta indicata fin dall'inizio è: linguaggio di programmazione C con sistema operativo Linux.

2.1 Alcune direttive generali

La preparazione di un programma richiede fondamentalmente tre operazioni:

(i) La scrittura del programma nel linguaggio prescelto (nel nostro caso il linguaggio C); il file che contiene il testo del programma viene chiamato *sorgente* (source) — nel senso che è il punto di partenza del tutto. È comodo dargli un nome della forma $< xxx >$`.c`, perché il tipo `.c` viene riconosciuto dal compilatore come indicatore che il contenuto del file è proprio un sorgente C.

(ii) La compilazione del file sorgente, eseguita dal `compilatore`. Questo passo consiste nella traduzione in linguaggio macchina delle istruzioni contenute nel programma e nella predisposizione della memoria (dati ed istruzioni). In questa fase viene prodotto un secondo modulo (file) detto *rilocabile* o *oggetto* (object). In questo modulo restano indefiniti i riferimenti a moduli esterni.

(iii) La raccolta di tutti i moduli rilocabili che nel loro complesso formano il programma così da costruire un modulo autoconsistente (tutti gli indirizzi sono ben definiti). Questo modo si chiama *assoluto* o *eseguibile*. Quest'ultima operazione viene svolta da un ulteriore programma, detto *linker* (letteralmente: collegatore).

L'operazione (i) si esegue col text editor; le due operazioni (ii) e (iii) possono essere compiute separatamente (quando il programma è complesso e costituito da molti moduli) oppure in una singola operazione.

Alla fine di questa procedura si è costruito un modulo che può essere caricato in memoria e mandato in esecuzione. Naturalmente, ci possono essere degli errori. Non resta che individuarli, e ripartire da capo correggendo il modulo sorgente.

Se non hai mai scritto un programma in vita tua dubito che questo paragrafo ti sia chiaro. Quindi passiamo ai fatti, e rivediamo il tutto con un paio di esempi.

2.2 Il primo programma: sai scrivere?

È un classico: il primo programma consiste nel chiedere al calcolatore di scrivere qualcosa sul terminale. Sembra un'operazione banale – e in fondo lo è – ma se non ci ha mai provato è probabilmente un ostacolo notevole: devi imparare le prime regole di sintassi, come creare un file sorgente, come tradurlo in un programma eseguibile, come mandarlo in esecuzione. Al solito, tante cose che prese singolarmente sono semplici, ma sono tante!... Prendiamola con calma.

2.2.1 Il modulo sorgente

Eccoti il testo del programma; poi verranno i commenti.

```
#include <stdio.h>
int main()
{
  printf("Ciao, vecchio pirata!\n");
  exit(0);
}
```

La prima riga è un ordine al compilatore. Significa: cerca il file `stdio.h` tra i moduli standard che un programmatore può includere, e inseriscilo esattamente a questo punto. L'effetto è che il contenuto di quel file viene sostituito alla prima riga, ed il compilatore lo considera come parte integrante del tuo programma. Fin qui è facile. Ma dove si trova quel file? e cosa contiene?

La risposta alla prima domanda è (quasi) facile: dipende da come è stato installato il sistema. Sui sistemi Linux di solito si trova nel directory `/usr/include/`. La seconda domanda è più problematica. La risposta immediata potrebbe essere: prova a guardarci. Ma forse serve qualche altra spiegazione. Tra poco ci arrivo.

Veniamo alla seconda riga. L'istruzione `int main()` è praticamente obbligata. Significa: qui comincia un modulo di programma il cui nome è `main`. Alla fine dell'esecuzione il modulo restituirà un valore di tipo `int`, un modo complicato per dire che restituirà un numero intero. La parentesi graffa aperta segna l'inizio delle istruzioni che formano il modulo; la parentesi graffa chiusa dell'ultima riga segna la fine del modulo.

C'è ancora una sottigliezza: che ci fanno le parentesi tonde? Semplice: indicano che il nome `main` (come qualunque nome che sia seguito da parentesi tonde) è una funzione, o un modulo di programma. Più avanti imparerai che ci sono un sacco di cose a cui puoi assegnare un nome, ed il compilatore deve pur avere un modo per distiguerle.

Il nome `main` identifica il modulo principale del programma: ogni programma deve contenere necessariamente un modulo (e uno solo) con questo nome. Quando il kernel del sistema attiva il programma passa il controllo della CPU alla prima istruzione di questo modulo.

L'istruzione `exit(0)` della penultima riga significa: io ho finito; torna al sistema, restituendogli il valore 0. Il kernel considera il programma concluso e lo elimina dalla memoria.

L'istruzione `printf("...")` è quella che ordina di scrivere sul terminale. Il contenuto delle parentesi specifica il contenuto della scritta: in questo caso la frase tra virgolette. Lo strano simbolo `\n` che compare alla fine della frase è l'ordine di tornare a capo. Se lo dimentichi, il cursore resta fermo alla fine della riga (prova!).

Avrai anche notato che le istruzioni sono terminate da un punto e virgola, e che sono incolonnate con un certo ordine. L'incolonnamento è solo una questione di estetica e di comodità: le righe spostate a destra di 2 caratteri sono quelle incluse tra due graffe; così diventano più riconoscibili. Il punto e virgola invece è essenziale, perché segnala al compilatore la fine di un'istruzione.

E veniamo alla parte più nascosta di tutta la faccenda. L'istruzione `printf` non è di quelle semplici. Se ricordi quello che ti ho spiegato nel primo capitolo sai che succedono un sacco di cose: il programma deve rivolgersi al kernel per chiedergli di inviare una sequenza di caratteri al terminale; il kernel poi farà quello che deve. L'invio della richiesta, ovviamente, richiede molte istruzioni. Dove stanno?

Provo a spiegarti l'arcano. Il nome `printf` identifica un blocco di istruzioni, denominato *funzione* o talvolta *modulo di libreria*, che esegue tutte le operazioni necessarie per il trasferimento di scritte sul terminale. [1] Proprio per questo il nome `printf` è seguito da una coppia di parentesi tonde.

---

[1] Il termine libreria può sembrare un po' misterioso. In effetti, è la traduzione folcloristica del termine inglese *library*. Sarebbe più appropriato parlare di biblioteca, in quanto si fa riferimento ad un file che contiene una serie di moduli preconfezionati per eseguire operazioni di utilità generale. Purtroppo, il termine è talmente entrato nell'uso che non posso esimermi dall'accettarlo. Se non lo facessi, rischierei di rendere il testo incomprensibile agli esperti — cioè a quelli cui questo testo non è indirizzato.

L'istruzione `printf(...)` è una chiamata a questo modulo, nel senso che il tuo programma deve saltare alla prima riga del modulo `printf`, eseguirlo, e poi riprendere l'esecuzione dalla riga successiva a quella della chiamata.

Il contenuto del modulo `printf` dipende strettamente dal sistema operativo che stai usando (il linguaggio C non è esclusiva dei sistemi Linux!), dall'hardware della macchina, &c. Per questo motivo chi fornisce il compilatore fornisce anche questo modulo, assieme a molti altri che scoprirai strada facendo. Il compilatore non ha bisogno di sapere dove si trovi il modulo, ma ha bisogno di alcune informazioni per controllare la correttezza del riferimento. Queste informazioni si trovano... nel file `stdio.h`. [2] Ecco spiegato – almeno in parte – il perché della prima riga.

A questo punto molte domande dovrebbero aver trovato una risposta. Non ti resta che creare un file che contiene il modulo sorgente del programma. Per questo puoi usare un qualunque text editor. Devi anche scegliere un nome per il tuo programma. Che ne dici di `pirata.c`? [3]

2.2.2 La compilazione

Creato il modulo sorgente, è tempo di passare alla fase successiva: la compilazione, ossia la traduzione del modulo sorgente in codice macchina.

Supponiamo che tu abbia effettivamente assegnato il nome `pirata.c` al file che contiene il modulo sorgente. Lo potrai compilare col comando

```
gcc -o pirata pirata.c
```

Ecco il significato dei campi di questo comando.

- Il campo `gcc` mette in azione il compilatore. [4] Il resto della riga contiene informazioni che vengono utilizzate dal compilatore stesso.
- Il campo `-o pirata` spiega al compilatore che deve produrre un modulo eseguibile col nome `pirata`. In effetti il comando in questa forma significa che la fase di link può essere eseguita immediatamente, senza ulteriori comandi.
- Il campo `pirata.c` passa al compilatore il nome del file che contiene il sorgente del programma. È qui che entra in azione la convenzione secondo la quale il tipo `.c` identifica un sorgente in linguaggio C.

Se il compilatore trova errori li segnala in questa fase, e interrompe l'operazione omettendo la costruzione del programma eseguibile. Errori tipici

---

[2] Il nome `stdio` è un'abbreviazione di standard I/O; il tipo `.h` identifica normalmente un file di sorgente C che contiene istruzioni da includere mediante la riga `#include`, e non moduli completi.

[3] Ricorda: assegnare al file il tipo `.c` non è strettamente obbligatorio, ma conviene se vuoi che il compilatore lo riconosca immediatamente come sorgente C. Il nome invece è del tutto arbitrario.

[4] `gcc` è un acronimo per Gnu C Compiler. Gnu – un acronimo ricorsivo per Gnu's not Unix – è il nome di un progetto per la produzione e la diffusione di software libero.

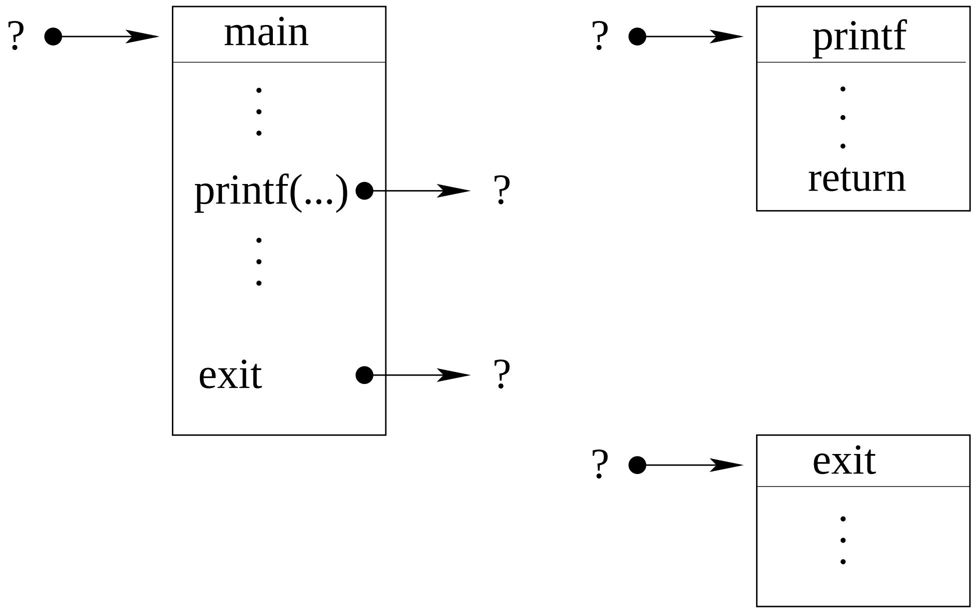

Figura 2.1. I moduli rilocabili necessari per il programma. Ciascuno di essi ha un *entry point*, un punto di rientro (tipicamente alla fine), ed eventualmente dei riferimenti esterni indefiniti.

e frequenti possono essere la dimenticanza di qualche parentesi o punto e virgola. Basta correggerli nel file sorgente, e tutto va a posto. Per errori più complicati, ... beh ... ci vuole pazienza ed esperienza. Tutti i programmatori, vecchi o giovani, ci sono passati.

Ma proviamo a guardare un po' dietro le quinte. La situazione è illustrata in figura 2.1. Il compilatore esegue una traduzione delle istruzioni C in istruzioni di macchina. Ma restano alcuni punti oscuri.

Il primo punto è: come viene attivato il modulo? Il compilatore può solo generare una successione di istruzioni in codice macchina, assegnando provvisoriamente un indirizzo iniziale che è lo stesso per tutti moduli. L'indirizzo della prima istruzione è detto *entry point*, o porta di ingresso del modulo.

Il secondo punto richiede qualche considerazione in più. Ti ho detto poco fa che `printf` non è un'istruzione semplice: è invece un riferimento ad un modulo di libreria. In questo caso il compilatore riesce a fare solo una traduzione parziale. Il nome `printf` resta solo un riferimento ad un modulo che si trova da qualche parte: lo si chiama un *riferimento esterno*. In termini precisi: il compilatore dovrebbe inserire un'istruzione di salto all'indirizzo del modulo `printf`, ma non sa quale sia questo indirizzo, e quindi lo lascia indefinito. Nel linguaggio dei manuali e dei compilatori si usano i termini *undefined reference* o *undefined external*, o qualcosa di simile. Una osservazione analoga si può fare per l'istruzione `exit`: il compilatore dovrebbe richiamare il modulo che esegue la chiusura del programma, ma non sa dove sia, e lascia un riferimento indefinito.

Un modulo che abbia uno o più entry point ed abbia ancora dei riferimenti esterni indefiniti viene detto *modulo oggetto* o *modulo rilocabile*. Il compilatore lo produce talvolta utilizzando lo stesso nome del file sorgente, in altri casi gli dà un nome costruito in modo casuale, ma assegnandogli in ogni caso il tipo `.o` anziché `.c`. La parola passa al linker.

Ultima nota: ma perché si chiama modulo rilocabile? Semplice: il compilatore non sa in quale zona di memoria verrà sistemato il modulo dentro l'eseguibile; quindi può solo assegnare ad istruzioni e dati degli indirizzi provvisori, che dovranno essere assegnati in modo definitivo dal linker. Torneremo su questo punto.

2.2.3 Il link

Il compito del linker è la raccolta di tutti moduli rilocabili necessari per costruire il programma eseguibile. Se il comando che hai dato è quello indicato sopra il linker viene attivato automaticamente, ed intende che deve iniziare con un file che si chiama `pirata.o` o ⟨*qualche cosa*⟩`.o` — quello prodotto dal compilatore.

Esaminando questo file il linker si accorge che c'è un entry point, identificato dal nome `main`, e dei riferimenti esterni non definiti — ad esempio `printf`.

L'entry point si sistema facilmente: poiché il suo nome è `main`, deve essere la prima istruzione eseguibile del programma, e quindi deve essere richiamato dal kernel.

Per i riferimenti esterni invece il linker deve andare a cercare dei moduli che li definiscano. In termini precisi: per ciascun riferimento esterno, identificato da un nome, deve cercare un modulo che abbia come entry point lo stesso nome. Dove lo cerca? Il comando che hai dato non contiene altre indicazioni; quindi il linker intende che i moduli necessari si trovano nella libreria C standard. Questa libreria contiene i moduli di uso più comune; proprio per questo viene usata automaticamente, senza bisogno di specificarla.

Nella libreria il linker trova il modulo `printf`, lo aggiunge al modulo `pirata.o`, e completa il riferimento. In termini precisi, inserisce l'indirizzo della prima istruzione del modulo `printf` nel punto in cui `pirata.o` effettua la chiamata. Analogamente, carica il modulo di chiusura del programma, che in figura 2.1 è stato chiamato `exit`, e completa il riferimento in `pirata.o`. Procede allo stesso modo per tutti i riferimenti esterni.

Durante la raccolta il linker provvede anche ad assegnare a ciascun modulo la zona di memoria dove andrà a risiedere ed a definire correttamente gli indirizzi di memoria assegnati provvisoriamente del compilatore. Il nome *assoluto* che viene talvolta usato per identificare il modulo eseguibile dipende proprio dal fatto che la rilocazione degli indirizzi è stata eseguita.

Il risultato è illustrato in figura 2.2. Una volta soddisfatti tutti i riferimenti esterni il linker ha finito il suo lavoro: prende tutti moduli rilocabili che ha raccolto e collegato tra loro e li riunisce in un unico modulo che scrive

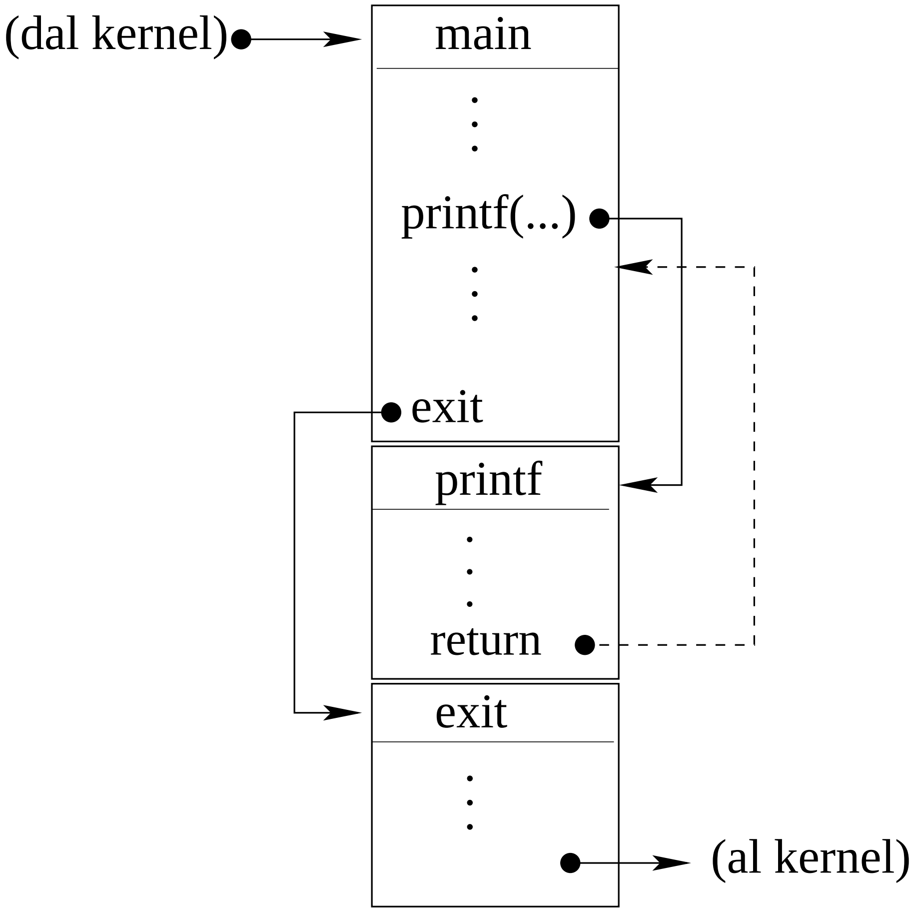

Figura 2.2. Il modulo eseguibile: tutti i moduli rilocabili sono stati sistemati in sequenza, e tutte le chiamate sono state correttamente definite. Il percorso del `return` è tratteggiato perché non è stabilito in modo univoco: l'indirizzo di rientro deve essere definito durante l'esecuzione, perché la stessa funzione potrebbe essere richiamata in più punti diversi.

sul file `pirata` (come gli hai ordinato); poi cancella il file ⟨*qualche cosa*⟩`.o` che ormai è diventato un oggetto inutile.

Anche il linker, ovviamente, può segnalare errori. Il caso più frequente è quello di riferimenti esterni rimasti indefiniti. Ad esempio, se per caso hai scritto `pritnf` invece di `printf` il linker ti segnalerà che c'è il nome `pritnf` che lui proprio non sa dove trovare. In questo caso il file eseguibile non viene creato: non potrebbe comunque funzionare.

2.2.4 L'esecuzione

Non resta che passare al momento più emozionante: l'esecuzione. Basta battere il comando

```
./pirata
```

Significa: cerca nel mio directory corrente il file col nome `pirata`. Questo contiene un modulo eseguibile, e io voglio che tu lo mandi in esecuzione.

Se tutto va bene, ti comparirà sullo schermo la frase

```
Ciao, vecchio pirata!
```

In caso contrario, beh, ... non ti resta che cercare l'errore. Ripercorri tutto il lavoro che hai fatto controllando punto per punto. Se proprio non ne esci, chiedi aiuto a chi ci è già passato. Non farti crucci inutili: i programmatori bravi non sono quelli che non fanno errori. Sono quelli che ne hanno fatti più degli altri, ma li hanno anche scoperti ed hanno imparato, se non ad evitarli, almeno a riconoscerli e correggerli.

2.3 Il secondo programma: sai anche leggere?

Superato lo scoglio del primo programma, passiamo al secondo. Questa volta proviamo ad interagire col programma con una domanda ed una risposta. Il programma deve:

i. chiederti di scrivere due numeri interi;
ii. leggerli e memorizzarli da qualche parte;
iii. calcolarne la somma;
iv. scrivere sul terminale il risultato.

Che significa "memorizzarli da qualche parte"? E come faccio a dirgli di calcolare la somma? Leggerli, poi, questi numeri! ...

2.3.1 Il modulo sorgente

Anche in questo caso inizio con lo scriverti il testo del programma. I commenti verranno dopo.

```
#include <stdio.h>
int main()
{
  int j,k;
  printf("Dammi due numeri: ");
  scanf("%d %d",&j,&k);
  printf("%d + %d = %d\n",j,k,j+k);
  exit(0);
}
```

Lo scheletro del programma, come vedi, è rimasto lo stesso: un modulo `main` che inizia alla seconda riga e si chiude con la graffa dell'ultima riga. L'ultima istruzione è ancora `exit(0)`, alla penultima riga.

La novità comincia con la terza riga. In linea di principio so come memorizzare dei dati: basta spiegare alla CPU che deve scrivere un numero in un cassetto di memoria. Ma quale cassetto? E ne basta uno? La terza riga risponde proprio a queste domande. L'istruzione `int j,k;` significa: voglio memorizzare due dati, che sono dei numeri interi; li chiamerò `j` e `k`. Il compilatore provvede a riservare due celle di memoria di lunghezza sufficiente per contenere i miei dati, e le userà tutte le volte che io farò riferimento ai nomi `j` o `k`. A che indirizzo di memoria si trovino io non lo so, e in fondo non mi importa saperlo: mi basta che il compilatore aggiusti tutto in modo consistente.

L'istruzione `printf("Dammi due numeri: ");` non dovrebbe presentarti difficoltà, dopo aver visto ed eseguito il primo programma. Ti faccio notare solo che manca il `\n` alla fine della scritta: è più elegante lasciare che il cursore resti sulla stessa riga ad aspettare la risposta.

L'istruzione `scanf(...)` è nuova: è palesemente una chiamata ad una funzione, ma sembra piuttosto complicata. Vediamola in dettaglio.

- `scanf` significa *scan formatted*. Scan significa che la funzione deve catturare uno alla volta una serie di caratteri dallo schermo ed interpretarli. Formatted significa che io scriverò i numeri in forma decimale, come faccio nella vita di tutti i giorni, e tradurli in binario sarà affar suo. Va da sé che `scanf` è il nome di un modulo esterno (è seguito da una coppia di parentesi tonde!), che il linker dovrà cercare nella libreria standard del linguaggio C. Tutto questo te l'ho già spiegato.
- Tra parentesi ci sono gli *argomenti* della funzione, separati da virgole. Il primo argomento è la scritta compresa tra virgolette; si dice di solito: una *stringa di caratteri*. Il secondo ed il terzo argomento sono `&j` e `&k`. Gli argomenti servono per trasferire informazioni tra il modulo chiamante e la funzione chiamata.
- Tutto quello che è contenuto tra virgolette è il cosiddetto *formato di conversione*. Nota che compare due volte la coppia di caratteri `%d`, con uno spazio in mezzo. La funzione `scanf` usa queste informazioni per effettuare la conversione dei dati. Il carattere `%` significa: a partire da questo momento devi interpretare i caratteri che vengono battuti. Il carattere `d` spiega come interpretarli: si tratta di un numero intero che la funzione deve convertire in binario e memorizzare. La coppia `%d` è ripetuta due volte perché voglio che legga due numeri; lo spazio inserito significa che i due numeri verranno battuti separati da uno o più spazi. Alla fine dell'inserimento dei dati occorrerà battere il tasto `<Return>` (o `<Enter>`, o `<Invio>`, dipende dalla tastiera)
- Tutto quello che viene dopo la virgola spiega dove debbano essere memorizzati i numeri che ha letto. Significa che il primo numero deve essere memorizzato nel cassetto che ho chiamato `j`, il secondo nel cassetto che ho chiamato `k`. Ma fa attenzione: non ho scritto semplicemente `j,k`; ho scritto invece `&j,&k`. Perché? La spiegazione sta in una regola generale, che coinvolge tutte le funzioni, compresa `printf`. Te la spiego tra poco.

L'istruzione `printf(...)` della sesta riga scrive il risultato. Come vedi, è un po' più complessa di quella della quarta riga: gli argomenti della funzione sono diventati quattro, il primo dei quali è un *formato di conversione* che ricorda quello della chiamata a `scanf`. Entriamo nei dettagli.

- La funzione `printf` prende il formato di conversione e fa scorrere un carattere per volta. Fin che non trova il carattere `%` si limita a copiare sul terminale. Appena trova il carattere `%` intende che c'è un dato da convertire. I dati da convertire sono gli argomenti successivi, che vengono presi nell'ordine in cui si trovano: ogni carattere `%` corrisponde ad un dato. Il carattere `%` deve essere seguito da altri caratteri che specificano che tipo di conversione fare.
- Nel nostro caso, la funzione `printf` trova, nell'ordine: `%d` che significa prendi l'argomento `j` e convertilo in formato decimale; i caratteri “ `+` ” (spazi compresi) che devono essere semplicemente trasferiti su terminale; un altro `%d` che forza la conversione dell'argomento `k`; i caratteri “ `=` ” da trasferire tali e quali su terminale; un ultimo `%d` che forza la conversione dell'argomento `j+k` (la somma la fa lui prima di passare il dato a `printf`); un `\n` che ordina di andare a capo.

Resta da spiegare il perché in scanf ho scritto `&j` mentre in printf ho scritto solo `j`. Questo è un po' delicato, quindi ti devo chiedere un momento di particolare attenzione.

Come ti ho spiegato, la memoria è simile ad una cassettiera numerata; ogni cassetto ha un'etichetta che riporta il suo *indirizzo*, e contiene un `dato`. Nel mio programma ho usato il nome `j` per identificare un *dato* intero, cioè il contenuto del cassetto. Per essere più precisi: la dichiarazione `int j` della terza riga spiega al compilatore che `j` è il nome che io voglio assegnare ad un dato intero. La scrittura `&j` indica l'indirizzo del cassetto che contiene il dato `j`. Sembra complicato, ma lo è.

Quando chiamo una funzione passandole un argomento posso comunicarle il dato, senza rivelarle in che cassetto si trova, oppure l'indirizzo. Nel primo caso si dice che l'argomento è passato *per valore*, nel secondo caso che è passato *per indirizzo*. Se devo stampare il valore di un numero non serve sapere l'indirizzo a cui si trova; quindi la funzione `printf` è stata programmata in modo da ricevere solo il dato, che io passo col suo nome `j`. Se devo ricevere un dato la cosa più semplice è dire alla funzione: “mettilo nel tal cassetto”, e passarle l'indirizzo. La funzione `scanf` mi deve restituire un dato (quello che ha letto); quindi devo passarle l'indirizzo `&j`. Tutto qui. [5]

---

[5] Forse ti domanderai: ma perché tutta questa complicazione? non bastava passare sistematicamente l'indirizzo? Certamente, ed infatti questa è la convenzione del linguaggio FORTRAN, ad esempio. Ma il passaggio dell'indirizzo comporta il rischio che il contenuto del cassetto possa essere modificato dalla funzione, ad insaputa del programmatore. Un errore di questo genere è estremamente pericoloso e difficile da scovare. Il meccanismo di chiamata per valore protegge, almeno in parte, da questo rischio. Se una funzione deve modificare un dato del modulo chiamante, il programmatore ne deve essere informato, e deve passare l'indirizzo del dato.

2.3.2 Compilazione, link , esecuzione

A questo punto non resta che creare un file che contenga il sorgente del programma, assegnandogli un nome. Potremmo anche chiamarlo `somma.c`, dimostrando scarsa fantasia.

Per la compilazione ed il link del programma il comando è ancora lo stesso, mutatis mutandis:

```
gcc -o somma somma.c
```

e non dovrebbero servire altri commenti, ma un consiglio potrebbe essere utile. A mia esperienza capita spesso di dimenticare di specificare l'indirizzo, e non il dato, nella chiamata alla funzione `scanf`. Un buon modo per accorgersene è usare il comando

```
gcc -Wall -o somma somma.c
```

Il campo aggiuntivo `-Wall` significa: voglio che tu mi avverta anche dei possibili errori, per quanto minimi. Ad esempio, se scrivo `j` invece di `&j` nella chiamata a `scanf` il compilatore mi avverte col messaggio

```
somma.c:5: warning: format argument is not a pointer (arg 2)
```

Traduzione dal computerese: nel file `somma.c`, alla riga 5, c'è qualcosa che puzza di errore: il secondo argomento dovrebbe essere un indirizzo *(pointer)*, ma non lo è. Sei sicuro? Il messaggio viene classificato come `warning` (avvertimento), e non come `error`, perché l'istruzione potrebbe essere corretta, ma tocca al programmatore deciderlo. Nel dubbio, il compilatore costruisce lo stesso l'eseguibile, ma l'esecuzione potrebbe riservare sorprese (prova!).

E veniamo all'esecuzione. Ecco un esempio di cosa dovrebbe comparire sul terminale ad esecuzione conclusa:

```
[...]$ gcc -Wall -o somma somma.c
[...]$ ./somma
Dammi due numeri: 13 17
13 + 17 = 30
[...]$
```

Se ci sono degli errori,... vedi le ultime righe del paragrafo 2.2.4.