Linguaggio C

Appunti per il corso di Laboratorio di Algoritmi e Strutture Dati

Stefano Aguzzoli

Alcune note introduttive

- Orario lezioni: Martedì: 18:30 – 21:30.
  Lezioni frontali in Aula 202,
  Esercitazioni in laboratorio in Aula 307,
  Settore Didattico, Via Celoria

- Frontali: 2,9,23 ottobre; 6,20 novembre; 4,18 dicembre.

- Laboratorio: 16,30 ottobre; 13,27 novembre; 11 dicembre; 8,15 gennaio.

- Eventuali Variazioni saranno comunicate per tempo sul sito: `http://homes.di.unimi.it/~aguzzoli/algo.html`.

- Orario ricevimento: Mercoledì: 15:00 - 16:00

- Lucidi: in pdf su: `http://homes.di.unimi.it/~aguzzoli/algo.html`

- Per supporto e altri strumenti relativi al C consultare anche il sito: `http://www.algoteam.di.unimi.it`

Programma del corso

Il corso verterà sull'insegnamento del linguaggio C e sull'uso del C per implementare strutture dati e algoritmi.

Il linguaggio C:

- Introduzione al C, panoramica sul linguaggio.
- Elementi lessicali, Tipi di dati fondamentali.
- Espressioni. Flusso del controllo.

- Funzioni
- Array, puntatori e stringhe
- Strutture e Unioni
- Strutture dati evolute
- Input e Output, C e UNIX
- Libreria Standard

Testi consigliati

- *Programmazione in C*
  (C Programming: A Modern Approach (second edition)),
  K. N. King.
  W. W. Norton & Company, 2008.

- *C Didattica e Programmazione* (A book on C, 4^{th} edition),
  Al Kelley, Ira Pohl. Pearson, Italia, 2004.

- *Linguaggio C* (seconda edizione),
  Brian W. Kernighan, Dennis M. Ritchie.
  Nuova edizione italiana, Pearson, Italia, 2004.

Compilatori

Useremo solo compilatori aderenti allo standard ANSI.

Per Linux: `gcc`.
Già installato sulla maggior parte delle distribuzioni.

Per Windows: `gcc`.
`MinGW` contiene `gcc`.
Una “distribuzione” facile da installare e usare: `cs1300`.
Scaricatela da
`http://homes.di.unimi.it/~aguzzoli/algo.htm`
o da `http://www.cs.colorado.edu/~main/cs1300`

Modalità d'esame

- viene proposta la specifica di un problema da risolvere mediante tecniche viste a lezione
- avete alcuni giorni per sottoporre una soluzione in C (sorgente gcc-compilabile) e un annesso tecnico
- controllo di correttezza/originalità
- colloquio

Voto finale combinato di **corso+laboratorio**

Consigli per il progetto d'esame

- i temi riguardano problemi computazionalmente difficili: l'uso improprio delle strutture dati comporta un voto basso

- l'interfaccia utente è quasi nulla

- la portabilità è un fattore importante

- l'annesso tecnico è importante

- i commenti al codice sono importanti

Cosa è il C.

- Cenni Storici.
  - Progettato nel 1972 da D. M. Ritchie presso i laboratori AT&T Bell, per poter riscrivere in un linguaggio di alto livello il codice del sistema operativo UNIX.
  - Definizione formale nel 1978 (B.W. Kernighan e D. M. Ritchie)
  - Nel 1983 è iniziato il lavoro di definizione dello standard (ANSI C) da parte dell'American National Standards Institute.
  - Standard ANSI (ISO C89) rilasciato e approvato nel 1990.

Alcune caratteristiche (positive) del C

- Elevato potere espressivo:
  - Tipi di dato primitivi e tipi di dato definibili dall'utente
  - Strutture di controllo
  (programmazione strutturata, funzioni e procedure)

- Caratteristiche di basso livello
  (gestione delle memoria, accesso diretto alla rappresentazione, puntatori, operazioni orientate ai bit)

- (N.B. Stiamo parlando di un linguaggio imperativo e per la programmazione strutturata: non si parla proprio di programmazione orientata agli oggetti).

- Stile di programmazione che incoraggia lo sviluppo di programmi per passi di raffinamento successivi (sviluppo top-down)

- Sintassi definita formalmente. Linguaggio *piccolo*

- Codice efficiente e compatto

- Ricche librerie (standard) per operazioni non definite nel linguaggio (input/output, memoria dinamica, gestione stringhe, funzioni matematiche, ...)

- Rapporto *simbiotico* col sistema operativo (UNIX)

Lo standard ANSI:

- standard chiaro, consistente e non ambiguo
- modifica alcune regole del C tradizionale (Kernighan e Ritchie, K&R) e ne stabilisce altre.
- compromesso che migliora la portabilità mantenendo alcune caratteristiche machine-dependent.
- C++ si basa sullo standard ANSI C.

Alcuni difetti:

- La sintassi non è immediatamente leggibile e a volte ostica. (vedremo cosa succede quando si combinano array, puntatori, puntatori a funzione, ...)
- La precedenza di alcuni operatori è “sbagliata”.
- E’ possibile scrivere codice estremamente contorto.

Lo standard o gli standard ?

Il processo di standardizzazione non è terminato, e andrà avanti fino a quando il C si continuerà ad usare e presenterà aspetti migliorabili.

Quando parliamo di standard ANSI in questo corso ci riferiremo sempre allo standard ISO C89. Il corso illustrerà le caratteristiche del C conformi a ISO C89.

I compilatori, anche `gcc`, contemplano anche altri standard: estensioni successive di ISO C89.
Le caratteristiche del linguaggio aggiunte negli standard successivi non sono fondamentali per un utilizzo pienamente soddisfacente del C nell'implementazione di algoritmi.
Vedremo fra poco alcune opzioni di `gcc` relative alla gestione degli standard e delle varianti del linguaggio.

Una prima panoramica sul linguaggio C

Salve mondo!

Tradizionalmente, il primo programma C è il seguente:

```
/* prog1.c */
#include <stdio.h>

int main(void)
{
    printf("Salve mondo\n");
    return 0;
}
```

Il C è un linguaggio *compilato*.
Per compilare il programma dell'esempio:

```
gcc -o prog1 prog1.c
```

`gcc -o prog1 prog1.c`

`gcc` è il comando che richiama il compilatore.
`-o prog1` è un'opzione di `gcc` che specifica il nome dell'eseguibile.
`prog1.c` è il nome del file contenente il programma
I nomi dei file contenenti codice sorgente C hanno estensione `.c`

Se la compilazione ha successo viene prodotto un file eseguibile:
`prog1.exe` sotto Windows, `prog1` sotto Linux.

Per eseguire il programma:

`prog1` sotto Windows, `./prog1` sotto Linux.

Il programma stampa sul video la scritta

`Salve mondo!`

Analisi del programma:

```
/* prog1.c */
```

Commenti: I commenti in C iniziano con `/*` e finiscono con `*/`, tutto ciò che questi due delimitatori contengono è ignorato dal compilatore

```
#include <stdio.h>
```

Preprocessore: Le righe il cui primo carattere non di spaziatura è `#` sono *direttive* per il **preprocessore**. In questo caso si richiede di espandere sul posto il contenuto del file di intestazione `stdio.h`. Tale file contiene il *prototipo* della funzione `printf`.

```
int main(void)
{
```

Definizione di funzione: Si definisce la funzione `main` come una funzione che non ha alcun parametro formale (`void`), e che restituisce un valore di *tipo* intero `int`.

Ogni programma C definisce una funzione `int main(...)`: l'esecuzione del programma partirà invocando `main`, a sua volta `main` richiamerà altre funzioni e così via. Come vedremo, `main` può essere invocata con parametri, specificandone opportunamente la definizione prototipale.

Il *corpo* della funzione è racchiuso tra le parentesi graffe { e }.

```
    printf("Salve mondo\n");
```

Funzione di libreria standard: `printf` è una funzione contenuta nella libreria standard del C: per informare il compilatore del suo uso corretto in termini di numero e tipo dei parametri e del tipo di ritorno, bisogna includere il file di intestazione `stdio.h`.
`printf` realizza un output formattato. `\n` è il carattere *newline*.
Il carattere `;` marca la fine di un istruzione.

```
    return 0;
```

Istruzione `return`: L'istruzione `return` *expr* ritorna il valore di *expr* alla funzione chiamante. In questo caso, poiché `main` deve ritornare un intero (perché ?), si ritorna 0 al sistema operativo.
`return 0` può essere omesso al termine della definizione di `main`.

Sistema C e organizzazione del codice

Il sistema C è formato dal linguaggio C, dal preprocessore, dal compilatore, dalle librerie e da altri strumenti di supporto.

Un programma C è costituito da un insieme di definizioni di funzioni. L'esecuzione parte dalla definizione della funzione `int main(...)`.

Un programma C può essere diviso su diversi file.

La produzione dell'eseguibile avviene in 3 fasi:
Nella prima fase, ogni file da compilare è trattato dal **preprocessore**. Le direttive al preprocessore determinano modifiche al file che le contiene, quali inclusione di file, espansione di macro, compilazione condizionale.

Prova: `gcc -E prog1.c > vedi` per vedere l'output del preprocessore.

Nella seconda fase ogni file da compilare preprocessato origina un file in formato *oggetto*, non ancora eseguibile.

Le librerie sono molto simili a file oggetto: infatti contengono la versione oggetto di moltissime funzioni predefinite.

La terza fase, a carico del *linker*, vede la produzione di un unico file eseguibile a partire dai file oggetto aggiornati e dalle parti necessarie delle librerie.

I file di intestazione (*name*`.h`) contengono le definizioni necessarie a ogni file sorgente per essere compilato in formato oggetto in modo tale da poter essere successivamente collegato agli altri file che fanno uso di definizioni condivise. Lo stesso discorso si applica all'uso di funzioni di librerie.

Nel nostro esempio precedente, la riga

```
#include <stdio.h>
```

è una direttiva al preprocessore che richiede l'inclusione del file di intestazione `stdio.h`.
L'effetto di tale inclusione è il rimpiazzamento nel file `prog1.c` della riga contenente la direttiva con il contenuto del file `stdio.h` stesso.
Tale file contiene definizioni necessarie alle funzioni di input/output della libreria standard, e di altre di uso comune. Ad esempio vi è il prototipo:

```
int printf(const char *format,...);
```

che informa il compilatore della natura degli argomenti di `printf` e di ciò che tale funzione può ritornare.

Il file `stdio.h` contiene non solo prototipi di funzione, ma in genere definisce elementi quali definizioni di tipi, di costanti, di macro, direttive al preprocessore. Ad esempio vi è la definizione della costante numerica `EOF` che rappresenta, in opportune situazioni, la fine di un file. Tipicamente:

```
#define EOF     (-1)
```

Le parentesi angolari < e > che racchiudono il nome del file da includere, comunicano al preprocessore di cercare il file in una serie di locazioni (directory) standard, in un ordine prefissato. Se si fosse scritto

```
#include "stdio.h"
```

la ricerca del preprocessore sarebbe iniziata dalla directory corrente.

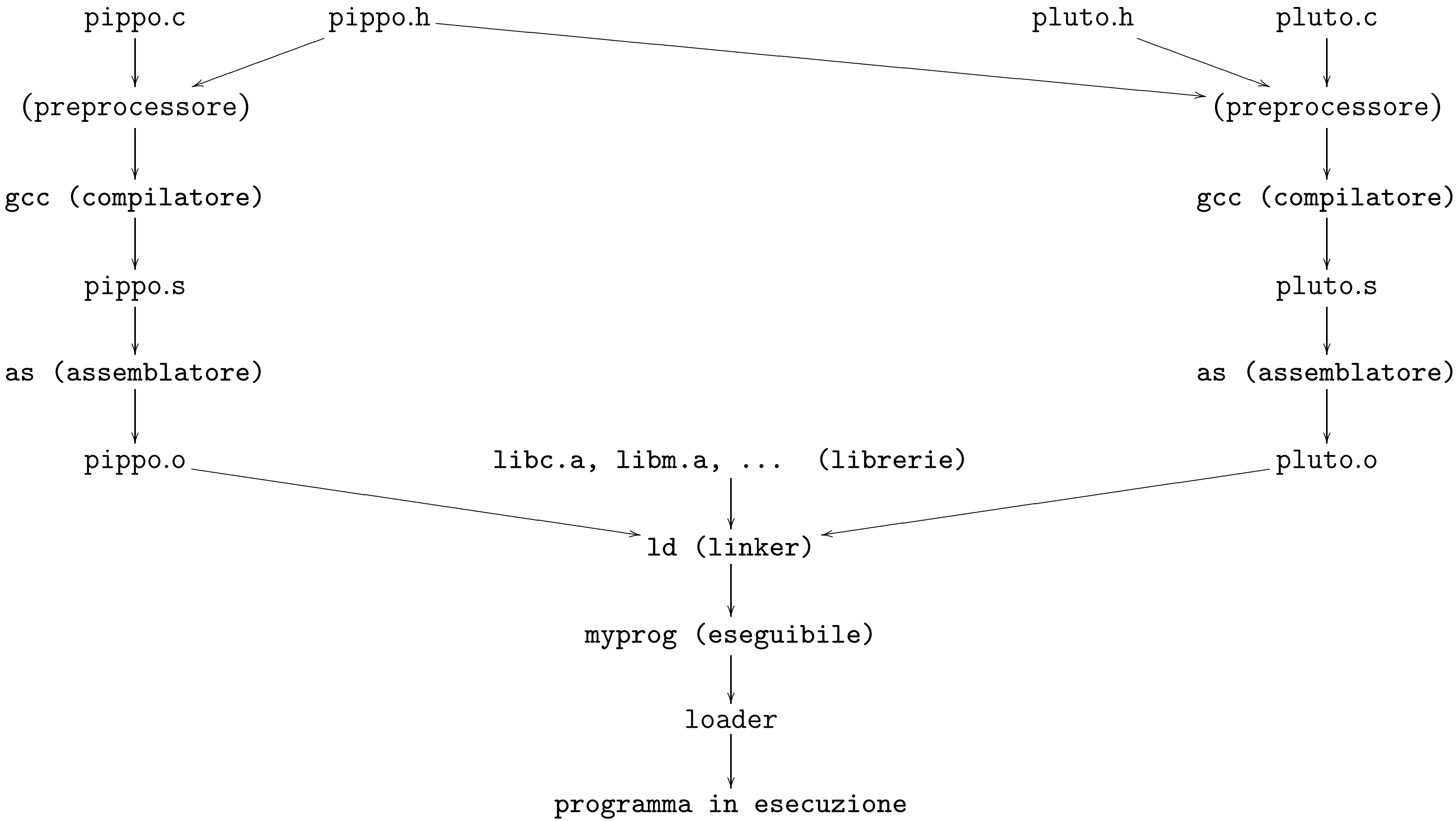
ESEMPIO:

Si supponga di compilare il progetto costituito dai due file `pippo.c`, `pluto.c` con i due file di intestazione `pippo.h` (supponiamo incluso da entrambi i due file `.c`) e, `pluto.h` (incluso solo da `pluto.c`):

Il progetto si compila con il comando `gcc -o myprog pippo.c pluto.c`

Si noti che non si devono specificare i nomi dei file di intestazione: essi sono inclusi dalle direttive al preprocessore `#include` contenute nei file sorgente `pippo.c`, `pluto.c`.

Ecco, nel prossimo lucido, una descrizione diagrammatica del processo di compilazione completo, dalla fase di preprocessamento all'esecuzione.


pippo.c
pippo.h
pluto.h
pluto.c
(preprocessore)
(preprocessore)
gcc (compilatore)
gcc (compilatore)
pippo.s
pluto.s
as (assemblatore)
as (assemblatore)
pippo.o
libc.a, libm.a, ... (librerie)
pluto.o
ld (linker)
myprog (eseguibile)
loader
programma in esecuzione

Alcune utili opzioni di `gcc`

`gcc` contempla diverse opzioni per controllare l'aderenza dei codici sorgenti allo standard ANSI ISO C89, o a sue successive estensioni.

Usare `-ansi` o `-std=c89` per controllare l'aderenza allo standard ISO C89.

Con queste opzioni `gcc` rimane ancora un po' *permissivo* e non si *lamenta* di alcuni usi non conformi allo standard (es. dichiarazioni inframmezzate a istruzioni, array con dimensioni non note a tempo di compilazione. etc...).

Usare `-pedantic` per forzare un controllo di stretta aderenza.

Usare `-Wall` per avere un rapporto su usi del linguaggio permessi ma che frequentemente sono abbinati a errori concettuali nella stesura del codice.

I messaggi segnalati da queste opzioni sono solo degli avvertimenti: la compilazione non si blocca a causa di questi problemi, e, a meno di altri errori, produce infine l'eseguibile.

È bene considerare sempre gli avvertimenti sollevati a causa di frammenti di codice non conformi allo standard, poiché tali frammenti pregiudicano la portabilità del codice.

Gli avvertimenti sollevati da `-Wall` vanno di volta in volta valutati confrontandoli con le intenzioni del programmatore: non sempre sono errori, ma usi intenzionali di alcune caratteristiche del C.

Prova a compilare i sorgenti allegati `warn1.c, ..., warn4.c`

Vediamo un altro esempio:

```
/* prog2.c */
#include <stdio.h>

#define LIRE_PER_EURO   (1936.27)

int main(void)
{
    const int n_euro = 5;
    double n_lire = n_euro * LIRE_PER_EURO;

    printf("%d euro valgono %.3f lire\n",n_euro,n_lire);
    return 0;
}
```

Lo si compili con

```
gcc -o prog2 prog2.c
```

Analisi del programma:

```
#define LIRE_PER_EURO   (1936.27)
```

Definizione di macro: La direttiva `#define` definisce una *macro*. Il preprocessore sostituirà dovunque nel file appaia la stringa `LIRE_PER_EURO` con la stringa `(1936.27)`. Le parentesi non sono obbligatorie nelle definizioni di macro, ma spesso sono necessarie per evitare alcuni effetti collaterali.

```
    const int n_euro = 5;
    double n_lire = n_euro * LIRE_PER_EURO;
```

Definizione di variabili: Queste righe definiscono due *variabili* locali alla funzione `main`.
La variabile `n_euro` è dichiarata di tipo `int`. Il tipo `int` è uno dei tipi base fondamentali del C.

Una variabile di tipo `int` può contenere valori interi positivi o negativi compresi in un intervallo dipendente dall'implementazione.
`n_euro` è inizializzata con il valore intero 5.

La variabile `n_lire` è dichiarata di tipo `double`. Il tipo `double` è uno dei tipi base fondamentali. Una variabile di tipo `double` può contenere valori *in virgola mobile* positivi o negativi in *doppia precisione*. Per valori in *singola precisione* esiste il tipo `float`.
`n_lire` è inizializzata con il risultato dell'espressione del membro destro.

Qualificatore `const`: La parola chiave `const` qualifica la variabile `n_euro` come costante. Il valore di `n_euro` non potrà essere cambiato nel corso del programma.
Nota: la moltiplicazione di `int` per `double` è automaticamente convertita in `double`.

```
printf("%d euro valgono %.3f lire\n",n_euro,n_lire);
```

Output formattato La funzione `printf` della libreria standard viene usata per stampare una stringa sullo schermo.

Il primo argomento di `printf` è una *stringa di formato* che specifica il numero e il tipo dei successivi argomenti. Se il tipo del $(k+1)$esimo argomento di `printf` combacia (o è automaticamente convertibile) col tipo richiesto dal $k$esimo argomento della stringa formato, allora il valore di tale argomento viene sostituito nella stringa che verrà stampata.

Gli argomenti della stringa formato, nella loro forma più semplice, vengono specificati dal carattere % seguito da un carattere che ne determina il significato.

Gli argomenti più comuni della stringa formato sono:

`%d` — intero decimale (vale a dire, in base 10)

`%c` — carattere

`%u` — intero decimale senza segno

`%s` — stringa (le vedremo)

`%x` o `%X` — intero esadecimale (base 16: cifre:0,1,...,9,a,b,...,f)

`%f` — numero in virgola mobile

`%e` o `%E` — numero in virgola mobile in notazione scientifica

`%g` o `%G` — numero in virgola mobile, il più corto tra `%e` e `%f`.

E' possibile specificare l'ampiezza e la precisione dei campi da stampare (esempio: `%.3f` specifica una precisione di 3 cifre decimali per il numero in virgola mobile).

Consideriamo ora il seguente programma organizzato su 3 file:

```
/* prog3.h */
#include <stdio.h>
#define LIRE_PER_EURO   (1936.27)

int geteuro(void);
void putlire(int, double);

/* prog3.c */
#include "prog3.h"

int main(void)
{
    int n_euro = geteuro();
    double n_lire = n_euro * LIRE_PER_EURO;

    putlire(n_euro,n_lire);
    return 0;
}
```

```c
/* aux3.c */
#include "prog3.h"

int geteuro(void)
{
    int ne;

    printf("Immetti numero di euro:\n");
    scanf("%d",&ne);
    return ne;
}

void putlire(int euro, double lire)
{
    printf("%d euro valgono %.3f lire\n",euro,lire);
}
```

Analisi del programma:

Il programma `prog3` è suddiviso su tre file.

- `prog3.h` è un file di intestazione che contiene le definizioni necessarie agli altri due file per poter essere collegati tra loro e con le funzioni di libreria utilizzate.

  In particolare, `prog3.h` include a sua volta il file `stdio.h`. Inoltre contiene la definizione della macro `LIRE_PER_EURO` e i prototipi delle due funzioni presenti in `aux3.c`.

  In questo modo, quando il compilatore trasformerà `prog3.c` in codice oggetto `prog3.o`, saprà quanti parametri e di quale tipo necessitano le funzioni usate in questo file ma definite in altri file. Inoltre conoscerà il tipo di ritorno di queste funzioni.

Notare che nei prototipi di funzione non è necessario riportare alcun nome per i parametri formali: tutto ciò che serve al compilatore è il loro tipo.

- **prog3.c** include il file di intestazione **prog3.h**.

  Questo file contiene un'unica funzione: **main**, che richiama le altre definite altrove.

  La funzione priva di argomenti **geteuro** restituisce un intero. Tale valore intero viene utilizzato per inizializzare la variabile **n_euro**.

  La funzione **putlire** non restituisce alcun valore, e necessita di due argomenti: il primo di tipo **int** e il secondo di tipo **double**.

- `aux3.c` contiene le definizioni delle due funzioni `geteuro` e `putlire`. Inoltre include il file di intestazione `prog3.h`.

  Nella funzione `geteuro` viene utilizzata la funzione `scanf` della libreria standard, il cui prototipo è in `stdio.h`. Tale funzione controlla l'input in dipendenza da una stringa di formato, che stabilisce che tipo di valori siano da leggere dall'input. `scanf` è duale di `printf`. La stringa formato determina numero e tipo dei valori da leggere, che vengono assegnati alle corrispondenti variabili presenti come argomenti successivi nell'invocazione di `scanf`, purché i tipi coincidano.

  Nel nostro esempio: `scanf("%d",&ne);` viene chiesto di leggere un valore intero decimale e di porre il valore letto nella variabile intera `ne`.

`&ne` applica l'operatore *indirizzo di ...* alla variabile `ne`. Infatti a `scanf` viene passato l'*indirizzo* della variabile, non la variabile stessa.

Il tipo dell'indirizzo di una variabile `int` è `int *`, vale a dire *puntatore a* `int`. Come vedremo, in C si fa largo uso esplicito dei puntatori.

L'utilizzo di parametri di tipo puntatore in `scanf` costituisce il modo standard del C per realizzare passaggi di parametri per referenza.

Infatti l'unica modalità nel passaggio di parametri realizzata direttamente nel C è il *passaggio per valore*.
Discuteremo diffusamente di ciò più avanti.

Come si compila `prog3` ?

Per compilare un programma su più file come `prog3`:

```
gcc -o prog3 prog3.c aux3.c
```

Si noti che il file di intestazione `prog3.h` non figura nel comando. Infatti si riportano solo i file di cui bisogna produrre una versione in formato oggetto: `prog3.o`, `aux3.o`.
Nel caso in cui, per esempio, `aux3.o` fosse già disponibile nella versione aggiornata, posso creare il file eseguibile con:

```
gcc -o prog3 prog3.c aux3.o
```

In questo caso `aux3.o` sfugge alle fasi di preprocessamento e compilazione, e viene solo usato dal linker per creare il file eseguibile.

Per creare uno (o più) file oggetto senza richiamare il linker:

```
gcc -c aux3.c
```

Questo modo di operare è spesso vantaggioso, soprattutto per progetti di molte migliaia di righe di codice: in questi casi, ripartire il codice su più file (in cui ogni file deve essere visto come un modulo contenente il codice per realizzare un determinato tipo di funzionalità) risulta conveniente per diverse ragioni.

Innanzitutto il codice così ripartito è più comprensibile e facilmente consultabile.

Inoltre si ha un effettivo vantaggio in termini di tempi di compilazione in quanto si devono ricompilare solo i file modificati dall'ultima compilazione, che, presumibilmente, in ogni momento dello sviluppo, non saranno molti.

L'utility `make` fornisce un valido aiuto nella gestione di progetti distribuiti su numerosi file. `make` è uno strumento generico e non si applica solo alla programmazione in C.

Nel file `makefile` si esplicitano le dipendenze fra i file che costituiscono il progetto.
Nel nostro caso il `makefile` apparirà così:

```
prog3: prog3.o aux3.o
    gcc -o prog3 prog3.o aux3.o

prog3.o: prog3.c prog3.h
    gcc -c prog3.c

aux3.o: aux3.c prog3.h
    gcc -c aux3.c
```

lanciando il comando `make` verranno effettuati solo i passi di compilazione strettamente necessari: i file aggiornati non verranno toccati.

Consideriamo l'esempio:

```
/* prog4.c: scrive la prima parola di input.txt su output.txt */
#include <stdio.h>

#define MAXWORD 80

int main(void) {
    FILE *fr = fopen("input.txt","r");
    FILE *fw = fopen("output.txt","w");
    char word[MAXWORD];

    fscanf(fr,"%s",word);
    fprintf(fw,"Parola letta: %s\n",word);

    /* Le ultime tre linee sono superflue in questo caso,
       poiche' sono le ultime istruzioni ad essere eseguite
       e i file verrebbero comunque chiusi automaticamente al
       termine dell'esecuzione del programma */
    fclose(fr);
    fclose(fw);
    return 0;
}
```

```
FILE *fr = fopen("input.txt","r");
FILE *fw = fopen("output.txt","w");
```

La struttura `FILE` è definita in `stdio.h` e permette l'accesso bufferizzato ai files.

La struttura `FILE` si utilizza attraverso puntatori: `fr` e `fw` sono di tipo: *puntatore a* `FILE`.

`fopen`, definita in `stdio.h`, restituisce un puntatore a `FILE`.
`fopen` apre il file il cui nome è il suo primo argomento, nella modalità specificata dal secondo argomento: "`r`": lettura, "`w`": scrittura, "`a`": accodamento.

Esistono anche le modalità “`r+`”, “`w+`”, “`a+`”, che aprono il file in aggiornamento (lettura *e* scrittura).

e anche “`rb`”, “`wb`”, “`ab`”, (e “`rb+`”, “`wb+`”, “`ab+`”), che specificano che il file sarà considerato un file binario: su UNIX/Linux non c'è differenza, nei sistemi Microsoft i file di testo sono rappresentati diversamente dai file binari.

`fscanf`, `fprintf`: analoghe a `scanf` e `printf`,
prototipo definito in `stdio.h`,
hanno un argomento aggiuntivo, il primo, che è un puntatore al file su cui operare: in lettura `fscanf` e in scrittura `fprintf`.

`fclose` chiude un file aperto.
Alla termine dell'esecuzione tutti i file aperti vengono chiusi automaticamente.

```
/* bubble.c */
#include <stdio.h>
#define DIM     10
void bubble(int [], int);  /* prototipo di bubble */

int main(void)
{
    int i, num[DIM];
    FILE *fr = fopen("numeri.txt","r");
    FILE *fw = fopen("ordinati.txt","w");
    /* leggo DIM numeri dal file */
    for(i = 0; i < DIM; i++)
        fscanf(fr,"%d",&num[i]);
    /* li ordino con bubblesort */
    bubble(num,DIM);
    /* li stampo sul file */
    for(i = 0; i < DIM; i++)
        fprintf(fw,"%d ",num[i]);
    fprintf(fw,"\n");
}
```

```
/* bubble ordina i primi n elementi
   dell'array di interi a[],
   usando bubblesort.
*/
void bubble(int a[], int n)
{
    int i,j,tmp;

    for(i = 0; i < n - 1; i++)
        /* sistemo l'i-esimo elemento piu' piccolo */
        for(j = n - 1; j > i; j--)
            if(a[j-1] > a[j]) {
                /* scambio */
                tmp = a[j-1];
                a[j-1] = a[j];
                a[j] = tmp;
            }
}
```

Analisi di `bubble.c`

```
void bubble(int [], int);
```

Prototipo della funzione `bubble`, definita dopo. Poiché `main` la usa, il compilatore deve già conoscerne il prototipo al momento di definire `main`.
Dichiarazione di tipo array: `int []` specifica che il primo argomento di `bubble` deve essere un *array* di `int`. Equivalentemente, avrei potuto usare `int *`, vale a dire *puntatore* a `int`. Vedremo come e perché gli *argomenti di funzione* di tipo array sono in realtà di tipo puntatore.

```
    int i, num[DIM];
```

Dichiarazione di variabili: locali a `main`. `i` è una variabile intera, `num` è un *array* di `DIM` interi. La dimensione di un array deve essere un'espressione costante intera.

```
for(i = 0; i < DIM; i++)
```

Istruzione di controllo del flusso: Il C, ovviamente, possiede istruzioni per controllare il flusso, vale a dire selezione ed iterazione. Il ciclo `for` in C è un costrutto molto flessibile e potente.

```
fscanf(fr,"%d",&num[i]);
```

Input formattato: legge un intero scritto in notazione decimale (`%d`) nella $i$esima componente di `num`. Notare che si passa a `scanf` l'indirizzo di `num[i]` (operatore `&`), per realizzare il passaggio per referenza.

Per accedere all'$i$esimo elemento di `num` si usa la notazione `num[i]`. Gli indici di un array C partono da 0.

```
    bubble(num,DIM);
```

Chiamata di funzione: viene invocata la funzione `bubble` passando l'array `num` (ne basta il nome) come primo argomento, e la sua dimensione come secondo.

```
void bubble(int a[], int n)
{ ... }
```

Definizione di funzione: viene definita `bubble`, si noti che qui i nomi dei parametri formali `a` e `n`, non sono tralasciati (cfr. con prototipo).

`bubble` è un'implementazione su `int` dell'algoritmo di ordinamento *bubblesort*, uno dei più ingenui e inefficaci algoritmi pensati per questo scopo. Il suo tempo di esecuzione è proporzionale al quadrato del numero di elementi da ordinare.

```
for(i = 0; i < n - 1; i++)
    for(j = n - 1; j > i; j--)
        ...
```

Corpo della definizione di `bubble`: Due cicli nidificati: il ciclo più interno, muove verso sinistra l'$i$esimo elemento più piccolo, attraverso confronti e, nel caso, *scambi*, fra il candidato attuale ad essere l'$i$esimo, e l'elemento immediatamente precedente.

```
        if(a[j-1] > a[j]) {
            /* scambio */
            tmp = a[j-1];
            a[j-1] = a[j];
            a[j] = tmp;
        }
```

Effettua lo scambio se la condizione `a[j-1] > a[j]` risulta vera.

NB. In C ogni espressione che restituisca un valore aritmetico può essere considerata una condizione: se il risultato è uguale a `0` allora la condizione risulta falsa, altrimenti risulta vera.

Viceversa, gli operatori logici, relazionali e d'uguaglianza restituiscono valori aritmetici (`1` per vero, `0` per falso).

Le prestazioni non sono buone:

Dopo $n - 1$ iterazioni del ciclo più esterno tutti gli elementi sono al loro posto.

Numero quadratico di confronti: $ncfr = \frac{n(n-1)}{2}$.

Ovviamente il numero di scambi è $\leq ncfr$.

Vediamo ora di modificare leggermente il codice di `bubble` per visualizzare meglio come funziona l'algoritmo e per valutare sugli esempi il numero di scambi effettuati.

```
int bubble(int a[], int n, FILE *fw)
{
    int i,j,k,tmp,nswap = 0;

    for(i = 0; i < n - 1; i++) {
        for(j = n - 1; j > i; j--) {
            for(k = 0; k < j-1; k++)
                fprintf(fw,"%d%c",a[k],k == i-1 ? '|':' ');
            if(a[j-1] > a[j]) {
                tmp = a[j-1]; a[j-1] = a[j]; a[j] = tmp;
                fprintf(fw,"%d*%d ",a[j-1],a[j]);
                nswap++;
            } else
                fprintf(fw,"%d %d ",a[j-1],a[j]);
            for(k = j+1; k < DIM; k++)
                fprintf(fw,"%d ",a[k]);
            fprintf(fw,"\n");
        }
        fprintf(fw,"\n");
    }
    return nswap;
}
```

Commenti sulla versione modificata di `bubble`:

Il prototipo è stato modificato: ora `bubble` ritornerà un intero (il numero di scambi effettuato) e riceverà come terzo argomento un puntatore a `FILE` su cui stampare.

Sono state inserite istruzioni per meglio evidenziare la dinamica dell'algoritmo.

```
k == i-1 ? '|':' '
```

Operatore ternario: Funziona come una versione contratta di *if-then-else*. L'espressione: *expr* `?` *expr1* `:` *expr2* ha il valore di *expr1* quando *expr* $\neq 0$, di *expr2* altrimenti.

```
nswap++
```

Operatore di incremento: equivale a `nswap = nswap + 1`.

Parametri passati per indirizzo. Puntatori

Vogliamo modificare la parte del codice di `bubble` che realizza lo scambio dei valori:

```
tmp = a[j-1]; a[j-1] = a[j]; a[j] = tmp;
```

sostituiamo questa riga con un'invocazione a una funzione di prototipo `void swap(int a, int b)` che scambi i suoi due valori:

Proviamo la seguente versione:

```
void swap(int a, int b)
{
    int tmp = a;

    a = b;
    b = tmp;
}
```

Non funziona, perché ?

Il C supporta unicamente il passaggio di parametri per valore.

Dunque, nella chiamata `swap(x,y)` vengono passati i valori di `x` e `y`. Questi valori vengono localmente manipolati (scambiati) senza nessun effetto sull'ambiente chiamante.

La soluzione in C: Passaggio per referenza realizzato passando il *valore* degli *indirizzi*:

```
void swap(int *a, int *b)
{
    int tmp = *a;

    *a = *b;
    *b = tmp;
}
```

Il tipo *indirizzo* di una variabile di tipo intero è il tipo **puntatore** a intero.

Per dichiarare un puntatore a `int`:

`int *a;`

La dichiarazione è contorta: la si legga come: *l'oggetto puntato da* `a` *è un intero*.

Infatti, nelle espressioni, come `*b = tmp;` , `*` è l'operatore di *dereferenziazione* che restituisce l'oggetto puntato.

Si legga `*b = tmp;` come: *assegna all'oggetto puntato da* `b` *il valore contenuto in* `tmp`.

Per invocare questa versione di `swap` bisogna passare due indirizzi validi di oggetti di tipo `int`. Eventualmente usando l'operatore `&` (operatore *indirizzo di ...*, duale di `*`).

Quindi per scambiare due valori interi:

```
...
int x = 5;
int y = 7;

swap(&x,&y); /* ora x contiene 7 e y contiene 5 */
```

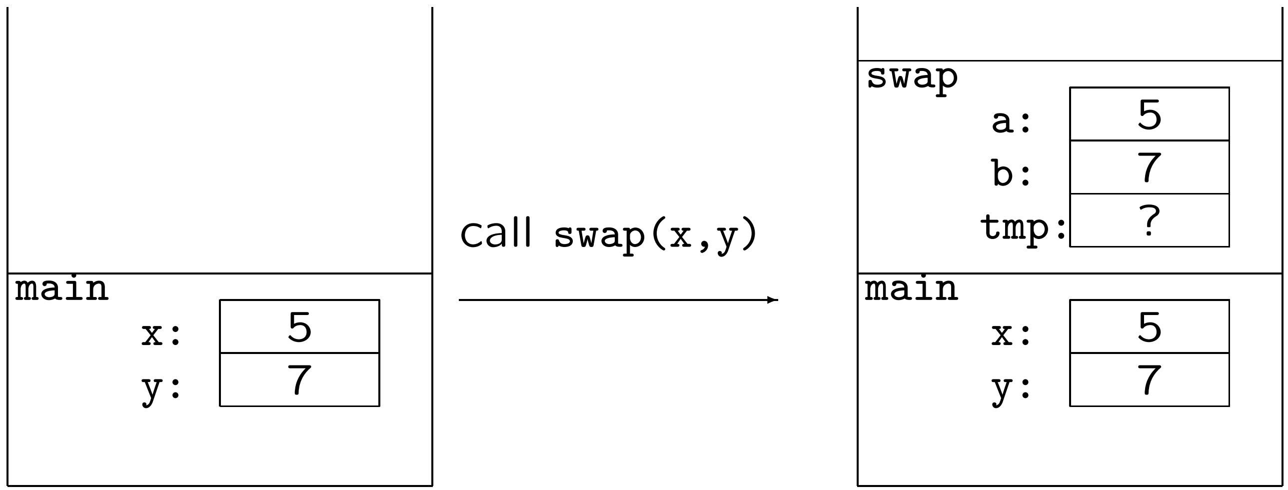
La funzione `bubble` diventa:

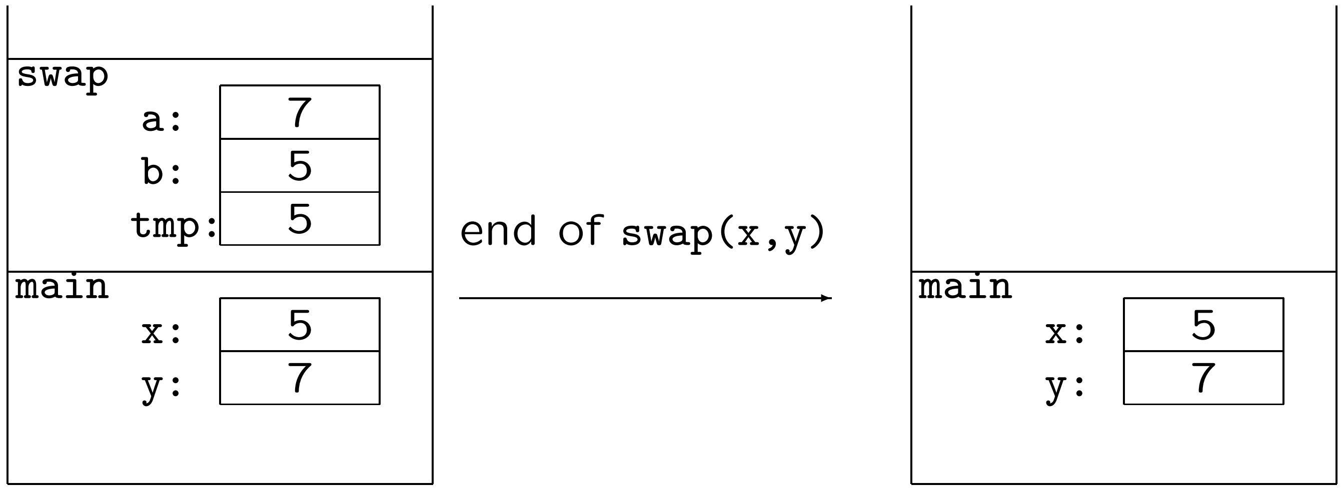
```
void bubble(int a[], int n)
{
    int i,j;

    for(i = 0; i < n - 1; i++)
        /* sistemo l'i-esimo elemento piu' piccolo */
        for(j = n - 1; j > i; j--)
            if(a[j-1] > a[j]) {
                /* scambio */
                swap(&a[j-1],&a[j]);
            }
}
```

Il comportamento di swap (nelle due diverse implementazioni)
Si consideri il frammento di codice seguente dove la versione di `swap` usata è quella con il prototipo `void swap(int a, int b)`:

```
int main(void) {
    int x = 5;
    int y = 7;
    swap(x,y);
}
```

Schematizziamo il contenuto dello stack dei record di attivazione:

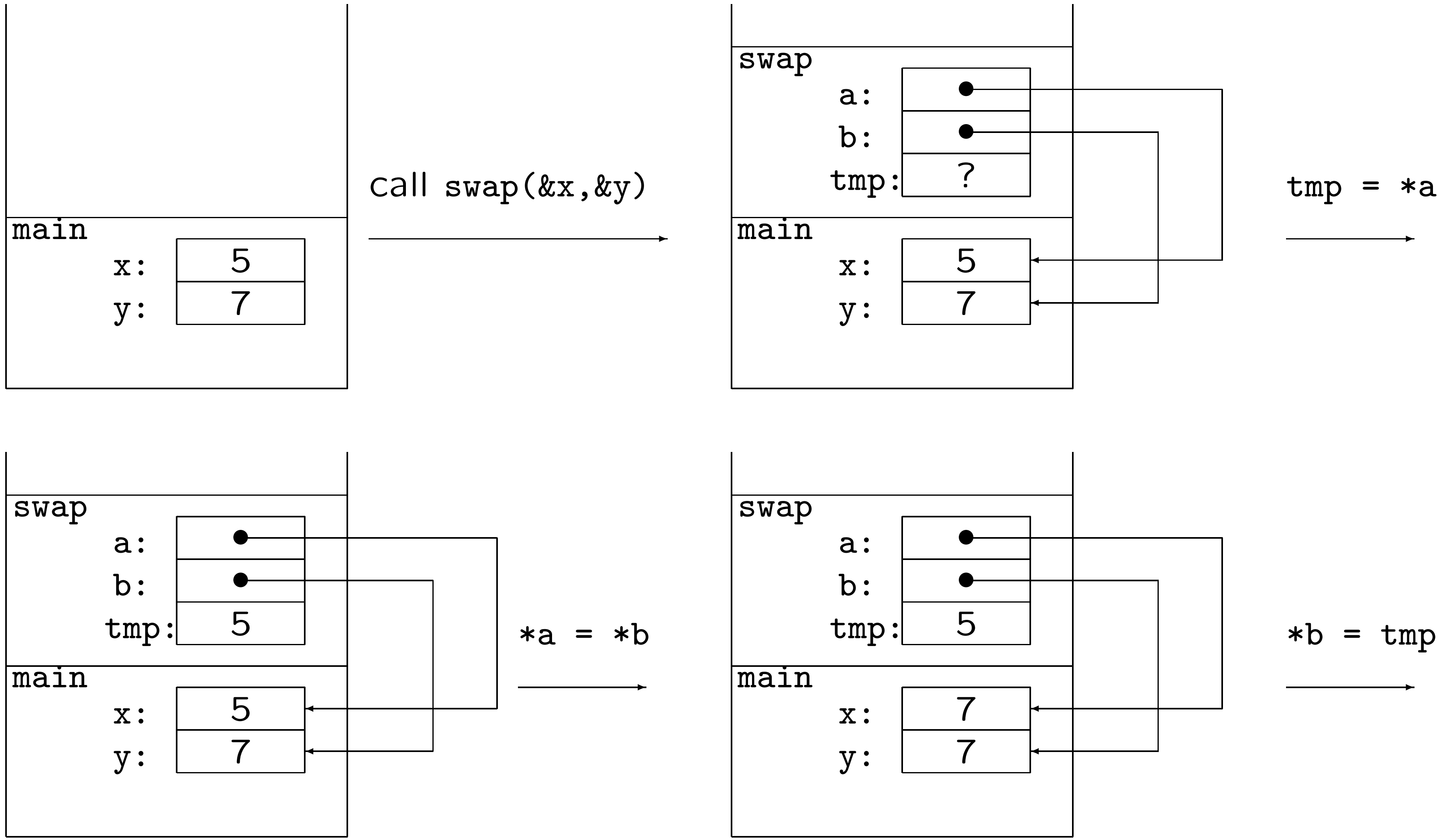


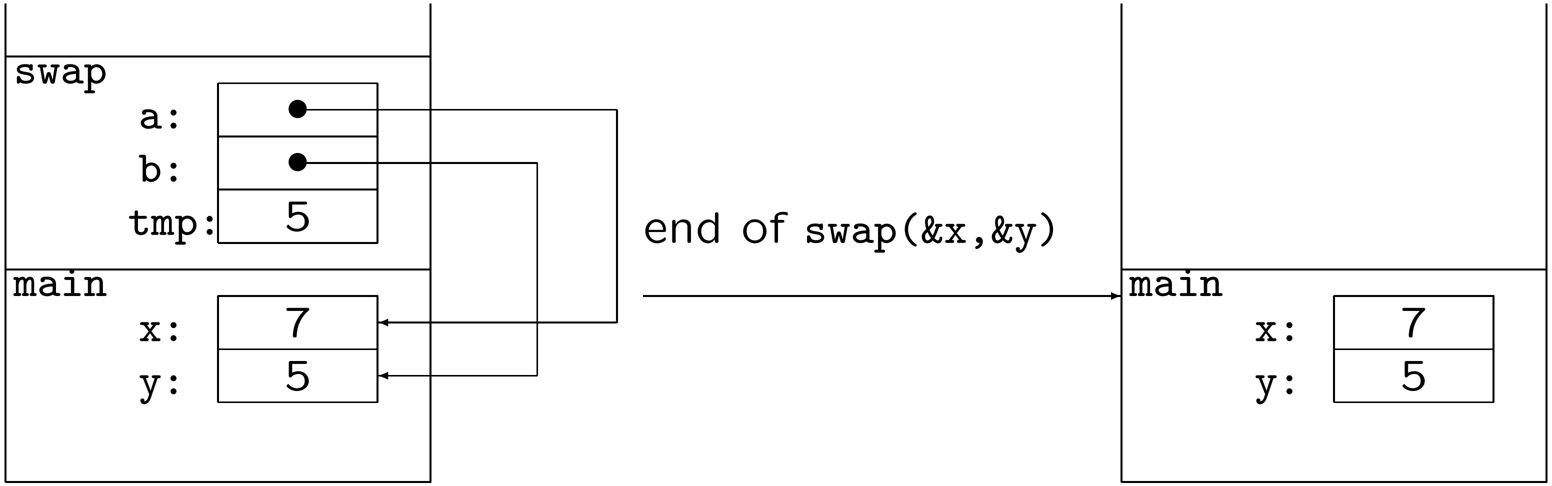


Non abbiamo ottenuto nulla da `swap`.

Si consideri ora la seconda definizione, quella con prototipo:
`void swap(int *a, int *b)`

```
int main(void) {
    int x = 5;
    int y = 7;
    swap(&x,&y);
}
```


main
x: 5
y: 7
call swap(&x,&y)
swap
a:
b:
tmp: ?
main
x: 5
y: 7
tmp = *a
swap
a:
b:
tmp: 5
main
x: 5
y: 7
*a = *b
swap
a:
b:
tmp: 5
main
x: 7
y: 7
*b = tmp



Il passaggio (*sempre per valore*) degli indirizzi di `x` e `y` a `swap` ha permesso a questa funzione di accedere alla regione di memoria associata a `x` e `y`, attraverso l'uso dell'operatore di dereferenziazione `*`.

Questo è il modo standard per simulare un passaggio di parametri per riferimento in C.

Un accenno alla relazione tra array e puntatori

Il nome di un array è un puntatore al primo elemento dell'array stesso.

Il prototipo di `bubble` (1^a vers.) può infatti essere specificato come:

```
void bubble(int *a, int n)
```

(o anche `void bubble(int *, int)` se stiamo dichiarando solo il prototipo e non siamo in sede di definizione della funzione).

Quando si richiama `bubble`, nulla cambia:

```
bubble(num, DIM);
```

`num`, il nome dell'array, è un'espressione che punta al primo elemento dell'array stesso:

```
num == &num[0];
```

Le **stringhe** sono array di elementi di tipo `char`, il cui ultimo elemento è `0`, o, detto in modo equivalente, il carattere `'\0'`.

```
char s[] = "pippo";
```

è equivalente a

```
char s[6] = "pippo";
```

è equivalente a

```
char s[] = {'p','i','p','p','o','\0'};
```

è equivalente a (NB: non del tutto equivalente, vedremo in che senso più avanti):

```
char *s = "pippo";
```

Strutture

Per aggregare variabili di tipo diverso sotto un unico nome il C fornisce le strutture (e le unioni).

```
struct punto {
    int x;
    int y;
};

struct rettangolocoloratoepesato{
    int colore;
    double peso;
    struct punto bottomleft, topright;
};

struct punto sommapunti(struct punto a, struct punto b)
{
    a.x += b.x;
    a.y += b.y;
    return a;
}
```

Elementi lessicali, operatori, espressioni

Elementi lessicali

Un programma C dal punto di vista sintattico è una sequenza di caratteri che il compilatore deve tradurre in codice oggetto.
Il programma deve essere sintatticamente corretto.

Concentriamoci sul processo di compilazione vero e proprio (trasformazione da `.c` a `.o`).
La prima fase di questo processo contempla il riconoscimento degli elementi lessicali: Raggruppamento dei caratteri in *token*.

```
int main(void)
{
    printf("Salve mondo\n");
    return 0;
}
```

Token: `int`,`main`,`()`,`void`, `{` ,`printf`, `"Salve mondo\n"`,`;`,`return`,`0`, `}` `int`,`void`,`return`: **parole chiave**

`main`,`printf`: **identificatori**

`"Salve mondo\n"`: **costante stringa**

`0`: **Costante intera**

`()`: **Operatore (chiamata di funzione)**

`;`, `}`, `{` : **Segni d'interpunzione**

Regole sintattiche

La sintassi del C può essere espressa formalmente mediante regole *Backus-Naur Form* (BNF).

Esempio: Definizione della categoria sintattica *digit* (*cifra*)

$$digit ::= 0|1|2|3|4|5|6|7|8|9$$

$\vert$	alternativa
$\{\}_1$	scegliere esattamente una delle alternative nelle graffe
$\{\}_{0+}$	ripetere la categoria tra graffe 0 o più volte
$\{\}_{1+}$	ripetere la categoria tra graffe 1 o più volte
$\{\}_{opt}$	opzionale

Esempio: $alphanumeric\_string ::= \{letter|digit\}_{0+}$

Identificatori:

$identifier ::= \{letter|\_\}_1\{letter|\_|digit\}_{0+}$

Costanti intere decimali:

$decimal\_integer ::= 0|positive\_decimal\_integer$
$positive\_decimal\_integer ::= positive\_digit\,\{digit\}_{0+}$
$positive\_digit ::= 1|2|3|4|5|6|7|8|9$

Costanti stringa:

$string\_literal ::= "\ \{character|escape\_sequence\}_{0+}\ "$
$escape\_sequence ::= \backslash'|\backslash"\,|\backslash ?|\backslash\backslash|\backslash \mathtt{a}|\backslash \mathtt{b}|\backslash \mathtt{f}|\backslash \mathtt{n}|\backslash \mathtt{r}|\backslash \mathtt{t}|\backslash \mathtt{v}|\backslash octal|\backslash \mathtt{x}\,hexadecimal.$

dove *character* non è " ne' \

Espressioni

Le espressioni sono combinazioni significative di **costanti**,**variabili**, **operatori** e **chiamate di funzioni**.

Costanti, variabili e chiamate di funzioni sono di per se' delle espressioni.

Un operatore combina diverse espressioni e produce un'espressione più complessa. il numero delle espressioni combinate è la *arità* dell'operatore.

Esempi:

```
i = 7
a + b
33 - bubble(num,DIM)
-4.207
"pippo"
```

Operatori

Ogni operatore possiede le seguenti proprietà:

arità: specifica il numero degli argomenti necessari per applicare l'operatore.

priorità: determina in che ordine vengono eseguite le operazioni in espressioni che coinvolgono altri operatori.

associatività: determina in che ordine vengono eseguite le operazioni in espressioni che coinvolgono altre occorrenze di operatori con la stessa priorità.

Per forzare un ordine diverso da quello determinato da priorità e associatività, si deve parentesizzare l'espressione con) e (.

Il C dispone degli ordinari operatori aritmetici, con le usuali arità, priorità e associatività.

```
(1 + 2 * 3) == (1 + (2 * 3))
```

```
(1 + 2 - 3 + 4 - 5) == ((((1 + 2) - 3) + 4) - 5)
```

Operatori aritmetici (in ordine di priorità):

+, -	(unari)	Associatività: da destra a sinistra
*, /, %		Associatività: da sinistra a destra
+, -	(binari)	Associatività: da sinistra a destra

Gli operatori aritmetici si applicano a espressioni intere o floating.
Vi è inoltre un'*aritmetica dei puntatori* (vedremo).

Operatori di incremento e decremento: ++ e --

++ e -- sono operatori unari con la stessa priorità del meno unario e associatività da destra a sinistra.

Si possono applicare solo a variabili (o meglio *lvalue*, come vedremo), di tipi interi, floating o puntatori, ma non a espressioni generiche (anche se di questi tipi).

```
int c = 5;

c++;  /* c == 6 */
c--;  /* c == 5 */
--c;  /* c == 4 */
++c;  /* c == 5 */
5++;  /* Errore! */
--(7 + bubble(num,DIM)); /* Errore! */
```

Semantica degli operatori di incremento/decremento

Postfisso: Il valore dell' espressione `c++` è il valore di `c`. Mentre `c` stesso è incrementato di 1.

Prefisso: Il valore dell' espressione `++c` è il valore di `c` incrementato di 1. Inoltre `c` stesso è incrementato di 1.

Analogo discorso vale per gli operatori di decremento.

```
int c = 5;

int b = 30 / c++; /* b == 6, c == 6 */
int d = 6 + --c;  /* d = 11, c == 5 */
```

Gli operatori prefissi modificano il valore della variabile cui sono applicati prima che se ne utilizzi il valore. Gli operatori postfissi modificano il valore della variabile dopo l'utilizzo del valore (vecchio) nell'espressione.

Operatori di assegnamento

In C, l'assegnamento è un'espressione:

```
int a,b,c;

a = b = c = 0;
printf("%d\n",b = 4); /* stampa: 4 */
```

Il valore di un'espressione *variabile* = *expr* è il valore assunto da *variabile* dopo l'assegnamento, vale a dire, il valore di *expr*.

Non ci si confonda con l'operatore di uguaglianza ==

```
int b = 0;

printf("%d,", b == 4); printf("%d,", b = 4); printf("%d\n", b == 4);
/* Che cosa stampa? */
```

Operatori di assegnamento

In C, l'assegnamento è un'espressione:

```
int a,b,c;

a = b = c = 0;
printf("%d\n",b = 4); /* stampa: 4 */
```

Il valore di un'espressione *variabile* = *expr* è il valore assunto da *variabile* dopo l'assegnamento, vale a dire, il valore di *expr*.

Non ci si confonda con l'operatore di uguaglianza ==

```
int b = 0;

printf("%d,", b == 4); printf("%d,", b = 4); printf("%d\n", b == 4);
/* stampa: 0,4,1 */
```

L'operatore di assegnamento ha priorità più bassa rispetto agli operatori aritmetici, e associa da destra a sinistra.

Operatori di assegnamento combinati:

`+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`

La loro semantica è:
variabile op= expr equivale a *variabile = variabile op expr*
(ma *variabile* è valutata una volta sola, questo aspetto è da ricordare quando *variabile* è una componente di un array, di una struttura, etc., esempio: `a[i++] += 2;`)

variabile può essere qualsiasi *lvalue* (vedi lucido seguente)

Esempio:

```
int a = 1;

a += 5; /* a == 6 */
```

Variabili e *lvalue*

È il momento di essere più precisi nel parlare di variabili e oggetti.

Per *oggetto* intendiamo un'area di memoria alla quale è stato associato un nome.

Un *lvalue* è un'espressione che si riferisce a un oggetto.

Solo gli *lvalue possono comparire a sinistra* di un operatore di assegnamento.

Le variabili semplici sono *lvalue*, ma sono *lvalue* anche (i "nomi") di elementi di array (es . `a[i] = ...`), le espressioni consistenti nella dereferenziazione di un puntatore (es. `*ptr = ...`), i campi delle variabili `struct` (es. `p.x = ...`), e altre costruzioni ancora (le vedremo).

Esempio di utilizzo di operatori di assegnamento combinati

```
/* power2.c */
#include <stdio.h>

#define MAXEXP  10

int main(void)
{
    int i = 0, power = 1;

    while(++i <= MAXEXP)
        printf("%6d", power *= 2);
    printf("\n");
    return 0;
}
```

`while(++i <= MAXEXP)` : il ciclo viene ripetuto fino a quando `i` non sarà maggiore di `MAXEXP`.
`power *= 2` : ad ogni iterazione, il valore di `power` viene raddoppiato, e poi stampato.

Un esempio di utilizzo di operatori di incremento e di assegnamento

```
#include <stdio.h>

void copia(char *t, char *s)
{
    int i = 0;

    while((t[i] = s[i]) != 0)
        i++;
}

int main(void)
{
    char s[] = "pippo";
    char t[10];

    copia(t,s);
    printf(t);
    return 0;
}
```

Commenti:

```
while((t[i] = s[i]) != 0)
```

Si esce dal ciclo quando la *condizione* argomento di `while` diventa falsa, più precisamente, quando l'*espressione* argomento di `while` viene valutata `0`.

```
(t[i] = s[i]) != 0
```

Poiché l'assegnamento è un'espressione, qui se ne usa il valore: quando essa varrà 0 il ciclo terminerà. Essa varrà 0 quando varrà 0 la variabile `t[i]`, vale a dire quando varrà 0 la variabile `s[i]`.

```
i++
```

a ogni iterazione, il valore di `i` viene incrementato.

s:	'p'	'i'	'p'	'p'	'o'	0				
	i									
t:	?	?	?	?	?	?	?	?	?	?

`t[i] = s[i]` vale `'p'`

---

s:	'p'	'i'	'p'	'p'	'o'	0				
		i								
t:	'p'	?	?	?	?	?	?	?	?	?

`t[i] = s[i]` vale `'i'`

. . .

`t[i] = s[i]` vale `'o'`

---

s:	'p'	'i'	'p'	'p'	'o'	0				
						i				
t:	'p'	'i'	'p'	'p'	'o'	0	?	?	?	?

`t[i] = s[i]` vale 0,

cioè la condizione del ciclo è *falsa*

Esercizio:

Cosa succede se rimpiazzo `copia` con:

```
void copia(char *t, char *s)
{
    int i = 0;

    while((t[i++] = s[i++]) != 0);
}
```

e se la rimpiazzo con:

```
void copia(char *t, char *s)
{
    int i = 0;

    while(t[i] = s[i])
        ++i;
}
```

Cosa succede se invece rimpiazzo `copia` con:

```
void copia(char *t, char *s)
{
    int i;

    for(i = 0;t[i] = s[i];i++);
}
```

Si noti che in questo caso l'istruzione `for` è immediatamente seguita da `;` : il ciclo è vuoto, nessuna istruzione viene ripetuta sotto il controllo del ciclo `for`. Solo l'espressione di controllo viene valutata ad ogni iterazione.

```
t[i] = s[i]
```

L'espressione di controllo è costituita semplicemente dall'assegnamento. Poiché l'assegnamento è un'espressione, il ciclo terminerà quando questa espressione varrà `0`. Cioè quando `s[i]`, che è il valore assegnato a `t[i]` ed è anche il valore di tutta l'espressione, sarà `0`.

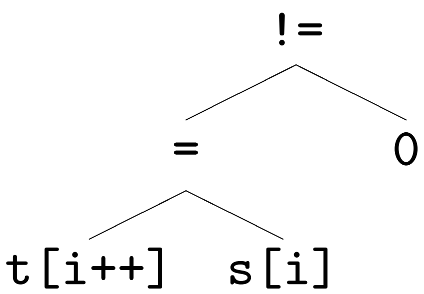
N.B. Abbiamo implementato `copia` per esercizio. La libreria standard contiene la funzione `strcpy`, il cui prototipo è in `string.h`.

Consideriamo questa versione di `copia`:

```
void copia(char *t, char *s)
{
    int i = 0;

    while((t[i++] = s[i]) != 0);
}
```

Funziona ?

E se rimpiazziamo la condizione del ciclo `while` con `(t[i] = s[i++]) != 0` ?



Associatività e Priorità determinano
la forma dell'albero di parsing,
ma non specificano del tutto l'ordine di valutazione.
L'espressione può essere valutata in due modi diversi
a seconda di quale sottoalbero di `=` viene valutato per primo.
Lo standard non risolve questo tipo di ambiguità.