Argomenti di `main`:

La funzione `main` può ricevere due argomenti:

– Il primo, di tipo `int` contiene il numero dei parametri con cui l'eseguibile è stato richiamato dalla riga di comando del sistema operativo.

– Il secondo, di tipo `char *[]` è un array di stringhe ognuna delle quali è una parola della riga di comando.
La prima di queste stringhe contiene il nome stesso dell'eseguibile.
(in alcuni sistemi DOS e Windows, il percorso intero dell'eseguibile).

Convenzionalmente il primo parametro è chiamato `int argc`,
il secondo `char *argv[]`.

Supponiamo di aver compilato `myprog.c`,
e supponiamo che il `main` sia:

```
int main(int argc, char *argv[])
```

Si esegua dal prompt del sistema operativo:

```
myprog par1 par2 par3
```

	argc	contiene	4
	argv[0]	contiene	"myprog"
Allora:	argv[1]	contiene	"par1"
	argv[2]	contiene	"par2"
	argv[3]	contiene	"par3"

Esempio: modifichiamo `sw.c`:

```
int main(int ac, char *av[])
{
    int i,n;
    char **w, **b;
    FILE *fr = stdin, *fw = stdout;

    switch(ac) {
    case 1:
        break;
    case 3:
        if(!(fw = fopen(av[2],"w")))
            errore("Impossibile scrivere sul file \"%s\"",av[2]);
    case 2:
        if(!(fr = fopen(av[1],"r")))
            errore("File \"%s\", di input non trovato",av[1]);
        break;
    default:
        errore("Numero di argomenti non permesso","");
    }
    w = leggitesto(fr,&n);
    b = calloc(n, sizeof(char *));
    mergesort(w,b,0,n-1);
    for(i = 0;i < n;i++)
        fprintf(fw,"%s\n\r",w[i]);
}
```

Nota:

L'intestazione di `main` si sarebbe potuta scrivere anche come:

```
int main(int ac, char **av)
```

infatti `char *av[]` dichiara un'array di puntatori a `char`, e nel contesto di una dichiarazione di parametri di funzione, ciò equivale a dichiarare `char **av`: cioè, puntatore a puntatore a `char`.

Esercizio:

Nella funzione `main` dell'esempio precedente, si è fatto un uso poco ortodosso dell'istruzione `switch`.

Motivate l'affermazione precedente.
Spiegate perchè il codice funziona.
Riscrivete il codice senza usare `switch`.

Puntatori a funzioni

In C, così come il nome di un array è un puntatore al suo primo elemento, il nome di una funzione è un puntatore alla funzione stessa.

Il C permette la manipolazione esplicita dei puntatori a funzione. La sintassi è intricata.

Sia `int f(short, double)`.
Allora il nome `f` è un puntatore di tipo:
`int (*)(short, double)`.

Dichiarando un puntatore dello stesso tipo, si può effettuare, ad esempio, un assegnamento:

```
int (*ptrtof)(short, double);  /* ptrtof puntatore a funzione */

ptrtof = f;                    /* assegna l'indirizzo di f a ptrtof */
(*ptrtof)(2,3.14);             /* invoca f */
```

Un uso tipico dei puntatori a funzione consiste nel passarli come argomenti di funzioni:
Modifichiamo `mergesort`:

```
void mergesort(int *a, int *b, int l, int r, char (*comp)(int, int))
{
    int i,j,k,m;

    if(r > l) {
        m = (r+l)/2;
        mergesort(a,b,l,m,comp);
        mergesort(a,b,m+1,r,comp);
        for(i = m+1; i > l; i--)
            b[i-1] = a[i-1];
        for(j = m; j < r; j++)
            b[r+m-j] = a[j+1];
        for(k = l; k <= r; k++)
            a[k] = (*comp)(b[i],b[j]) ? b[i++] : b[j--];
    }
}
```

Ora `mergesort` accetta un puntatore a una funzione per la comparazione di due elementi come ulteriore argomento.

Usiamo questa versione di `mergesort` con le seguenti funzioni di comparazione nel file `comp.c`:

```
char smaller(int a, int b)
{
    return a < b;
}

char greater(int a, int b)
{
    return a > b;
}

#include <stdlib.h>
#include <time.h>
char coin(int a, int b)
{
    static char flag = 0;

    if(!flag) { srand(time(NULL)); flag = 1; }
    return rand() % 2;
}
```

Ad esempio, per ordinare in ordine decrescente, invocheremo:

```
mergesort(a,b,l,r,greater);
```

Avvertenze:
Funzioni "corte" come `smaller` avremmo potuto scriverle come macro, ma in questo caso non avremmo avuto alcun puntatore a funzione, e non avremmo potuto invocare `mergesort` con `smaller` come parametro.

Poiché il nome di una funzione è già un puntatore, il C accetta che il prototipo di funzione sia specificato anche come:

```
void mergesort(int *a, int *b, int l, int r, char comp(int, int))
```

Il tipo di un puntatore a funzione è determinato anche dal tipo di ritorno e dalla lista di parametri: `int (*)(void)` è un tipo diverso da `int (*)(int)`.

Si presti attenzione alla sintassi: `int *f(int);` dichiara il prototipo di una funzione che restituisce un puntatore a intero.
`int (*f)(int);` dichiara un puntatore a funzione.

A dispetto della sintassi "poco intutitiva", gli array di puntatori a funzioni spesso sono di grande utilità.

```
int main(void)
{
    int i,e,a[] = { 14, 16, 12, 11, 5, 21, 17, 14, 12 };
    static char (*choice[])(int, int) = { smaller, greater, coin };

    printf("Array da ordinare:");
    for(i = 0; i < 9; i++) printf("%d ",a[i]);
    putchar('\n');

    do {
        printf("Scegli:\n");
        printf("0: ordina in modo crescente\n");
        printf("1: ordina in modo decrescente\n");
        printf("2: rimescola casualmente\n");
        if(!(e =(scanf("%d",&i) && i >= 0 && i <= 2))) {
            printf("\nImmetti o 0 o 1 o 2.\n");
            while(getchar() != '\n');/* per svuotare l'input */
        }
    } while(!e);

    ordina(a,0,8,choice[i]);
    for(i = 0; i < 9; i++) printf("%d ",a[i]);
    putchar('\n');
}
```

`static char (*choice[])(int, int) = { smaller, greater, coin };`
definisce un array di puntatori a funzioni che ritornano `char` e che richiedono due parametri `int`.
Inizializza l'array con i tre puntatori `smaller, greater, coin`.

`char smaller(int, int);`	dichiara il prototipo di funzione
`char (*f)(int, int);`	dichiara un puntatore a funzione
`char (*af[])(int, int);`	dichiara un array di puntatori a funzione

Leggi:

`af[i]`	è l'$i$esimo puntatore a funzione
`(*af[i])`	è una funzione che ritorna `char` e riceve due `int`.

Si può usare `typedef` per semplificare dichiarazioni/definizioni:

```
typedef char (*PTRF)(int, int); /*PTRF e' sinonimo del tipo puntatore a funzione*/
                                /*che ritorna char e prende due int             */
PTRF choice[] = { ... };   /* choice e' il nostro array di puntatori a funzione */
```

Strutture e Unioni

Strutture

Gli array sono tipi costruiti a partire dai tipi fondamentali che raggruppano dati omogenei: ogni elemento dell'array ha lo stesso tipo.
Gli elementi sono acceduti tramite un indice numerico: (`a[i]`).

Con le *strutture* si possono definire tipi che aggregano variabili di tipi differenti.
Inoltre, ogni membro della struttura è acceduto per nome: (`a.nome`).

Per dichiarare strutture si usa la parola chiave `struct`:

```
struct impiegato {
    int id;
    char *nome;
    char *indirizzo;
    short ufficio;
};
```

Il nome di un tipo struttura è costituito dalla parola `struct` seguita dall'etichetta.
Per dichiarare variabili della struttura appena introdotta:

```
struct impiegato e1, e2;
```

La definizione del tipo può essere contestuale alla dichiarazione di variabili del tipo stesso:

```
struct animale {
    char *specie;
    char *famiglia;
    int numeroesemplari;
} panda, pinguino;
```

Una volta definito un tipo struttura lo si può usare per costruire variabili di tipi più complicati: ad esempio array di strutture:

```
struct complex {
    double re, im;
} a[120];              /* dichiara un array di 120 struct complex */
```

I nomi dei membri all'interno della stessa struttura devono essere distinti, ma possono coincidere senza conflitti con nomi di membri di altre strutture (o di normali variabili).

Strutture con etichette diverse sono considerate tipi diversi, anche quando le liste dei membri coincidono.

```
struct pianta {
    char *specie;
    char *famiglia;
    int numeroesemplari;
} begonia;

void f(struct animale a) {}

int main(void) { f(begonia); } /* errore: tipo incompatibile */
```

Accesso ai membri di `struct`

Vi sono due operatori per accedere ai membri di una struttura.

Il primo di questi è l'operatore `.`:

struct_variable . *member_name*

```
panda.numeroesemplari++;   /* e' nato un nuovo panda */
```

La priorità di `.` è massima e associa da sinistra a destra.

```
struct complextriple {
    struct complex x, y, z;
} c;

c.x.re = c.y.re = c.z.im = 3.14;
```

Spesso le variabili dei tipi `struct` vengono manipolate tramite puntatori.

Il secondo operatore per l'accesso ai membri di una struttura (->) si usa quando si ha a disposizione un puntatore a una variabile di tipo `struct`:

pointer_to_struct -> *member_name*

Questo costrutto è equivalente a:

(**pointer_to_struct*) . *member_name*

```
struct complex *add(struct complex *b, struct complex *c)
{
    struct complex *a = malloc(sizeof(struct complex));

    a->re = b->re + c->re;
    a->im = b->im + c->im;
    return a;
}
```

-> ha la stessa priorità e la stessa associatività dell'operatore . .

La definizione di un nuovo tipo `struct` introduce i nomi dei membri della `struct` stessa: poiché questi devono essere noti al compilatore quando trasforma un file sorgente in formato oggetto, è buona norma collocare la definizione dei tipi `struct` in appositi file d'intestazione, da includere dove necessario.

```
/* file complex.h */

struct complex {
    double re, im;
};

/* file complex .c */
#include "complex.h"

struct complex *add(struct complex *b, struct complex *c)
{
    ...
}
```

Digressione: Uso di `typedef`

L'istruzione `typedef` introduce sinonimi per tipi costruibili in C. Spesso viene usata per semplificare dichiarazioni complesse e/o per rendere più intuitivo l'uso di un tipo in una particolare accezione.

Poiché introducono nuovi nomi di tipi che il compilatore deve conoscere per poter tradurre i file sorgente, le definizioni `typedef` si pongono di norma nei file d'intestazione.

Lo spazio dei nomi usato da `typedef` è diverso dallo spazio delle etichette per i tipi `struct`, quindi si può riutilizzare un'etichetta di un tipo `struct` come nome sinonimo per un tipo:

```
typedef struct complex complex;
```

La sintassi di `typedef` è:

`typedef` *dichiarazione di variabile*

Premettendo `typedef` alla dichiarazione (priva di inizializzazione) di un identificatore si definisce l'identificatore come sinonimo del tipo della dichiarazione.

Esempi d'uso di `typedef`

```
int m[10][10];          /* definisce la variabile m come array di array */
typedef int matrix[10][10]; /* definisce matrix come sinonimo di array di 10 array di 10 int */

typedef char (*PTRF)(int,int); /* definisce un tipo di puntatore a funzione */
PTRF a[10];                    /* definisce un array di quel tipo di puntatori a funzione */

typedef unsigned int size_t;  /* definisce size_t come sinonimo di unsigned int */

typedef struct complex complex; /* definisce complex come sinonimo di struct complex */
complex a;                      /* dichiara una variabile a di tipo struct complex */
```

Digressione: File di intestazione

Quali informazioni sono usualmente contenute nei file d'intestazione `.h` da includere nei file sorgente `.c`:

- definizioni di macro, inclusioni di file d'intestazione.
- definizioni di tipi `struct`, `union`, `enum`.
- `typedef`.
- prototipi di funzione.
- dichiarazioni `extern` di variabili globali.

Digressione: Già che ci siamo:

- I file sorgente `.c` **non** si devono mai includere!

- Per condividere informazioni fra diversi file sorgente `.c`, creare un apposito file di intestazione `.h` e includerlo nei sorgenti. Al resto ci pensa `gcc`: es. `gcc -o nome s1.c s2.c s3.c s4.c` .

- Un file di intestazione `.h` non contiene mai *definizioni* di funzioni, ma solo prototipi.

- Una macro con parametri, per poter essere usata da diversi file sorgenti `.c`, deve essere definita in un file `.h` incluso da questi sorgenti.

- Riducete al minimo necessario le variabili globali nei vostri programmi.

- Riducete la visibilita' di variabili globali e funzioni usate solo in un particolare file `.c` dichiarandole `static`.

In ANSI C è lecito l'assegnamento tra strutture:

```
struct s {
    int dato;
    char a[2000];
};

struct s s1 = { 15, "stringa di prova" }; /* inizializzazione di struct */
struct s s2 = s1;                          /* inizializzazione per copia di s1 */
struct s s3;

s3 = s1;                                   /* assegnamento: copia membro a membro */
```

L'assegnamento di una variabile struttura a un'altra avviene attraverso la copia membro a membro.
Se la struttura contiene un array come membro, anche l'array viene duplicato.
L'assegnamento `s3 = s1;` equivale a:

```
s3.dato = s1.dato;
for(i = 0; i < 2000; i++) s3.a[i] = s1.a[i];
```

In ANSI C, l'assegnamento di strutture ha la semantica della copia membro a membro.
Una situazione analoga si presenta nel passaggio di parametri alle funzioni: variabili di tipo `struct` possono essere passate come parametri alle funzioni e restituite dalle funzioni stesse.
Il passaggio di una variabile `struct` avviene quindi per valore.

```
struct s cambiadato(struct s s1, int dato)
{
    s1.dato = dato;
    return s1;
}
```

Questo è in netto contrasto con la gestione degli array, dove si passano e si assegnano i puntatori ai primi elementi degli array, non gli array stessi.

Quando si passa una variabile struttura a una funzione viene creata una copia locale di ogni membro della variabile stessa, membri di tipo array compresi.

Questo può risultare molto oneroso nei termini dello spazio necessario per memorizzare la copia e nel tempo necessario a realizzare la copia stessa.

Spesso una soluzione migliore prevede il passaggio di parametri di tipo `struct` attraverso puntatori.

```
void cambiadatobis(struct s *s1, int dato) /* s1 e' un puntatore a struct s */
{
    s1->dato = dato;
}
```

Inizializzazione:
Le variabili `struct` di classe `extern` e `static` sono per default inizializzate ponendo tutti i membri a 0.
Le variabili `struct` possono essere esplicitamente inizializzate (anche quelle di classe `auto`), in maniera analoga al modo in cui si inizializzano gli array.

```
struct struttura {
    int id;
    char *s;
    int dato;
};

struct struttura v = { 15, "pippo" }; /* v.id = 15, v.s = "pippo", v.dato = 0 */
```

`struct` **anonime**:
E' possibile non specificare alcuna etichetta per una `struct`, in tal caso, le uniche variabili dichiarabili di quel tipo `struct` sono quelle dichiarate contemporaneamente alla `struct`:

```
struct {
    int id;
    char *s;
    int dato;
} v, w;
```

ESEMPIO: Rivisitiamo l'implementazione di `stack` e il suo uso nell'applicazione per il controllo della parentesizzazione:

```
/* stk.h */

struct stack {
    int pos;
    int dim;
    char *buf;
};

typedef struct stack stack;

void push(stack *s, char e);
char pop(stack *s);
char empty(stack *s);
stack *createstack(void);
void destroystack(stack *s);
```

Nel file d'intestazione riportiamo la definizione di `stack`, la `typedef` che introduce il nome di tipo `stack`, e i prototipi delle funzioni che manipolano lo `stack`.

Nella struttura `struct stack`:

`char *buf` è un array di caratteri allocato dinamicamente, che memorizzerà il contenuto dello `stack`.

`int dim` è la dimensione corrente dell'array `buf`, che viene allocato `BLOCKSIZE` byte per volta.

`int pos` è l'indice della posizione corrente d'inserimento/cancellazione (la "cima") dello `stack`.

Il file `stk.c` contiene l'implementazione delle funzioni.

```
/* stk.c */

#include <stdio.h>
#include <stdlib.h>
#include "stk.h"
```

```
void manageerror(char *s, short code)
{
    fprintf(stderr,"Errore: %s\n",s);
    exit(code);
}

#define BLOCKSIZE   10

void push(stack *s, char e)
{
    if(s->pos == s->dim) {
        s->dim += BLOCKSIZE;
        if(!(s->buf = realloc(s->buf,s->dim)))
            manageerror("Impossibile riallocare lo stack",1);
    }
    s->buf[s->pos++] = e;
}

char pop(stack *s)
{
    if(s->pos)
        return s->buf[--s->pos];
    else
        return EOF;
}
```

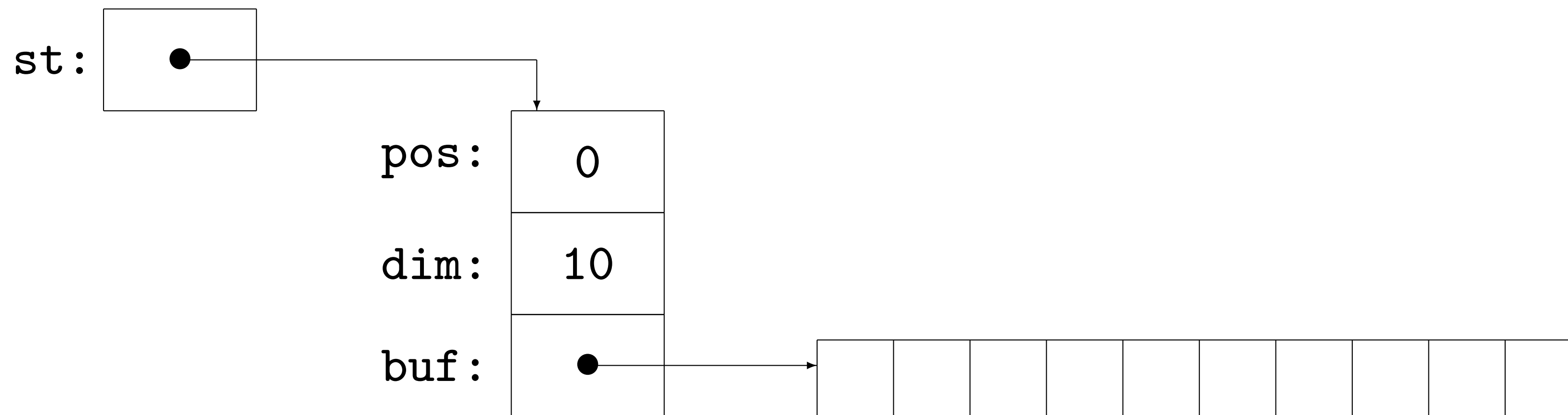
```
char empty(stack *s)
{
    return !s->pos;
}

stack *createstack(void)
{
    stack *s = malloc(sizeof(stack));

    if(!s)
        managerror("Impossibile creare stack",2);
    if(!(s->buf = malloc(s->dim = BLOCKSIZE)))
        manageerror("Impossibile creare stack",3);
    s->pos = 0;
    return s;
}

void destroystack(stack *s)
{
    if(s) {
        free(s->buf);
        free(s);
    }
}
```

```
st = createstack();
```



Abbiamo modificato leggermente anche `paren.c`: eccone il `main`.
(Per compilare `gcc -o paren paren.c stk.c`)

```
int main(void)
{
    int c;
    char d;
    stack *s = createstack();

    printf("Immetti stringa:\n");
    while((c = getchar()) != EOF && c != ';') {
        if(parentesi(c)) {
            if(!empty(s)) {
                if(!chiude(c,d = pop(s))) {
                    push(s,d);
                    push(s,c);
                }
            } else
                push(s,c);
        }
    }
    printf("%s\n", empty(s) ? "Corretto" : "Sbagliato");
    destroystack(s);
}
```

Unioni

Le *unioni* sono strutture "*salva-spazio*".

La sintassi della dichiarazione di una unione è del tutto analoga alla dichiarazione di una struttura, ma si usa la parola chiave `union` al posto di `struct`.

```
union unione {
    int id, dato;
    char array[20];
} u;
```

In una `union` la memoria viene condivisa da ogni membro: nell'esempio la variabile `u` può contenere:
o l'intero `id`, o l'intero `dato`, o l'array di `char` array.

La dimensione di `u` è quella del più grande dei suoi membri.

Il problema di interpretare correttamente il valore attuale di una variabile `union` (vale a dire, a quale membro della `union` riferire il valore), è a carico esclusivo del programmatore.

```
union aux {
    unsigned intero;
    unsigned char cifre[sizeof(unsigned)];
};

int main(void)
{
    short i;
    union aux n;

    printf("Immetti intero\n");
    scanf("%u",&(n.intero));
    for(i = sizeof(unsigned); i; i--)
        printf("%u,",n.cifre[i-1]);
    putchar('\n');
}
```

Cosa fa questo programma ?

Il programmatore deve gestire esplicitamente l'informazione riguardante quale sia il membro di una variabile `union` con cui interpretare il valore della variabile stessa.
Spesso questa informazione è memorizzata insieme alla `union`:

```
struct elemento {
    char tipo;  /* 0: i, 1: c, 2: d, 3: p */
    union {
        int i;
        char c;
        double d;
        int *p;
    } dato;
};

void f(struct elemento *e) {
    switch(e->tipo) {
    case 0: useint(e->dato.i); break;
    case 1: usechar(e->dato.c); break;
    case 2: usedouble(e->dato.d); break;
    case 3: useptr(e->dato.p); break;
    }
}
```