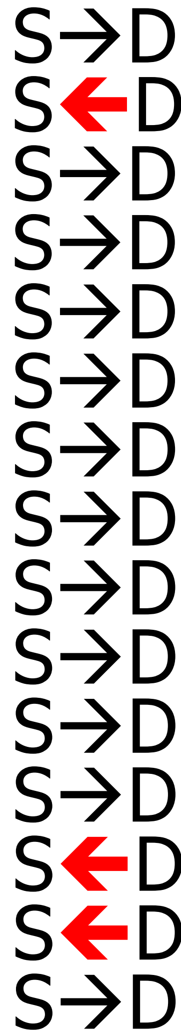
Complementi sul C - 2

Ver. 3

© 2021 - Claudio Fornaro - Corso di programmazione in C (C89)

Precedenza e associatività

```
() [] -> .
! ~ ++ -- + - * & (cast) sizeof
* / %
+ - (somma e sottrazione)
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= |= <<= >>=
,
```



- S→D: da Sinistra a Destra, S←D: da Dst. a Sin.

Esempi

- `https://cdecl.org/`
- `*p.x` equivale a `*(p.x)` in quanto il '`.`' ha priorità maggiore di `*`, identifica l'oggetto puntato dal membro `x` della `struct p`
- `*p->n++` equivale a `*((p->n)++)` in quanto `->` ha priorità maggiore degli altri operatori e questi hanno la stessa priorità ma associatività da destra a sinistra, `p` è un puntatore a una `struct` in cui il membro `n` è un puntatore che viene dereferenziato e poi incrementato
- `char **q` equivale a `char *(*q)`

  `q` è un puntatore a puntatore a `char`

Esempi

- `int mx[5][7]` equivale a `int (mx[5])[7]`
  `mx` è vettore di 5 vettori di 7 `int`
- `int *vett[5]` equivale a `int *(vett[5])`
  `vett` è un vettore di 5 puntatori a `int`
- `int (*vett)[5]`
  `vett` è un puntatore a un vettore di 5 `int`
- `int (*f)(long)`
  `f` è un puntatore a funzione con un argomento `long` e che restituisce un `int`
- `int *f(long)`
  `f` è una funzione con argomento un `long` e che restituisce un puntatore a `int`

Esempi

- `char (*(*f())[])()`
  Si parte dall'identificatore che è `f`, le parentesi hanno priorità su `*` quindi `f` è una funzione. L'asterisco `*` davanti a `f()` è dentro una coppia di parentesi, quindi `f` è una funzione che restituisce un puntatore. Tutto ciò ha a sinistra `*` e a destra `[]`, le parentesi hanno priorità su * quindi tipo `f` è una funzione che restituisce un puntatore ad un array di puntatori. Tutto questo è tra parentesi e ha a sinistra `()` che indica che sono puntatori a funzione e il `char` è il tipo restituito da queste funzioni:
- `f` è funzione che restituisce un puntatore ad un array di puntatori a funzione che restituiscono un `char`

Esempi

- `char (**f()[])()`
  `f` è una funzione che restituisce un array di puntatori a puntatori a funzioni che restituiscono un `char`
- `char **f()[]()`
  `f` è una funzione che restituisce un array di funzioni che restituiscono puntatori a puntatori a `char`
- `char (*(*v[3])(double,double))[5]`
  `v` è un array di 3 puntatori a funzione (con parametri due `double`) che restituiscono un puntatore a un array di 5 `char`

Esempi

- `char *(*(*v[3])())()`
  `v` è un array di 3 puntatori a funzione che restituiscono un puntatore a funzione che restituiscono un puntatore a `char`
- `void (*signal(int, void (*)(int)))(int)`
  `signal` è una funzione (con un primo parametro `int` e un secondo parametro puntatore a funzione che richiede un int e restituisce void) che restituisce un puntatore (*) a funzione che richiede un parametro int e restituisce void

Esempi

```
void (*signal(int sig, void (*handler)(int))
      )(int)
```

`signal` è una funzione che ha come parametri

- un `int` (`sig`)
- un puntatore a funzione (`handler`) che ha come parametro un `int` e come tipo restituito un `void`

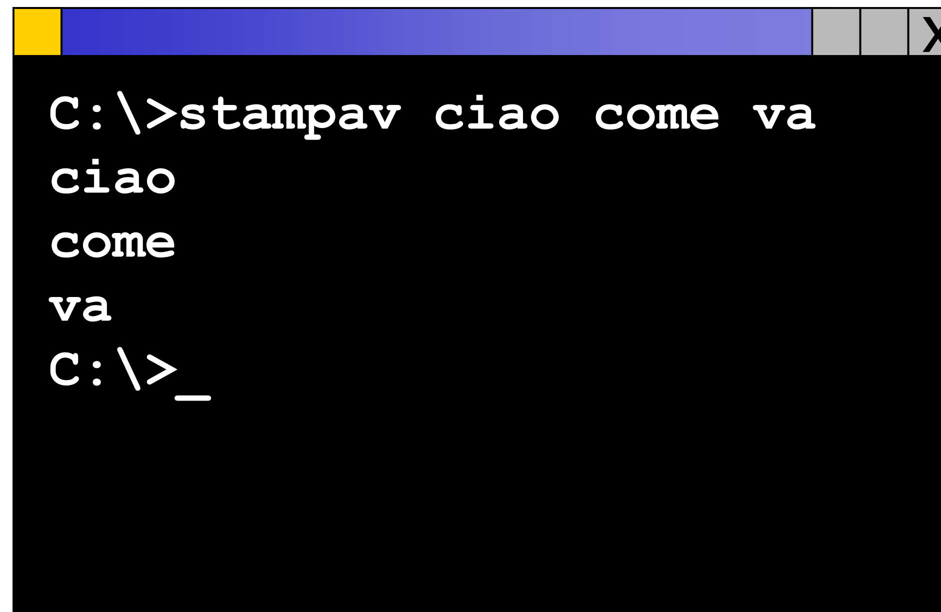
e restituisce un puntatore a funzione che ha come argomento un `int` e restituisce un `void`

Rispetto alla dichiarazione precedente qui sono specificati i nomi dei parametri `sig` e `handler` che verranno utilizzati nel corpo della funzione

Argomenti nella riga di comando

- Permettono di passare al programma dei valori tramite riga di comando (di shell)

  ```
  C:\>stampav ciao come va
  ```



- Lo Standard li chiama in modo più generale "parametri di programma"

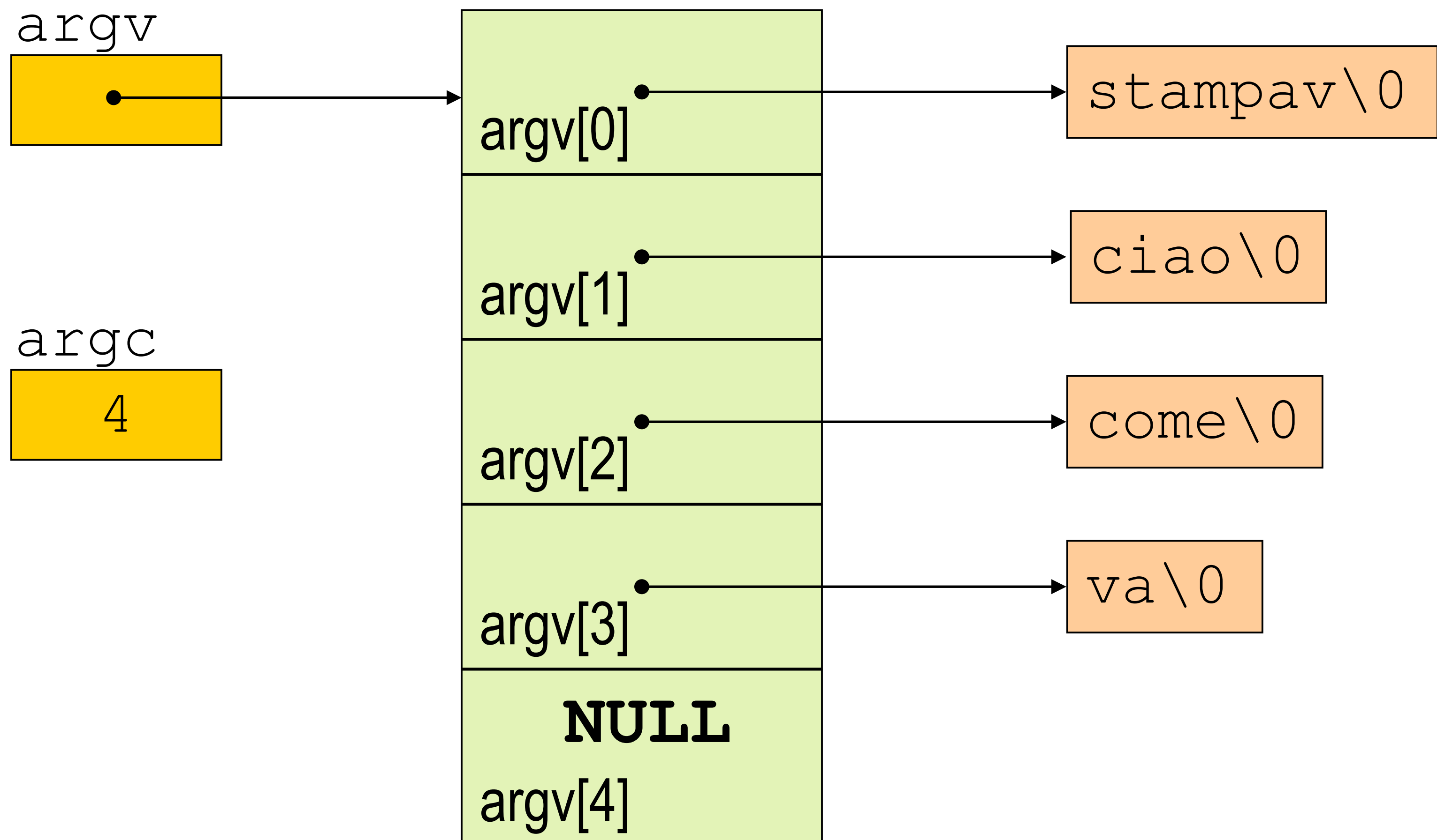
Argomenti nella riga di comando

- Bisogna definire il `main` in uno di questi modi:
  - `main(int argc, char *argv[])`
  - `main(int argc, char **argv)`

  Viene automaticamente creato un vettore di puntatori a stringhe, ciascuna delle quali contiene uno degli argomenti. In dettaglio:
  - `argc` è il numero di elementi sulla riga di comando, incluso il nome del comando
  - `argv` è un puntatore ad un vettore di puntatori a carattere: contiene gli indirizzi delle singole stringhe che costituiscono la riga di comando. L'ultimo elemento del vettore è sempre `NULL`

Argomenti nella riga di comando

- La struttura prodotta per `argv` è la seguente



Argomenti nella riga di comando

- Quindi nell'esempio:
  - `argc` è il numero di parametri, incluso il comando (il nome del programma), vale 4
  - `argv[0]` è il nome del programma ("`stampav`")
  - `argv[1]` è il primo parametro ("`ciao`")
  - `argv[2]` è il secondo parametro ("`come`")
  - `argv[3]` è il terzo parametro ("`va`")
  - `argv[argc]` è il terminatore `NULL`
- È importante verificare sempre che il numero di parametri effettivamente passati sia concorde con quanto richiede il programma
- Il nome dei parametri non è obbligato

Parsing della riga di comando

- Esempio di `stampav`: visualizza in colonna tutti gli argomenti passati sulla riga di comando (1° modo)

```
int main(int argc, char *argv[])
{
  int i;
  for (i=1; i<argc; i++)
    printf("%s\n", argv[i]);
  return EXIT_SUCCESS;
}
```

N.B. Questo programma funziona con qualsiasi numero di argomenti

Parsing della riga di comando

- Esempio di `stampav`: visualizza in colonna tutti gli argomenti passati sulla riga di comando (2º modo)

```
int main(int argc, char *argv[])
{
  char **p;
  for (p=&argv[1]; *p!=NULL; p++)
    printf("%s\n", *p);
  return EXIT_SUCCESS;
}
```

N.B. Questo programma funziona con qualsiasi numero di argomenti

Parsing della riga di comando

- Esempio di `stampav`: visualizza in colonna tutti gli argomenti passati sulla riga di comando (3º modo)

```
int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s\n", *++argv);
  return EXIT_SUCCESS;
}
```

N.B. Questo programma funziona con qualsiasi numero di argomenti, ma modifica `argc` e `argv`

Parsing della riga di comando

- Altro esempio: programma che elabora il file indicato sulla riga di comando (se c'è)

```
int main(int argc, char *argv[])
{
  FILE *fp;
  if (argc == 2)  ← controllo n. argomenti
  {
    if ((fp=fopen(argv[1],"r"))== NULL)
    {
      fprintf ("File %s non aperto\n",
                  argv[1]);
      return EXIT_FAILURE;
    }
```

Parsing della riga di comando

- Gli argomenti sulla riga di comando sono stringhe, per ottenere il valore numerico nel caso si trattino di numeri, si possono usare le già citate funzioni seguenti (`<stdlib.h>`):
  - *varInt* = `atoi`(*stringa*)
    converte *stringa* in un valore `int`
  - *varLong* = `atol`(*stringa*)
    converte *stringa* in un valore `long`
  - *varDouble* = `atof`(*stringa*)
    converte *stringa* in un valore `double`
  - Sono da preferire le funzioni `strto`(`l`,`ul`,`d`) in quanto le precedenti danno 0 in caso di errore

Volatile

- La keyword `volatile` davanti a una definizione di variabile richiede al compilatore di non ottimizzare l'accesso alla variabile, quindi di ricaricarla dalla memoria ogni volta che viene utilizzata

```
volatile int *clock;
```

- Tipicamente questo è necessario se la variabile viene aggiornata non dal programma stesso, ma dal sistema (es. una locazione di memoria che contiene l'indicazione dell'ora del sistema, aggiornata dal clock di sistema)

Register

- La keyword `register` davanti alla definizione di variabile automatica richiede al compilatore di memorizzarla nei registri del processore in quanto sarà usata frequentemente, *se possibile*
- Delle variabili `register` non si può ricavare l'indirizzo di memoria con & se queste vengono realmente allocate in un registro
- I compilatori sono in grado di individuare autonomamente se una variabile è da mettere in un registro, in ogni caso `register` può aiutarli a ottimizzare il codice in quanto indica che non verrà modificata tramite puntatori

Assert

- Macro per il debug definita in `<assert.h>`
  `void assert(int` *condizione*`);`
- La *condizione* viene valutata al run-time, se è diversa da 0 il programma continua
- Se la *condizione* è pari a 0 (condizione fallita):
  - viene mandato su `stderr` (in console mode) un messaggio contenente: la *condizione* stessa, il nome del file e il numero della riga dove è stata valutata l'`assert`
  - viene invocata la funzione `abort` per terminare il programma con segnalazione di anomalia
- Tutte le macro `assert` possono essere disabilitate definendo la costante `NDEBUG` prima della `#include<assert.h>`

Abort

- Termina un processo in modo anomalo e manda un segnale al sistema operativo, restituisce un *exit code* di valore 3
  `abort();`
- Non restituisce alcun valore
- Non fa il flush dei buffer e non chiama le funzioni di `atexit`
- Definita in `<stdlib.h>`

Atexit

- Permette di far sì che, quando il programma termina normalmente (`return`, `exit`), chiami automaticamente una o più funzioni (ad es. per rilasciare memoria, chiudere file, ecc.)
- I nomi delle funzioni da chiamare sono registrati uno alla volta con chiamate:
  `atexit(`*nomeFunzione*`)`
- *nomeFunzione* è l'indirizzo di una funzione (nome o puntatore) con prototipo
  `void` *nomeFunzione*`(void)`
- Ogni chiamata registra una funzione, le funzioni vengono chiamate in ordine LIFO, dà 0 se ha successo, ≠0 se c'è qualche errore
- Definita in `<stdlib.h>`

Atexit

```
#include <stdlib.h>
#include <stdio.h>
void fn1() { printf("dopo\n"); }
void fn2() { printf("viene "); }
void fn3() { printf("Questo "); }

main()
{ atexit(fn1);
  atexit(fn2);
  atexit(fn3);
  printf("Questo viene prima\n");
}
```

Esercizi

1. Scrivere un programma denominato `calcola` che richieda 3 parametri sulla riga di comando:
   1. operando 1
   2. operatore (`+`,`-`,`*`,`/`)
   3. operando 2

   e calcoli: *op1 oper op2*.
   In caso di errore (numero parametri scorretto o operatore non riconosciuto) lo segnali.

   Esempio

   ```
   c:\> calcola 12 * 5
   60
   ```

Esercizi

2. Scrivere un programma denominato `cat` che concateni tutti i file (di testo) passati sulla riga di comando (tranne l'ultimo) nell'ultimo file indicato:
   `cat` *file1 file2* ... *file*
   concatena *file1*, *file2*, ... in *file*

   N.B. i file da concatenare potrebbero non avere il ritorno a capo in fondo all'ultima riga, lo si aggiunga se manca (non per l'ultimo file da concatenare)

Salti non locali

- Permettono a una funzione annidata a livelli profondi di restituire direttamente il controllo a un livello più esterno del precedente
- Esempio:
  `a()` chiama `b()` che chiama `c()` che chiama `d()`

  Un salto non locale permette ad esempio a `d()` di tornare immediatamente ad `a()` senza passare dalle funzioni intermedie `c()` e `b()`
- Questo metodo viene utilizzato soprattutto per la gestione degli errori e dei segnali (ad es. divisione per 0, interrupt, etc.)
- Funzioni e strutture dati sono in `<setjmp.h>`

Salti non locali

- Lo stato attuale del programma (alcuni registri della CPU, tra cui Program Counter, Stack Pointer e Frame Pointer) viene salvato dalla funzione `setjmp` in una variabile di tipo `jmp_buf`
- Il salto non locale avviene eseguendo la funzione `longjmp` che ripristina lo stato antecedente alla chiamata a `setjmp` utilizzando i dati memorizzati nella variabile: quindi l'esecuzione torna alla `setjmp` e prosegue dall'istruzione successiva ad essa
- Non è un'istruzione strutturata

Preparazione al salto

- `int setjmp(jmp_buf` *env*`)`
  non solo memorizza lo stato del sistema, ma imposta anche il punto di arrivo del salto tramite la variabile indicata (*env*), questa deve essere di tipo `jmp_buf`
  La chiamata a `longjmp` ritornerà a questo punto e l'esecuzione continuerà dall'istruzione successiva
  Per poter distinguere se è stata eseguita la `setjmp` o invece si sta tornando dal salto si utilizza il valore di ritorno:
  - 0 se è stata chiamata per impostare il punto
  - ≠0 se è di ritorno dall'esecuzione della `longjmp`

Preparazione al salto

- Si noti che la funzione che contiene la `longjmp` deve poter accedere alla variabile *env*, quindi questa:
  - viene passata di funzione in funzione fino a quella che contiene la `longjmp`
  - oppure viene definita come variabile esterna

Esecuzione del salto

- `void longjmp(jmp_buf` *env*`, int` *val*`)` ripristina lo stato salvato in *env*, l'esecuzione continua come se fosse appena stata eseguita la `setjmp` che però ora non restituisce 0, ma *val* (che non deve essere pari a 0)
- Poiché viene ripristinato lo stack alla situazione precedente la chiamata a `setjmp`, lo stack stesso viene liberato da tutte le allocazioni (variabili) dovute alle chiamate di funzione successive alla chiamata `setjmp`

Schema della setjmp

- if (**setjmp(env) == 0**)

```
    /*
```

esegue questo blocco se si tratta della chiamata diretta che salva lo stato

```
    */
else
    /*
```

esegue questo blocco se è di ritorno dalla `longjmp`

```
    */
```

Esempio

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf buf;
void f2()
{ longjmp(buf,1); }
void f1()
{ f2(buf); }

int main()
{
  if (setjmp(buf)==0)
  {
    printf("ESEGUITA CHIAMATA SETJMP\n");
    f1(buf);
  }
  else
    printf("RITORNO DIRETTO DA f2\n");
}
```

Gestione dell'errore

- La `setjmp` permette di isolare (proteggere) blocchi di codice che possono generare problemi e farli seguire da un blocco di codice per il trattamento del caso di errore

```
if (setjmp(buf)==0)
{
    elaborazione...
    if (errore)
      longjmp(buf,1);
}
else
{gestione errore del blocco A}
```

Restrizione sulla setjmp

- Le variabili *automatiche* che *sono state modificate* dopo la chiamata a `setjmp`, al ritorno per effetto di una `longjmp` hanno contenuto indefinito
- Per preservarne il contenuto si deve usare la clausola `volatile` davanti alle definizione (questo indica al compilatore di non fare alcuna ottimizzazione, ad es. di non collocarle nei registri):

```
volatile int x;
struct nodo * volatile sp;
```

Funzioni variadic

- Le funzioni possono avere un numero variabile di parametri sono dette *funzioni variadic* o *funzioni varargs*
- *tipo* *nomeFunzione*(*arg1*, *arg2*, ...);
- L'ellissi ... (tre punti) indica che ci *possono* essere altri parametri dopo quelli fissi
- Deve esserci almeno un parametro fisso
- L'ellissi può essere indicata solo come ultimo elemento di una lista di parametri
- Le macro e le definizioni da utilizzare sono indicate in `<stdarg.h>`

Inizializzazione variadic

- All'interno di una funzione variadic bisogna definire una variabile di tipo `va_list`, questa è un *puntatore* e punterà in sequenza a ciascuno degli argomenti aggiuntivi ("anonimi"), elencati dopo quelli fissi:
  `va_list` *argP*`;`
- La macro `va_start` inizializza la variabile *argP* in modo che punti al primo degli argomenti anonimi, è necessario fornire il nome dell'ultimo argomento fisso (nell'esempio seguente è ***arg2***):
  `va_start(`*argP*`,` ***arg2***`);`

Utilizzo

- La macro
  `va_arg(`*argP*`, `*tipo*`)`
  restituisce un valore del *tipo* indicato (viene applicato un cast) prelevandolo dall'elemento anonimo puntato da *argP* nella lista variabile, dopodiché modifica *argP* affinché punti al successivo elemento anonimo:
  `x = va_arg(`*argP*`,int);`
- Ogni chiamata a `va_arg` preleva il successivo valore dalla lista variabile

Terminazione

- Dopo che gli argomenti anonimi sono stati elaborati, per indicare che non si intende più scandire gli elementi di *argP*, si deve chiamare la macro `va_end` prima che termini la funzione che la chiama:
  `va_end(`*argP*`);`
- I parametri possono essere scanditi più volte: è sufficiente chiamare `va_end` e poi nuovamente `va_start`

Esempio

```
#include <stdarg.h>
#include <stdarg.h>
void stampa(int quanti, ...);
main()
{
   stampa(0);
   stampa(1, "END");
   stampa(2, "END", "with no exit code");
   stampa(3, "END", "with code ", 8);
   return 0;
}
```

Esempio

```
void stampa(int quanti, ...)
{
  va_list ap;
  va_start(ap, quanti);
  if (quanti >= 1)
    printf(" %s", va_arg(ap, char*) );
  if (quanti >= 2)
    printf(" %s", va_arg(ap, char*) );
  if (quanti >= 3)
    printf(" %d", va_arg(ap, int) );
  printf("\n");
  va_end(ap);
}
```

`quanti`: ultimo argomento fisso

Numero di argomenti anonimi

- Per indicare alla funzione quanti sono gli argomenti passati, si può:
  - passare tale valore come argomento fisso
  - terminare la lista degli argomenti facoltativi con un valore speciale (*sentinella*) di tipo opportuno
- Esempio
  Chiamata:

```
maxstr(4, s0,s1,s2,s3);
```

  Nella funzione si ha:

```
while (quanti-- > 0)
  printf("%s", va_arg(ap, char*));
```

Numero di argomenti anonimi

- Esempio
  Chiamata (tutti valori interi >0):
  ```
  sumposit(1, 2, 3, 0);
  ```
  Nella funzione si ha:
  ```
  while ((x=va_arg(ap, int)) != 0)
    printf("%s", x);
  ```
  sentinella
- Esempio
  Chiamata (tutte stringhe):
  ```
  maxstr(s0,s1,s2,s3,(char *)NULL);
  ```
  Nella funzione si ha:
  ```
  while ((p=va_arg(ap, char*)) !=NULL)
    printf("%s", p);
  ```
  notare che il cast di NULL corrisponde al tipo in va_arg

Tipi dei parametri

- Per i parametri anonimi non c'è alcuna indicazione di tipo nel prototipo della funzione, quindi si hanno le *usual arithmetic conversions* viste nelle espressioni:
  - per gli interi si applicano le promozioni integrali
  - un valore `float` viene convertito in `double`
- Una chiamata come `va_arg(`*`ap`*`,`**`float`**`)` non è corretta perché i `float` vengono convertiti in `double`

Tipi dei parametri

- Quindi per passare parametri:
  - `float` e `double` usare `double`
  - `char`, `int` e `short` usare `int`
  - vettori usare un puntatore

  Un puntatore sentinella a `NULL` deve avere il cast (come nel secondo esempio di `maxstr`)
- Attenzione: anche l'ultimo argomento fisso deve essere di un tipo che non subisce promozioni di default
- Per passare puntatori a funzione conviene definire il tipo della funzione con `typedef`

Esercizi

3. Scrivere la funzione
   `char *mstrcat(char *prima, ...)`
   che concateni un numero arbitrario di stringhe nella `prima` passata come argomento. Per il resto abbia lo stesso comportamento della `strcat`. Si scriva un `main` di prova.