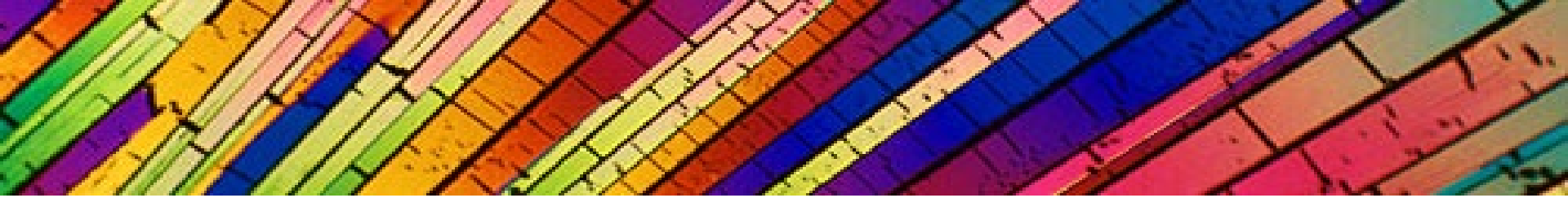

Il preprocessore C

Ver. 3

Funzionalità

- Il preprocessore modifica il codice C *prima* che venga eseguita la compilazione vera e propria
- La prima operazione effettuata è ridurre i commenti a un singolo spazio, quindi le direttive possono essere seguite da commenti
- Poi esegue le *direttive di preprocessing* per:
  - l'inclusione di file (`#include`)
  - la definizione di simboli (`#define`)
  - la sostituzione di simboli (`#define`)
  - la compilazione condizionale (`#if`, …)
  - la definizione di macroistruzioni con parametri (`#define`)

Funzionalità

- Le *direttive di preprocessing* iniziano sempre con il carattere # (è preferibile che non sia seguito da uno spazio)
- Non essendo istruzioni C non hanno il ‘;’ finale
- Una direttiva non può generare un’altra direttiva
- Il preprocessore del C (cpp) può essere un programma esterno al compilatore chiamato dal compilatore stesso; oppure può essere realizzato internamente al compilatore stesso, ma dal punto di vista concettuale non ci sono differenze

Inclusione di file

- La riga con `#include` viene sostituita dal contenuto testuale del file indicato
  `#include <stdio.h>`
- Il file incluso è di norma un header file (`.h`) e contiene la definizione di simboli (e in generale altre direttive del preprocessore), prototipi, dichiarazioni di variabili `extern` e di tipi di dati (`typedef`)
- È buona norma non includere file contenenti codice eseguibile (file `.c`)

Inclusione di file

- Il nome del file che segue la direttiva può avere due forme:
  - `#include <`*file*`.h>`
    Il file viene cercato nelle directory standard degli include file del compilatore (il compilatore ha opzioni per includere altre directory tra queste)
  - `#include "`*file*`.h"`
    Il file viene cercato *prima* nella directory dove si trova il file C e *poi*, se non trovato, nelle directory del compilatore (come sopra)

Inclusione di file

- Il nome del file può essere completo di percorso, diversi sistemi operativi usano diversi caratteri (es. '`/`' e '`\`') per separare i nomi delle directory, ma alcuni compilatori li convertono automaticamente
- È assolutamente sconsigliabile utilizzare indicazioni specifiche di un sistema operativo (es. indicare il nome del drive in Windows)
- Ciò che segue la `#include` non è un letterale stringa quindi il simbolo '`\`' non è interpretato come escape (e non deve essere raddoppiato)

Inclusione di file

- La `#include` può anche essere seguita da un simbolo definito precedentemente con una `#define`, può essere utile con le direttive condizionali):

```
#define STDIO "teststdio.h"
…
#include STDIO
```

- Le direttive `#include` vengono *in genere* collocate in testa a ciascun file sorgente C, prima della prima funzione

Inclusione di file

- I file inclusi possono a loro volta contenere altre direttive `#include`
- L'inclusione multipla di uno stesso file in genere non genera problemi se contiene solo macro, prototipi e dichiarazioni `extern`, mentre genera problemi se definisce dei tipi (il compilatore segnala la tentata ridefinizione)
- Per evitare l'inclusione multipla dei propri file header (quelli indicati tra virgolette) si usano le direttive condizionali (vedere più avanti)

Definizione di simboli

- `#define` *nome*
  definisce l'*esistenza* del simbolo *nome*
  `#define DEBUG`
- I simboli (*macro name*) sono utilizzati da altre direttive del preprocessore, ad es. da `#if`
- *nome* è un identificatore e viene per convenzione scritto tutto in maiuscolo
- Lo scope di *nome* si estende dalla riga con la `#define` (*di solito* all'inizio del file) fino alla fine del file sorgente dove è definito e non tiene conto della struttura a blocchi del codice
- `#undef` *nome*
  annulla una `#define` precedente

Macro semplici

- Definiscono un macro name e il suo valore:

  `#define` *nome elenco_di_sostituzione*
- Lo Standard le chiama "*object-like macro*"
- In ogni punto del programma dove *nome* appare come identificatore (ossia sembra una variabile), il preprocessore semplicemente lo *sostituisce* con *elenco_di_sostituzione*
- Poiché la sostituzione avviene prima della vera e propria compilazione, il codice:

```
#define MAX 100
for (i=0; i<MAX; i++)
```

  dopo il passaggio del precompilatore diventa:

```
for (i=0; i<100; i++)
```

Macro semplici

- *nome* non può essere definito più volte nella stessa translation unit (a meno che la ridefiniz. sia identica alla precedente o si usi `#undef`):
- *elenco_di_sostituzione*:
  - è una sequenza di caratteri: può essere un numero (`10`), una stringa ("`abc`"), un'espressione (`x+y`)
  - termina a fine riga
  - può contenere spazi e commenti
  - può utilizzare *nomi* (simboli) di altre `#define`
  - se manca, si ha la semplice definizione di *nome* e tutte le occorrenze di *nome* sono eliminate
  - può continuare su più righe purché ogni riga trane l'ultima termini con il carattere '`\`' (dal punto di vista logico si tratta di una sola lunga riga)

Macro semplici

- Quando *elenco_di_sostituzione* è un valore costante numerico o stringa viene chiamato anche "costante manifesta" (dal K&R, non dallo Standard)
- Dopo la sostituzione (detta anche "espansione"), le righe con le `#define` vengono ridotte a righe vuote
- Se dopo *nome* (minuscolo) c'è una coppia di parentesi VUOTE (senza spazi intermedi), si ha una macro semplice (non parametrica) che *assomiglia* a una funzione:

```
#define getchar() getc(stdin)
```

Macro semplici

- L'*elenco_di_sostituzione* può contenere l'invocazione di altre macro
- Le sostituzioni vengono ripetute tutte più volte finché non c'è altro da sostituire (nell'ordine indicato)

```
#define C 3
#define B C
#define A B
```

  trasforma *ogni* `A`, `B` e `C` in `3`
- Se un *nome* è stato espanso e compare *nuovamente* per effetto di altre espansioni, non viene ri-espanso

Macro semplici

- Attenzione: essendo l'espansione di macro una *semplice sostituzione di caratteri*, si può incorrere in errori
- Ad esempio il seguente ciclo non va fino a 202

```
#define MAX 100+1
for (i=0; i<MAX*2; i++)…
```

ma fino a 102 in quanto la sostituzione produce il seguente codice:

```
for (i=0; i<100+1*2; i++)…
```

Per evitare problemi come questo, nella `#define` si indica il valore tra parentesi :

```
#define MAX (100+1)
```

Macro parametriche

- Sono macro con argomenti e sembrano funzioni (lo Standard le chiama proprio "*function-like macro*")
- Definiscono un simbolo *nome* seguito da una coppia di parentesi contenenti argomenti separati da virgole, questi vengono usati come parametri nell'*elenco_di_sostituzione* che, inquesto caso, ha la forma di un'espressione:

```
#define max(A,B) ((A)>(B)?(A):(B))
```

- Nella parte sottolineata NON possono esserci spazi, possono esserci nell'*elenco*

Macro parametriche

- Definita la macro parametrica

```
#define MAX(A,B) ((A)>(B)?(A):(B))
```

quando viene richiamata nel codice, i *simboli* `A` e `B` della macro `MAX` vengono sostituiti dagli argomenti indicati nella chiamata (che può contenere spazi):

```
x = MAX(d, f);
```
viene espansa in
```
x = ((d)>(f)?(d):(f));
```

- Le chiamate di macro possono essere annidate:

```
massimo = MAX(a, MAX(b, c));
```
diventa:
```
massimo = ( (a)>(((b)>(c)?(b):(c)))
? (a) : (((b)>(c)?(b):(c))));
```

Macro parametriche

- Le macro parametriche sostituiscono simboli per cui funzionano con qualsiasi tipo di dato
- L'espansione di una macro non è una chiamata di funzione perché non fa altro che inserire lo stesso codice in punti diversi:
  - l'esecuzione è più veloce (la chiamata di funzione richiede tempo)
  - ma il codice dopo la precompilazione e il programma eseguibile risultante sono più grandi
- In C99 le *funzioni inline* sono l'alternativa migliore per avere un'esecuzione più veloce

Macro parametriche - parentesi

- Come già visto per le macro semplici, a maggior ragione per le macro parametriche nell'*elenco_di_sostituzione* è bene mettere sempre i parametri tra parentesi
- Ad esempio:
  - `#define quadrato(x) x*x`
    chiamata come `quadrato(z+1)` viene *erroneamente* sostituita da `z+1*z+1`
  - `#define quadrato(x) (x)*(x)`
    chiamata come `quadrato(z+1)` viene *correttamente* sostituita da `(z+1)*(z+1)`

Macro parametriche - parentesi

- Altri esempi

```
#define max(A,B)  ((A)>(B)?(A):(B))
```

  - la chiamata `max(*p,*q)`
    viene correttamente sostituita da
    `((*p)>(*q)?(*p):(*q))`
    ma anche senza parentesi avrebbe funzionato
  - la chiamata `max(p+q, r+s)`
    viene *correttamente* sostituita da
    `((p+q)>(r+s)?(p+q):(r+s))`
    ma senza parentesi ci sarebbe stato il confronto tra `q` e `r`, evidentemente errato

Macro parametriche - side effec

- Attenzione agli effetti collaterali dovuti alla ripetizione dei parametri nell'espressione
- Ad esempio

  `max(i++,j++)`

  viene sostituita da

  `((i++)>(j++)?(i++):(j++))`

  che è *illecito* in C (e se fosse lecito il valore più grande tra `i` e `j` verrebbe erroneamente incrementato due volte)

Macro parametriche -

- In *elenco_di_sostituzione* il nome di un parametro preceduto da `#` produce un letterale stringa ("*stringization*") tra virgolette contenente il *valore* dell'argomento della macro, convertendo i simboli `"` in `\"` e `\` in `\\` ed eliminando eventuali spazi iniziali e finali
- Ad esempio, definendo:

```
#define p(e) printf(#e "=%g\n", e)
```

la chiamata della macro: `p(x/y)`
viene sostituita da:

```
printf("x/y" "=%g\n", x/y)
```

e le due stringhe vicine vengono concatenate dal compilatore in `"x/y=%g\n"`

Macro parametriche -

- Il parametro della macro che nell'*elenco_di_sostituzione* è nella forma `#param` se è un simbolo esso stesso, allora non viene espanso:

```
#define N 10
#define str(s) #s
```

alla chiamata

```
str(N) produce "N"
```

Per avere l'espansione del simbolo (ossia avere `"10"`) bisogna attuare due sostituzioni:

```
#define xstr(s) str(s)
```

alla chiamata

```
xstr(N) produce "10"
```

Valutazione delle macro

- Le macro sono valutate in questo modo:

1. elabora i token preceduti da `#` o relativi a `##`
2. esegue la sostituzione delle macro per ciascun argomento
3. sostituisce ogni parametro con il risultato della precedente sostituzione
4. ricomincia da 1. (ma se trova la stessa macro non la espande nuovamente)

Valutazione delle macro

- Nel caso precedente, `xstr(N)` non contiene nella stringa di sostituzione né `#` né `##` quindi non viene eseguito il punto 1 e passa ai punti 2 e 3 dove `xstr(N)` viene espansa in `str(10)`
- Ora per effetto del punto 4. torna al punto 1 dove `str(10)` viene riconosciuta come macro e quindi espansa in `"10"`
- Il problema di usare solo `str(N)` è il fatto che il punto 1 agisce sul parametro prima della sostituzione del punto 2, quindi serve una macro ausiliaria per bypassarlo

Macro parametriche -

- In *elenco_di_sostituzione* l'operatore di preprocessore `##` (detto anche *token-pasting*, ossia *incolla-token*) posto tra due argomenti di macro li concatena rimuovendo gli spazi intermedi e producendo un nuovo *token* (elemento che il compilatore riconosce)
- Ad esempio la seguente

```
#define JOIN(x,y) x ## y
```

la chiamata della macro

```
printf(JOIN(vett,6)[i]);
```

viene espansa in:

```
printf(vett6[i]);
```

Macro parametriche -

- Le macro che nell'*elenco_di_sostituzione* contengono l'operatore ## non possono essere annidate in quanto i simboli che precedono e seguono ## non vengono espansi

Macro parametriche senza arg.

- In C99 nella chiamata di una macro parametrica si possono omettere argomenti (ma non le virgole di separazione)
- Il parametro corrispondente all'argomento mancante viene sostituito da nulla (eliminato)
- Nel caso dell'operatore `#`, viene creata una stringa vuota `""`
- Nel caso dell'operatore `##`, viene creato un token fittizio (placeholder), rimosso alla fine della concatenazione. Ad esempio:
  `#define JOIN(x,y,z) x##y##z`
  `JOIN(wh,,ile)` *produce il token* `while`

Macro parametriche var. arg.

- In C99 è possibile definire macro con un numero variabile di argomenti
- Utile per passare argomenti a una funzione che richiede un numero variabile di argomenti
- Nella definizione del *nome* si indica come ultimo parametro il token … (*ellissi*), dopo quelli fissi (se ce ne sono)
- Nell'*elenco_di_sostituzione* la macro `__VA_ARGS__` rappresenta tutti gli elementi che corrispondono all'ellissi
- Se per … non viene specificato nulla, `__VA_ARGS__` è vuoto

Macro parametriche e __func__

- Si è già visto nelle funzioni che in C99 la costante `__func__` contiene il nome della funzione dove viene utilizzata
- Per le operazioni di debug possono essere utili le definizioni seguenti, da mettere come prima e ultima istruzione rispettivamente nelle funzioni che si vogliono monitorare:

```
#define ENTRA() \
  printf("%s chiamata\n", __func__
#define ESCE() \
  printf("%s uscita\n", __func__
```

Macro complesse

- Il corpo della macro (*elenco_di_sostituzione*) può contenere più istruzioni C
- Ad esempio

```
#define swap(x,y) t=x; x=y; y=t
```

  la chiamata della macro:

```
swap(a,b);
```

  viene espansa in:

```
t=a; a=b; b=t;
```

- Si noti che il `;` in fondo alla chiamata è quello che si ritrova in fondo all'espansione ed è quindi necessario, inoltre `t` deve essere stato definito precedentemente

Macro complesse

- Per poter scrivere il corpo della macro su più righe, si devono terminare tutte tranne l'ultima con il carattere `\` (non seguito da spazi)
- Ad esempio

```
#define swap(x,y) \
    t=x; \
    x=y; \
    y=t
```

← *nessun ;*

- L'espansione è identica alla precedente, valgono tutte le considerazioni fatte nella slide precedente su `t` e il `;`

Macro complesse

- Se serve definire una variabile temporanea per la sola macro, si crea un blocco con una coppia di parentesi graffe dove definire una variabile locale al blocco
- Ad esempio

```
#define swap(x,y) \
        { int t; t=x; x=y; y=t; }
```

Macro complesse

- Questa soluzione non è sempre valida, ad es.

```
if (test)
   swap(a,b);
else
   …
```

viene espansa in

```
if (test)
   { int t; t=x; x=y; y=t; };
else
   …
```

dove il `;` finale chiude l'`if` e l'`else` resta senza `if`, quindi il compilatore darebbe errore

Macro complesse

- Si potrebbe chiamare la `swap` senza il `;` finale, ma apparirebbe "anomalo"
- La soluzione migliore è definire un ciclo fittizio

```
#define swap(x,y)                \
do {                             \
   int t; t=x; x=y; y=t;         \
}while (0)   ← qui si omette il ;
```

Macro complesse

- In questo caso si ottiene:

```
if (test)
   do {                                \
      int t; t=x; x=y; y=t; \
   }while (0);
else
   …
```

dove il `;` finale, proveniente dalla chiamata alla `swap`, chiude solo l'unica istruzione **`do-while`** e non il ramo then

Macro complesse

- Si noti che quando si usano le macro come fossero funzioni, i parametri vengono eventualmente modificati perché la macro agisce proprio su questi e non su copie locali

Macro complesse

- È possibile far in modo che una macro composta restituisca un valore, proprio come una funzione, sfruttando l'operatore virgola (perché dà come valore quello dell'espressione più a destra, quindi l'ultimo dei calcoli), il tutto messo tra parentesi, altrimenti dà il primo valore (vedere le slide sull'operatore virgola)
- Ad esempio, la funzione seguente scambia `x` e `y` e restituisce il maggiore dei due:

```
#define maxswap(x,y) \
          (t=x, x=y, y=t, x>y?x:y)
```

chiamata come: `k = maxswap(a,b);`

Direttive condizionali

- Permettono di include o escludere parti di codice dalla compilazione e dal preprocessing

  `#if` *espressione_1*
  istruzioni
  `#elif` *espressione_2*
  istruzioni
  `...`
  `#else`
  istruzioni
  `#endif`

- Solo uno dei gruppi di *istruzioni* sarà elaborato dal preprocessore e poi compilato: il primo la cui *espressione* è vera

Direttive condizionali

- Le *espressioni* devono essere espressioni costanti intere o confronti (sono riconosciuti gli operatori matematici, relazionali e logici), sono considerate vere se diverse da 0, non possono contenere `sizeof`, cast o costanti `enum`
- La `#if` considera un simbolo non definito come 0, quindi se `DEBUG` non è definito

  `#if DEBUG`

  viene considerata falsa, mentre

  `#if !DEBUG`

  viene considerata vera

Direttive condizionali

- L'operatore `defined(`*nome*`)` produce 1 se *nome* è stato definito (con `#define`), 0 altrimenti, può essere usato anche per i simboli solo definiti ma senza valore associato, come in `#define DEBUG`
- `#if defined(`*nome*`)` e la forma abbreviata
  `#ifdef` *nome*
  verificano che *nome* sia definito
- `#if !defined(`*nome*`)` e la forma abbreviata
  `#ifndef` *nome*
  e verificano che *nome* NON sia definito
- `#elif` ed `#else` hanno significato ovvio

Direttive condizionali

- Possono essere usate per isolare le istruzioni da usare solo per il debug del programma:

```
#ifdef DEBUG
  printf("Valore di x: %d\n", x);
#endif
```

Una direttiva

```
#define DEBUG
```

posta a inizio del programma controlla l'esecuzione di `printf` di debug come queste, a programma ultimato basta eliminare la `#define` e tutto il codice viene eliminato. `DEBUG` talvolta viene definito dal compilatore

Direttive condizionali

- Per evitare che un header file sia incluso più volte (può capitare quando un header ne include altri), si può usare lo schema seguente (quello che segue è il file `hdr.h`):

```
#ifndef HDR
#define HDR
... definizioni specifiche di hdr.h
#endif
```

- Se `hdr.h` venisse incluso una seconda volta, il simbolo `HDR` sarebbe già definito e il contenuto di `hdr.h` non verrebbe nuovamente incluso nella compilazione

Direttive condizionali

- Per far *compilare* parti diverse a seconda del sistema operativo, si può usare lo schema:

```
#ifdef LINUX
   …
#elif WINDOWS
   …
#else
   …
#endif
```

Un'opportuna `#define` iniziale farà selezionare e compilare solo una delle tre porzioni di codice. Alcuni compilatori definiscono simboli appropriati proprio per questo uso

Direttive condizionali

- Per dare un valore di default a un simbolo si può usare lo schema seguente:

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

- Per escludere dalla compilazione un grosso blocco di codice (anche con commenti):

```
#if 0
    codice da non eseguire
#endif
```

  Ma il codice saltato deve avere i commenti chiusi correttamente perché questi sono gestiti prima delle direttive

Macro predefinite

- Sono simboli definiti dal preprocessore:

`__DATE__` Data di compilazione del file sorgente nel formato di *"Mmm dd yyyy"*

`__FILE__` Letterale stringa con il nome del file .c

`__LINE__` Numero di linea della riga corrente nel file sorgente, può essere alterato da `#line`

`__STDC__` Indica che il compilatore è del tutto aderente allo standard ANSI (C89 e C99) e rifiuta eventuali estensioni

`__TIME__` Ora di compilazione del file sorgente nella forma *"hh:mm:ss"*.

`__TIMESTAMP__` Data e ora di compilazione del file

Macro predefinite

- La direttiva `#line` seguita da un numero intero imposta il valore di `__LINE__` della riga *successiva* del codice C, la numerazione delle righe successive prosegue la numerazione:

  ```
  #line 100
  ```

  la riga successiva sarà considerata la 100
- Il numero deve essere tra 1 e 2^{15}-1 (32767) in C89, tra 1 e 2^{31}-1 (2.14×10^9) in C99
- `#line` può impostare il valore di `__FILE__`:

  ```
  #line 100 "file1.c"
  ```

  Questo vale per le righe a partire da quella successiva. Utile per il debug.

Macro C99 predefinite

- Il C99 ha altre macro predefinite, tra queste:

  `__STDC_VERSION__` È un valore `long` che vale 199409L per il C89 (Amendement 1) e 199901L per il C99, è la data di revisione dello standard, revisioni successive hanno valori diversi

  `__STDC_HOSTED__` L'implementazione del linguaggio è detta *hosted* se è perfettamente conforme allo standard C99, la macro vale 1; al contrario l'implementazione è *freestanding* (usata per sistemi senza sistema operativo quali si sistemi embedded e il kernel), vale 0

  `__SDC_IEC_599__` È definita e vale 1 se l'aritmetica floating point utilizza lo standard IEC 60599 (ossia IEEE 754)

Direttiva nulla

- La direttiva nulla è una `#` in una riga a sé stante, equivale ad una riga vuota
- Utilizzabile per distanziare le varie parti di una direttiva `#if`

```
#if INT_MAX < 100000
#
#error int type is too small
#
#endif
```

Direttiva #error

- La direttiva `#error` serve per stampare un messaggio di errore contenente al suo interno un *testo* specificato (non servono le virgolette)
  `#error` *testo*
- La forma dell'output dipende dal compilatore
- Molti compilatori fermano la compilazione
- L'uso tipico è con le direttive condizionali, ad es. per evitare che il codice sia compilato su un sistema con `int` a 16 bit si può scrivere:

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

Simboli con nomi di funzioni

- Un *nome* espanso, anche se compare *nuovamente* per effetto di altre espansioni, non viene ri-espanso
- Allora se si scrive:

```
#define sqrt(x)((x)>0?=sqrt(x):0)
```

si dispone di una nuova `sqrt` che dà 0 se il valore è negativo: il preprocessore sostituisce il simbolo `sqrt` con l'istruzione condizionale, ma la `sqrt` in essa contenuta non viene nuovamente sostituita per cui, restando dopo il preprocessing, invocherà la funzione `sqrt`

Direttiva #pragma

- La direttiva `#pragma` serve per richiedere un comportamento speciale del compilatore
  `#pragma` *sequenza_di_token*
- La sintassi dipende dal compilatore
- In C89 non ci sono comandi standard per la direttiva `#pragma`, il preprocessore deve ignorare senza generare errore le direttive `#pragma` che non conosce
- In C99 ci sono 3 comandi per la direttiva `#pragma`, tutti preceduti da `STDC`:
  FP_CONTRACT, CX_LIMITED_RANGE, FENV_ACCESS (vedere lo Standard)

L'operatore _Pragma

- In C99 l'operatore `_Pragma(`*letterale_stringa*`)` elimina dal *letterale_stringa* le virgolette e le sequenze di escape e quindi crea una direttiva `#pragma` seguita dal risultato
- Permette di creare una direttiva `#pragma` a partire da una non-direttiva (da un'altra direttiva non sarebbe possibile)
- Ha utilizzo altamente specializzato (es. nel codice del compilatore stesso)

Esercizi

1. Scrivere una macro `swap(t,x,y)` che scambi il valore di due argomenti di tipo `t`.
2. Scrivere una macro `printArray(v,n)` che visualizzi il contenuto del vettore `v` di lunghezza `n`.
3. Scrivere una macro `sumArray(v,n,sum)` che sommi gli `n` valori del vettore `v` e metta il risultato in `sum`.