Il preprocessore

Il preprocessore (1)

- E’ un macroespansore, che processa i file sorgente prima di passarli al compilatore.
  - Il compilatore non vede il file originale, ma solo quello generato dal preprocessore.
  - Nato come programma separato, è ora di solito integrato col compilatore.
  - Le parole riservate del preprocessore **non sono parole riservate in C** e viceversa.
- Non conosce il C: esegue solo **sostituzioni testuali**, senza preoccuparsi del contesto.
  - Tuttavia riconosce e ignora i commenti e le stringhe.

Il preprocessore (2)

- Considera direttive tutte e sole le linee che iniziano con #.
  - Tratta le altre come testo da processare.
- Una direttiva inizia con una parola chiave.
  - Prima del # e tra # e la parola chiave si possono mettere spazi e tabulazioni orizzontali.

Direttive per il preprocessore

- Il preprocessore accetta 5 tipi di direttive:
  - inclusione di file,
  - compilazione condizionale,
  - definizione di macro,
  - generazione di errori,
  - direttive per il compilatore.

Inclusione di file

Inclusione di file (1)

- Ha due sintassi:
  `# include <pathname>`
  `# include "pathname"`

- La direttiva viene **sostituita** dal testo del file indicato.
  - Il file deve contenere a sua volta codice C.
- Il file da includere può a sua volta contenere direttive di inclusione.
  - I compilatori accettano almeno 15 livelli di inclusioni: normalmente ci si limita a pochi.

Inclusione di file (2)

- La differenza tra le due forme sta in come inizi la ricerca per il pathname:
  - nel caso <pathname>: in un insieme di directory dipendenti dal particolare sistema (Es.: /usr/include in Linux);
  - nel caso "pathname": si cerca prima a partire dalle directory specificate come opzioni al compilatore (per default la directory corrente), poi come nel caso precedente.

Come organizzare le inclusioni

- I path per le inclusioni devono essere **relativi** alla directory di componente o di progetto, **mai assoluti**.
  - Si deve usare la forma <pathname> solo per le librerie standard o comunque separate dal progetto.
  - Si deve usare la forma "pathname" per gli header file che fanno parte del progetto.

Header file

- Sono file di dichiarazioni, che vengono inclusi in più sorgenti.
  - Normalmente il nome del file termina con “.h”.
- Possono contenere dichiarazioni di costanti, tipi, variabili e funzioni, ma **non definizioni** di variabili e funzioni.
  - Essendo inclusi in più file, provocherebbero un errore di definizione multipla.
  - Normalmente per ogni sorgente xyz.c si scrive un file xyz.h con le interfacce esportate, per chi deve utilizzare le funzioni di xyz.c.
    - Il file xyz.h va incluso anche in xyz.c, percontrollare la coerenza tra dichiarazioni e definizioni.

File per le funzioni inline

- Le funzioni `inline` che devono essere utilizzate da file sorgenti diversi vanno messe in file separati.
  - Non esiste una convenzione universale, di solito il nome del file termina con “.in” o “.inl”.
- Devono contenere solo **definizioni** di funzioni **inline**.
  - Possono essere inclusi senza problemi in file differenti.
  - Normalmente per ogni sorgente xyz.c si scrive un file xyz.inl con le definizioni delle funzioni `inline` esportate, per chi le deve utilizzare.
  - Le dichiarazioni vanno in xyz.h, **senza** la parola chiave `inline`.

Come organizzare i file

- L’organizzazione più semplice e comune prevede una directory per ogni componente, a sua volta contenente:
  - una directory per i sorgenti;
  - una directory per gli header file;
  - una o più directory per gli oggetti;
    - una per ogni macchina per la quale si compila;
  - una o più directory per librerie ed eseguibili generati.
- In applicazioni grandi, mescolare file di tipo diverso provoca facilmente errori catastrofici.
- Un file sorgente deve contenere **solo funzioni correlate**, generalmente poche.

Nomi dei file

- Devono essere significativi, rispetto al contenuto.
- Per pulizia, i suffissi devono essere:
  - .c per i sorgenti;
  - .h per gli header file;
  - .inl per i file contenenti funzioni `inline`;
  - .o o .obj per gli oggetti (dipende dal sistema operativo);
  - lo standard suggerito dal sistema per librerie ed eseguibili.

Macro

Definizione di macro (1)

- La sintassi di una definizione di macro è:

  `# define` <*identificatore*> <*sequenza di token*>

- Tutte le **successive** occorrenze dell'identificatore, non fra apici (singoli o doppi) e non in commenti, sono sostituite con la sequenza di token.
  - Per andare a capo nella sequenza di token bisogna scrivere \ prima del terminatore di linea.
  - Per consuetudine, se l'espansione è una costante, l'identificatore si scrive con tutte maiuscole.

Definizione di macro (2)

- Definizioni multiple dello stesso identificatore sono ammesse solo se identiche.
- Non possono essere ridefinite le parole chiave del preprocessore.
  - Possono essere ridefinite le parole chiave del C (ma **non ha senso**).
- L'espansione non può generare una direttiva per il preprocessore.
  - Può generare parole chiave C.

Annullamento di definizione di macro

- La sintassi di un annullamento di macro è:

  `# undef` <*identificatore*>

- Annulla la definizione dell'identificatore; eventuali successive occorrenze non saranno sostituite.
- Serve ad annullare definizioni di default o importate da libreria, che possono creare problemi.

Esempi di macro (1)

```
# define INT_MAX  32767
```

```
i = INT_MAX;                  → i = 32767;
i = 2+INT_MAX;                → i = 2+32767;
printf("INT_MAX = ");         → invariato
// less than INT_MAX;         → invariato
n = SHORTINT_MAX;             → invariato: è un
                                altro identificatore
```

Esempi di macro (2)

```
# define forever  for(;;)
```

```
forever  ⟶  for(;;)
```

```
# define forever  while(true)
```
definizione doppia: errore di preprocessore

```
# undef  forever
# define forever  while(true)
```

```
forever  ⟶  while(true)
```

Macro predefinite

- Alcune macro sono predefinite e standard.
- Sono inoltre state standardizzate alcune macro condizionali, che sono definite se e solo se l'implementazione soddisfa certi requisiti.
- Tali macro possono essere usate, ma non definite o un-definite.
- La macro `__cplusplus` non deve essere definita dall'implementazione

Elenco delle macro predefinite

Macro	Valore
`__DATE__`	Data corrente (di compilazione)
`__FILE__`	Nome del file corrente
`__LINE__`	Numero di linea corrente
`__STDC__`	Costante intera 1
`__STDC_HOSTED__`	1, se l'implementazione è hosted, 0 altrimenti
`__STDC_VERSION__`	Costante intera `199901L`
`__TIME__`	Ora corrente (di compilazione)
`__VA_ARGS__`	Parametri variabili
`__func__`	Nome della funzione corrente

Elenco delle macro condizionali

Macro	Valore
`__STD_IEC_559__`	1, se l'implementazione dell'aritmetica reale è conforme allo standard IEC 60559
`__STD_IEC_559_COMPLEX__`	1, se l'implementazione dell'aritmetica complessa è conforme allo standard IEC 60559
`__STDC_ISO_10646__`	Costante intera della forma <yyyy><mm>L, se l'implementazione del tipo `wchar_t` è conforme allo standard IEC 10646, con tutte le modifiche sino all'anno e mese indicati

Esempio di macro predefinite

```
# define MAJOR_RELEASE_NUMBER  3
# define MINOR_RELEASE_NUMBER  1

(void)printf("Versione %u.%u del %s\n",
  MAJOR_RELEASE_NUMBER,
  MINOR_RELEASE_NUMBER, __DATE__);
```

Attenzione! La data è quella di **compilazione**, non di **esecuzione .**

Macro con parametri

- La sintassi di una definizione di macro con parametri è :

  `#` `define` <*identificatore*> (<*parametri formali*>)
  <*sequenza di token*>

- Si comportano come le altre macro, ma le occorrenze dei parametri formali nell'espansione sono rimpiazzate dai corrispondenti parametri attuali.
  - I parametri sono una lista di identificatori, separati da virgole.
  - Tra l'identificatore e la parentesi aperta non devono esserci caratteri, neppure spazi o tabulazioni.

Esempio di macro con parametri

```
# define max(a, b) ((a) > (b)? (a): (b))

x = max(y, 9);
```

```
x = ((y) > (9)? (y): (9));

f(max(p + q, r + s));
```

```
f(((p + q) > (r + s)? (p + q): (r + s)));
```

Attenzione alle macro (1)

- Nella definizione **tutti** gli utilizzi dei parametri formali vanno messi **tra parentesi**.

  Altrimenti possono capitare errori come il seguente:

```
# define square(x) x * x

a = square(n)          ⇒  a = n * n;
```

  ma:

```
a = square(n + 1)      ⇒  a = n + 1 * n + 1;
```

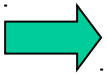
  La versione corretta è

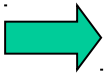
```
# define square(x) ((x) * (x))
```

Attenzione alle macro (2)

- L'intera definizione della macro va **tra parentesi** se non è una singola costante.
  Altrimenti possono capitare errori come i seguenti:

```
# define TOTAL          MAX + 1
# define twice(n)       (n) + (n)
# define max(a, b)      (a) > (b)? (a): (b)

a = 2 * TOTAL;               a = 2 * MAX + 1;

a = 3 * twice(x);            a = 3 * (x) + (x);

a = max(x, y) + 3;
                        

a = (x) > (y)? (x): (y) + 3;
```

Attenzione alle macro (3)

- La sostituzione è puramente testuale; eventuali **effetti collaterali** nelle macro possono essere **pericolosissimi**.

  Possono capitare errori come il seguente:

```
# define max(a, b)          ((a) > (b)? (a): (b))

a = max(x++, y);

a = (x++) > (y)? (x++): (y);
```

  Quante volte si incrementa x?

- Ogni macro deve valutare **esattamente una volta** i parametri:

```
max(f(x), n)
```

  Quante volte si incrementa x?

Regole di sostituzione nelle macro

- Una macro non viene più sostituita durante la sua stessa espansione.
- Spazi multipli (inclusi i commenti) vengono rimpiazzati da un solo spazio.
- Spazi in testa e in coda a un espansione vengono eliminati.
- I parametri ai quali non si applicano operatori # e ## vengono macroespansi finché possibile **prima** di inserirli nell'espansione della macro.

Uso corretto delle macro

- Le macro non dovrebbero **modificare variabili**.

- Per esempio:

```
# define conta(n) (counter += (n))
```

  Se `counter` è una variabile locale, cosa succede utilizzando la macro in una funzione diversa da quella originariamente prevista?

Macro con parametri (1)

- Vanno sempre progettate e trattate nella documentazione come funzioni a tutti gli effetti, con una “scorciatoia” implementativa per motivi di prestazioni.
- Devono avere un prototipo e osservare regole rigide sui parametri.
- Dev’essere possibile rimpiazzarle con funzioni **in ogni momento**.

Macro con parametri (2)

- Le macro non devono essere pensate come un'implementazione di funzioni con argomenti di tipo variabile. Se necessario debbono essere definite più macro analoghe.

- Se è stata definita la macro:

```
# define max(x, y) ((x) > (y)? (x):(y))
```

  e la si utilizza con argomenti interi, per calcolare il massimo tra `double` si utilizzerà la macro:

```
# define d_max(x, y) ((x) > (y)?(x):(y))
```

Macro con parametri (3)

- Meglio ancora, le due possono essere così differenziate:

```
# define     int_max(x, y)        \
   ((int)(x) > (int)(y)?(int)(x): (int)(y))

# define     double_max(x, y)     \
   ((double)(x) > (double)(y)?    \
   (double)(x): (double)(y)
```

Ottenendo la garanzia della conversione dei parametri.

Macro con parametri (4)

- Ogni macro deve valutare **esattamente una volta** i parametri; la popolare macro:
  `# define max(x, y) ((x) > (y)? (x): (y))`
  può dare effetti sgradevoli:

  `max(i++, 9)`
  Quante volte viene incrementata la variabile `i`?

  `max(f(x), n)`
  Quante volte viene chiamata la funzione `f`?

Macro e funzioni

- Nella prima versione del C le macro con parametri erano viste come un modo efficiente per implementare piccole funzioni.
- Con l'introduzione delle funzioni `inline` **non hanno praticamente più ragione di esistere**.
  - Non vi è ragione di rinunciare ai **controlli** sugli argomenti e alla sicurezza delle funzioni.
- Al massimo possono esserci alcuni casi particolari, non sostituibili con funzioni, come:

```
# define elements(array) \
  (sizeof(array) / sizeof(array [0]))
```

Parametri mancanti

- Nell’uso di una macro è permesso omettere qualche parametro (incluso il primo e l’ultimo).
  - Nella macroespansione saranno trattati come stringhe vuote.
- Trattasi però di prassi da **evitare assolutamente**.

Esempio di parametri mancanti

```
# define sum(a, b, c) \
    ((a + 0) + (b + 0) + (c + 0))

sum(x, y,)
sum(1,,x)
```

equivalgono a:

```
((x + 0) + (y + 0) + (+ 0))   ⇒ x + y
((1 + 0) + (+ 0) + (x + 0))   ⇒ 1 + x
```

Macro con parametri variabili

- La lista dei parametri formali di una macro può chiudersi col simbolo `...`, al posto del quale sarà ammesso un qualsiasi numero di parametri.
- Nell'utilizzo, l'identificatore predefinito `__VA_ARGS__` sarà rimpiazzato dai parametri variabili.

Esempio di macro con parametri variabili

```
# define  debug_print(format, ...) \
  if (DEBUGGING)                   \
     (void)fprintf(stderr, format, \
          __VA_ARGS__);

  debug_print("%d %d", x, y);
```

equivale a:

```
  if (DEBUGGING)
     (void)fprintf(stderr, "%d %d", x, y);
```

Compilazioni condizionali

Compilazione condizionale (1)

- La sintassi di una compilazione condizionale è:

  `# if` <*espressione costante*>

  <linee di codice>

  { `# elif` <*espressione costante*>

  <linee di codice>

  }

  [`# else`

  <linee di codice>

  ]

  `# endif`

Compilazione condizionale (2)

- Se l’espressione costante è diversa da zero, al compilatore vengono passate le linee corrispondenti.
- In caso contrario vengono esaminate una alla volta le espressioni condizionali seguenti ai vari `# elif` (se presenti), passando al compilatore le linee corrispondenti al primo con espressione non zero.
- Se le espressioni sono tutte uguali a zero, vengono passate le linee corrispondenti alla parte `# else` (se presente).

Compilazione condizionale (3)

- Gli `elif` possono essere presenti in numero arbitrario (anche nessuno).
- L'`else` è facoltativo.
- Le espressioni sono valutate **dal preprocessore**.
  - La sintassi è quella del C.
- E' ammesso il "nesting", vale a dire che le parti condizionate possono contenere altri `# if` … `# endif`.

Esempio di compilazione condizionale

```
# define DEBUG 1
…
# if DEBUG == 1
  (void)printf("valore di i: %d\n", i);
# endif
…
# if DEBUG == 1
  (void)printf("valore di j: %d\n", j);
# endif
```

Aritmetica del preprocessore (1)

- Il preprocessore valuta le espressioni con regole leggermente differenti da quelle del linguaggio.
  - Non ha nozione di tipi.
  - L'insieme di caratteri utilizzato è quello della macchina sulla quale si compila e **può differire** da quello della macchina target.
  - Gli operatori cast, `&` (unario), `*` (unario) e `sizeof` non possono venir utilizzati.
  - Se si utilizzano identificatori non definiti, hanno valore zero e **non si ha segnalazione d'errore**.

Aritmetica del preprocessore (2)

- Le espressioni aritmetiche del preprocessore vengono valutate nei limiti dei tipi `intmax_t` e `uintmax_t`, definiti nel file stdint.h.
  - I due tipi garantiscono **almeno** gli intervalli di rappresentazione minimi di `long long` e `unsigned long long`, ossia a 64 bit, ma **possono avere un intervallo di valori più ampio**.
- Il compilatore, invece, quando calcola il valore di espressioni costanti deve **totalmente emulare** il comportamento che avrebbe la macchina target, se i calcoli venissero svolti durante l'esecuzione.

Esempio di aritmetica del preprocessore

- L'espressione:

```
# if (1 << 16 ) == 0
```

  non esamina se la macchina è a 16 bit, ma è **sempre falsa** per il preprocessore.

Caratteri del preprocessore

- Scrivendo:

```
# if  'z' - 'a' == 25
    f(x)
# endif

if ('z' - 'a' == 25)
    g(x);
```

  Non è detto che la funzione `f` sia chiamata se e solo se è chiamata la funzione `g`.

  La compilazione condizionale fa riferimento al set di caratteri della macchina **sulla quale si compila**, l'istruzione `if` a quello **della macchina target**.

Esame di definizione (1)

- Vi sono altre due direttive per la compilazione condizionale, con la sintassi:

  `# ifdef` <*identificatore*> o `# ifndef` <*identificatore*>

  <linee di codice>

  `# endif`

  [`# else`

  <linee di codice>

  ]

  `# endif`

- Tali direttive **non danno vantaggi** rispetto ai corrispondenti operatori e **se ne sconsiglia l'uso**.

Esame di definizione (2)

- Se l'identificatore è stato definito, tramite `# define`, prima della direttiva, sono rese visibili al compilatore le linee di codice corrispondenti a `# ifdef`, altrimenti sono rese visibili quelle seguenti alla direttiva `# else` (se presente).
  - L'inverso avviene per `# ifndef`.
  - La parte `# else` è facoltativa.

Operatori del preprocessore

- Lo Standard ha introdotto quattro operatori, riconosciuti dal preprocessore:

  `defined`,

  `_Pragma`,

  `#`,

  `##`.

- Tranne `_Pragma`, sono validi solo entro direttive per il preprocessore o definizioni di macro.
  - Per il compilatore **non sono parole chiave**.

Operatore defined

- Operatore unario, il cui operando è obbligatoriamente un **identificatore**, tra parentesi facoltative.
- Produce valore 1 se l'operando è definito per il preprocessore, 0 altrimenti.
- Valido solo nelle direttive per il preprocessore.
- Permette di sostituire le direttive `# ifdef` e `# ifndef`, aggiungendo flessibilità.

Uso dell'operatore defined

```
  # if defined(XYZ)
```

o

```
  # if defined XYZ
```

rimpiazza il vecchio:

```
  # ifdef XYZ
```

aggiungendo maggiore flessibilità alle condizioni. P. es.:

```
  # if defined(XYZ) || \
      !defined(ABC) || C1 > C2
```

Stato di definizione

- Un identificatore è definito anche se il suo valore è nullo o è una stringa vuota.
  - P. es.:

```
# define TEST 0
```

il valore è zero

```
# define DEBUG
```

il valore è una stringa vuota

```
# if defined(TEST)
```

vero

```
# if defined(DEBUG)
```

vero

Operatore _Pragma (1)

- Operatore unario, con argomento obbligatoriamente stringa e tra parentesi.
- Equivale a una direttiva `# pragma`, avente per contenuto la stringa.
- In pratica:

  `_Pragma("<`*testo*`>")`

  equivale a

  `# pragma` <*testo*>

Operatore

- Operatore unario, il cui operando è obbligatoriamente un identificatore.
- Sostituisce l'operando con la stringa corrispondente al suo valore, tra doppi apici.
  - Caratteri " e \ eventualmente presenti vengono prefissati da \.
  - Spazi e caratteri equivalenti multipli vengono ridotti a un solo spazio;
  - Spazi e caratteri equivalenti in testa e in coda vengono rimossi.
- Valido solo nella definizione di macro.

Uso dell'operatore

- Con l'operatore # si possono creare stringhe parametriche (unico modo, perché il preprocessore non effettua sostituzioni tra apici singoli e doppi).

  P. es., definendo:

```
# define string(x) #x

printf(string(hello));
```

  equivale a:

```
printf("hello");
```

Operatore

- Operatore binario
- Elimina ogni separatore tra i suoi operandi nella macroespansione.
- Valido solo nella definizione di macro.
- Attenzione: l 'operatore ## può creare **identificatori indesiderati** e pertanto va usato con molta cautela.

Uso dell'operatore

- Definendo:

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

glue(pri, ntf)(string(glue(hel, lo)))
```

equivale a:

```
printf("hello");
```

Usi tipici del preprocessore (1)

- È consigliabile proteggere i file inclusi contro l'inclusione multipla.
  - P. es., nel file xyz.h si può scrivere:

```
# if !defined(XYZ_H)
# defineXYZ_H
```

Contenuto del file: macrodefinizioni e dichiarazioni

```
# endif
```

- Tutti i file di inclusione della libreria standard sono protetti in modo analogo.

Usi tipici del preprocessore (2)

- È preferibile centralizzare in un unico file incluso le associazioni tra nomi di variabile e tipi corrispondenti e definire tutte le variabili globali non inizializzate in un unico file.
  - P. es.: nel file xyz.h si scrive:

```
# if !defined GLOBAL
# defineGLOBAL extern
# endif
GLOBAL  int  x;
GLOBAL  long l;
…
```

Usi tipico del preprocessore (3)

Un unico file vars.c definisce tutte le variabili:

```
# define  GLOBAL
# include "xyz.h"
```

In questo file le macroespansioni **definiscono** le variabili:

```
int   x;
long  l;
```

Usi tipico del preprocessore (4)

Gli altri file includono normalmente xyz.h

```
# include    "xyz.h"
```

ed espandono **dichiarazioni**:

```
extern  int  x;
extern  long l;
```

Questo metodo offre due vantaggi:

- per cambiare il tipo o il nome di una variabile basta intervenire in un solo punto;
- la coerenza è garantita.

Non è applicabile alle variabili globali inizializzate.

Usi tipici del preprocessore (5)

- Si possono eliminare in un solo colpo le dichiarazioni `inline`, per facilitare il debugging.
  - Per prima cosa si scrive, in un file incluso da tutti:

```
# if defined USE_INLINE
# define     INLINE    inline
# else
# define     INLINE
# endif
```

Usi tipici del preprocessore (6)

– Poi si spostano tutte le definiscono di funzioni `inline` non `static` relative al file xyz.c in un file xyz.inl, dichiarandole `INLINE`: es.:

```
INLINE    void     f(int x)
      {
      …
      }
```

Usi tipici del preprocessore (7)

- Per finire si aggiungono alcune righe all'inizio del file sorgente e alla fine dello header file.

file xyz.h:

```
# if defined USE_INLINE
# include    "xyz.inl"
# endif
```

file xyz.c:

```
# if !defined USE_INLINE
# include    "xyz.inl"
# endif
```

Usi tipici del preprocessore (8)

- Implementare la verifica di asserzioni.

  Per esempio:

```
void f(int n)
   {
        // se n < 0 fermo l'esecuzione
   assert(n >= 0);
   a [n] = 5;     // sono certo che n >=0
   }
```

  E' necessario:

  – poter eliminare rapidamente i controlli quando il programma sarà a punto;
  – in caso di errore, sapere quale controllo è fallito.

Usi tipici del preprocessore (9)

- Implementazione (in assert.h).

```
# if defined(NDEBUG)
# define     assert(x)
# else
# define     assert(x) \
         ((x)? 0: error(__LINE__, __FILE__))
# endif
```

- Per eliminare i controlli:

```
# define NDEBUG
```

oppure `cc -DNDEBUG xyz.c`

Usi tipici del preprocessore (10)

La funzione `error` scrive un messaggio su standard error e arresta l'esecuzione.

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

int     error(int line, const char *name)
   {
   (void)fprintf(stderr, "Assertion failed "
        "on line %d of file %s\n", line, name);
   exit(EXIT_FAILURE);
   }
```

Errori e e direttive per il compilatore

La direttiva error

- Serve a provocare l'arresto in errore della compilazione.
  - Il testo che segue la parola `error` viene incluso nel messaggio d'errore generato.
- Viene generalmente usata per indicare che alcune definizioni sono scorrette.

Esempio di direttiva error

Supponendo d'aver opportunamente definito le costanti `WINDOWS_10` e `LINUX`, si può imporre che la costante `SYSTEM` sia definita al momento della compilazione:

```
# if    SYSTEM == WINDOWS_10
…
# elif  SYSTEM == LINUX
…
# else
# error Bad SYSTEM
# endif
```

La direttiva line

- Serve a modificare il numero di riga e il nome del file
- Utilizzata dal preprocessore per rendere più comprensibile il file generato e dagli strumenti di sviluppo s/w per la gestione dei messaggi d'errore.
- Ha la sintassi:

  `# line <numero>`

  o

  `# line <numero> " <nome file> "`

  Non va utilizzata.

La direttiva vuota

- Costituita dal solo carattere # su una riga
- Viene ignorata.

La direttiva pragma

- Serve a passare al compilatore flag per controllarne il funzionamento.
- E' stata in parte standardizzata: se l'identificatore `STDC` segue la parola `pragma`, sono ammessi solo i seguenti formati:

  `# pragma STDC FP_CONTRACT` <value>

  `# pragma STDC FENV_ACCESS` <value>

  `# pragma STDC CX_LIMITED_RANGE` <value>

  dove <value> può assumere solo i valori `ON`, `OFF` e `DEFAULT`.
- Se `pragma` non è seguito da `STDC`, il formato resta dipendente dall'implementazione.

Direttiva pragma STDC FP_CONTRACT

- Permette o vieta la contrazione di espressioni reali, cioè il calcolo di parte di una espressione con precisione **maggiore** di quella normale.

Direttiva pragma STDC FENV_ACCESS

- Informa il compilatore che una porzione del programma accede ai flag di stato e controllo del processore floating, tramite apposite funzioni standard, o gira con valori non standard di tali flag.
  - Se un programma richiama le suddette funzioni o gira con valori non standard dei flag in parti di codice dove non si è assicurato il permesso di farlo, il comportamento è indefinito.

Direttiva pragma STDC CX_LIMITED_RANGE

- Informa il compilatore che parti di codice utilizzano intervalli limitati dei numeri complessi, tali da consentire l'uso delle formule "semplici" per moltiplicazione, divisione e valore assoluto.
  - Per esempio, se parte reale e immaginaria non superano, in modulo, la metà della radice quadrata del massimo rappresentabile.
- Permette alcune ottimizzazioni al compilatore.