Il preprocessore C

- Il preprocessore è quella parte del compilatore che si occupa di pre-elaborare un sorgente prima della compilazione vera e propria
- In pratica, permette di generare un nuovo sorgente prima che questo venga compilato effettivamente
- Per esempio, il preprocessore sostituisce le macro (costanti simboliche) con il relativo valore ed inserisce una copia del contenuto dei file di intestazione al posto della relativa riga di inclusione
- Il linguaggio C non può fare a meno della presenza di un preprocessore
  - Accenneremo solo alle direttive più importanti

Istruzioni del preprocessore

- L'utilità della presenza di un preprocessore, tra le altre cose, sta nella possibilità di:
  - definire gruppi di istruzioni alternativi a seconda di circostanze determinate (per esempio, la fase di debugging)
  - definire costanti simboliche
  - includere il contenuto di altri file
- Le direttive del preprocessore sono state regolate anch'esse con lo standard ANSI e costituiscono un linguaggio a sé stante. In generale:
  - le direttive iniziano con il simbolo #, preferibilmente nella prima colonna (ma può essere preceduto da spazi)
  - le direttive non utilizzano alcun simbolo di conclusione (non si usa il punto e virgola)
  - una riga non può contenere più di una direttiva

#include

- La direttiva #include permette di includere un file. Generalmente si tratta di un cosiddetto file di intestazione, contenente una serie di definizioni necessarie al file sorgente in cui vengono incorporate
  - #include <stdio.h>
  - #include "stdio.h"
  - Nel primo caso si fa riferimento a un file che dovrebbe trovarsi in una posizione stabilita dalla configurazione del compilatore
  - Nel secondo caso si fa riferimento ad una posizione precisa definita dall'utente, che richiede l'indicazione di un percorso se non si tratta della stessa posizione in cui si trova il sorgente in questione
- Un file incorporato attraverso la direttiva #include, può a sua volta includerne altri, e questa possibilità va considerata per evitare di includere più volte lo stesso file

#define

#define <macro> [<sequenza-di-caratteri>]

- La direttiva #define permette di definire dei nomi, in questo caso si parla di macro (definite anche costanti simboliche o costanti manifeste), che, quando utilizzati nel sorgente, vengono sostituiti automaticamente con la sequenza che appare dopo la definizione.
  - #define SALUTO ciao come stai
  - #define SALUTO "ciao come stai"
- Il secondo caso è diverso dal primo, perché ci sono in più gli apici doppi. Questa volta, la macro SALUTO potrebbe essere utilizzata in un'istruzione come --- printf(SALUTO); --- mentre non sarebbe stato possibile quando la sostituzione era stata definita senza apici
- Per convenzione, i nomi utilizzati per le macro sono espressi solo con lettere maiuscole

#define

- Vantaggi dell'uso di #define: migliore leggibilità dei programmi, portabilità e facilità di modificare i valori delle costanti simboliche
- Le macro sono tipicamente usate per definire le dimensioni di qualcosa, per esempio gli array, o per fissare la corrispondenza reale con valori determinati che dipendono dalla piattaforma (per esempio, EOF)
- #define può continuare sulla riga successiva usando \ subito prima del carattere di interruzione di riga
- Normalmente le righe #define sono inserite all'inizio dei file, ma possono apparire in qualunque punto ed hanno effetto sulle righe che seguono

#define

- La direttiva può essere utilizzata in modo più complesso, facendo anche riferimento ad altre macro già definite
  - #define UNO 1
  - #define DUE UNO+UNO
  - #define TRE DUE+UNO
- In presenza di una situazione come questa, utilizzando la macro TRE, si ottiene prima la sostituzione con DUE+UNO, quindi con UNO+UNO+1, e infine con 1+1+1 (dopo, tocca al compilatore)
- E' sensato anche dichiarare una macro senza assegnargli alcun valore. Ciò può servire per le direttive #ifdef e #ifndef

#define con argomento

#define <macro>(<argomento>) <sequenza-di-caratteri>

- **La direttiva #define può essere usata in modo simile a una funzione, per definire delle sostituzioni che includono in qualche modo un argomento**
  - **#define DOPPIO(a) (a)+(a)**
  - **...**
  - **i = DOPPIO(i);**
  - **...**

    l'istruzione i=DOPPIO(i) si traduce in i=(i)+(i)
- **Si osservi il fatto che, nella definizione, la stringa di sostituzione è stata composta utilizzando le parentesi**
  - **Questo permette di evitare problemi successivamente, nelle precedenze di valutazione delle espressioni**
- **Le macro con parametri vengono spesso usate per sostituire chiamate di funzione con codice in linea che risulta più efficiente**

#define con argomento

#define <macro>(<argomento>) <sequenza-di-caratteri>

- **La direttiva #define può essere usata in modo simile a una funzione, per definire delle sostituzioni che includono in qualche modo un argomento**
  - **#define DOPPIO(a) (a)+(a)**
  - **...**
  - **i = DOPPIO(i);**
  - **...**

    l'istruzione i=DOPPIO(i) si traduce in i=(i)+(i)
- **Si osservi il fatto che, nella definizione, la stringa di sostituzione è stata composta utilizzando le parentesi**
  - **Questo permette di evitare problemi successivamente, nelle precedenze di valutazione delle espressioni**
- **Le macro con parametri vengono spesso usate per sostituire chiamate di funzione con codice in linea che risulta più efficiente**

#if, #else, #elif e #endif

```
#if <espressione>
    <espressione>
#endif
```

- Le direttive #if, #else, #elif e #endif, permettono di delimitare una porzione di codice che debba essere utilizzato o ignorato in relazione a una certa espressione che può essere calcolata facendo uso soltanto delle definizioni precedenti

```
#define DIM_MAX 1000
...
int main() {
  ...
  #if DIM_MAX>100
  printf("Dimensione enorme.");
  ...
  #else
  printf("Dimensione normale.");
  ...
  #endif
  ...
}
```

#if, #else, #elif e #endif

```
#if 0
  ...
#endif
```

- Un modo efficace per evitare la compilazione di parti di codice durante la fase di debugging

- Una alternativa potrebbe essere quella di usare i commenti, ma sarebbe in generale meno funzionale perché un commento, all'interno di altri commenti, causa collisioni

```
/*
... /* questo fa parte del commento */
questo non fa piu' parte del commento
*/
```

#if, #else, #elif e #endif

- L'istruzione #elif serve per costruire una catena di alternative else-if
- Gli operatori di confronto che si possono utilizzare per le espressioni logiche sono i soliti
- Le direttive condizionali possono essere annidate e possono contenere anche altri tipi di direttiva del preprocessore

```
#define NAZIONE ita
...
...
int main() {
#if NAZIONE==ita
char valuta[] = "EURO";
...
#elif NAZIONE==usa
char valuta[] = "USD";
...
#endif
...
}
```

#ifdef e #ifndef

- Le direttive #ifdef e #ifndef servono per definire l'inclusione o l'esclusione di codice in base all'esistenza o meno di una macro
- L'esempio mostra il caso in cui sia dichiarata una macro DEBUG (che non si traduce in alcunché) e in base alla sua esistenza viene incluso il codice che mostra un messaggio particolare

```
#define DEBUG
...
int main() {
...
#ifdef DEBUG
printf( "Punto di controllo n. 1\n");
...
#endif
...
```

#ifdef e #ifndef

- L'esempio mostrato è analogo al precedente, con la differenza che la direttiva #ifndef permette la compilazione delle istruzioni che controlla solo se la macro indicata non è stata dichiarata
- L'uso delle direttive #else e #endif avviene nel modo già visto per la direttiva #if

```
#define OK
...
int main(void) {
#ifndef OK
printf( "Punto di controllo n. 1\n");
...
#endif
...
}
```

#undef

#undef <macro>

- La direttiva #undef permette di eliminare una macro a un certo punto del sorgente

```
#define NAZIONE ita
...
/* In questa posizione, NAZIONE risulta definita */
...
#undef NAZIONE
...
/* In questa posizione, NAZIONE non e' definita */
...
```

Macro predefinite

- Il compilatore ANSI C prevede 5 macro predefinite
  - Il loro scopo è quello di annotare (a fini diagnostici) informazioni legate alla compilazione nel file eseguibile finale

- __FILE__ contiene una stringa che rappresenta il nome del file sorgente;
- __LINE__ contiene un intero che rappresenta il numero di riga corrente;
- __DATE__ contiene una stringa mese/giorno/anno che rappresenta la data corrente;
- __TIME__ contiene una stringa ore:minuti:secondi che rappresenta l'ora corrente;
- __STDC__ contiene un intero diverso da 0 se il compilatore C che si utilizza segue lo standard ANSI C.

Classi di memorizzazione

- Quando si dichiarano delle variabili, senza specificare alcuna classe di memorizzazione (cioè quando lo si fa normalmente come negli esempi visti fino a questo punto), il loro campo d'azione è relativo alla posizione della dichiarazione:
  - le variabili dichiarate all'esterno delle funzioni sono globali, cioè accessibili da parte di tutte le funzioni, a partire dal punto in cui vengono dichiarate
  - le variabili dichiarate all'interno delle funzioni sono locali, cioè accessibili esclusivamente dall'interno della funzione in cui si trovano

- Le stesse regole di visibilità valgono per i blocchi

Classi di memorizzazione

- **Si distinguono quattro tipi di classi di memorizzazione, a ciascuna delle quali corrisponde una parola chiave per la loro dichiarazione:**
  - ***automatica*** **auto**
  - ***registro*** **register**
  - ***statica*** **static**
  - ***esterna*** **extern**
- **auto: vale in modo predefinito e non occorre indicarla quando si dichiarano le variabili (automatiche)**
- **Una variabile dichiarata come appartenente alla classe register viene posta in un registro del microprocessore**

Classi di memorizzazione

- **Si distinguono quattro tipi di classi di memorizzazione, a ciascuna delle quali corrisponde una parola chiave per la loro dichiarazione:**
  - ***automatica*** **auto**
  - ***registro*** **register**
  - ***statica*** **static**
  - ***esterna*** **extern**
- **static genera due situazioni distinte a seconda della posizione in cui viene dichiarata la variabile**
  - **Se è definita al di fuori delle funzioni, risulterà accessibile solo all'interno del file sorgente in cui viene dichiarata**
  - **Se invece è interna a una funzione, la variabile mantiene il suo valore tra una chiamata e l'altra della funzione**

Classi di memorizzazione

- **Si distinguono quattro tipi di classi di memorizzazione, a ciascuna delle quali corrisponde una parola chiave per la loro dichiarazione:**
  - ***automatica*** **auto**
  - ***registro*** **register**
  - ***statica*** **static**
  - ***esterna*** **extern**

- **La dichiarazione extern è usata**
  - **quando da un file sorgente si vuole accedere a variabili globali dichiarate in modo normale in un altro file**
  - **quando in un file si vuole poter accedere a variabili dichiarate in una posizione più avanzata dello stesso**
  - **In questo modo si informa esplicitamente il compilatore e il linker della presenza di queste**

Classi di memorizzazione

```
extern int iTuaVariabile;
...
int miafunzione(...) {
iTuaVariabile = ...
}
...
```

- **Questo caso rappresenta la situazione in cui una variabile dichiarata in un altro file sorgente diventa accessibile alle funzioni del file attuale attraverso la dichiarazione extern**
  - **Perché ciò possa funzionare, occorre che la variabile iTuaVariabile sia stata dichiarata in modo normale, senza la parola chiave static**

Programmi in un singolo modulo

- **Principali inconvenienti:**
  - **se si fanno anche solo delle piccole modifiche, bisogna sempre ricompilare tutto il programma**
  - **le funzioni definite nel programma non possono essere utilizzate facilmente in altri programmi**
- **Il primo inconveniente porta inevitabilmente a tempi di compilazione piuttosto elevati**
- **Il secondo inconveniente non può efficacemente essere risolto semplicemente ricopiando le funzioni perché**
  - **se ad un certo punto si trova un codice più efficiente per la funzione, si dovrebbe rimpiazzare ogni copia della vecchia funzione con la nuova**
  - **ciascuna copia della funzione occupa spazio disco**

Suddivisione in più moduli

- **I programmi C spesso sono articolati in più file sorgenti separati che vengono compilati in modo indipendentemente e infine collegati in un unico eseguibile**
- **Nella produzione di un eseguibile, vengono ricompilati solo i file sorgente che hanno subito modifiche successivamente alla generazione del file oggetto relativo**
- **Per permettere la condivisione del codice bisogna**
  - **rimuovere il codice delle funzioni che si vogliono rendere meglio fruibili**
  - **compilarlo separatamente e, quindi,**
  - **fare il linking (collegamento) del modulo oggetto risultante in qualsiasi programma che lo voglia utilizzare**

File di intestazione (header)

- **Quando un programma è suddiviso in più moduli sorgente:**
  - **se si utilizzano delle macro del preprocessore, queste dovranno essere dichiarate in tutti i sorgenti che ne fanno uso;**
  - **se all'interno di un sorgente si fa riferimento ad una funzione dichiarata altrove, il sorgente deve contenere una dichiarazione del prototipo della funzione (le variabili globali vanno trattate in modo analogo)**
- **Per facilitarne l'inclusione in tutti i sorgenti che lo necessitano, conviene predisporre dei file di intestazione (con suffisso *.h*) con le dichiarazioni delle macro del preprocessore e dei prototipi delle funzioni**

Make
(in ambiente «consolle»…)

- **La ricompilazione di un programma suddiviso in più moduli può essere un'operazione alquanto laboriosa: si potrebbe predisporre uno script per eseguire tutte le operazioni necessarie ma la tradizione impone di usare Make**
- **Make è una utility che aggiorna un file sulla base di regole di dipendenza contenute in un file con un formato speciale detto makefile**
  - **Per default, make si aspetta che le regole di dipendenza si trovino nel file makefile o, preferibilmente, Makefile**
  - **È possibile specificare un nome diverso per il file contenente le regole (il cui nome è consigliabile abbia suffisso .make)**
- **Il makefile contiene la lista di tutte le interdipendenze che esistono tra i file che sono usati per creare l'eseguibile**

Make
(in ambiente integrato)

- **Makefile è molto utilizzato in ambito scientifico, all'interno di sistemi UNIX**
  - **Vale la pena conoscerne le caratteristiche, ma al momento del suo eventuale utilizzo**
- **In un ambiente integrato (come lcc) il controllo delle dipendenze viene effettuato «in automatico»**
  - **Il risultato verrà trascritto in un makefile (ricordate il «generate makefile» delle opzioni di compilazione…) che poi verrà eseguito**
- **Vediamo qualche esempio**