

Il linguaggio C

Il preprocessore

Il preprocessore

- Viene invocato prima della compilazione vera e propria
- Eseguisce delle manipolazioni testuali sul file
 - sostituisce nomi con testo
 - elimina e inserisce testo
- Le linee di codice che iniziano per # sono direttive per il preprocessore
- Il risultato del preprocessore si può vedere con **gcc -E**

C : un semplice programma (2)

DIRETTIVE al preprocessore
Vengono elaborate prima della
compilazione

```
#include <stdio.h>
#define N 3
```

```
int main (void) {
    int i, tmp, max=0;
    printf("Inserisci %d interi positivi\n",N);
    for (i = 0; i < N; i++) {
        scanf("%d", &tmp);
        max = (max > tmp)? max : tmp ;
    }
    printf("Il massimo è %d \n",max);
    return 0;
}
```

Pre-processing: #include

Direttive di inclusione

- Vengono rimpiazzate dal contenuto del file di include specificato
- Ci sono due modi per specificare il file di include
 - **#include <include_file>**
`include_file` viene cercato nelle directory standard, tipicamente
 - `/usr/include`
 - `/usr/local/include`
 - **#include "path_include_file"**
`include_file` viene cercato nel path specificato

Pre-processing: `#define`

Definizione di macro

- Permettono di associare a un nome un testo, anche parametrico
- Ci sono tre modi per specificare una macro
 - `#define nome`
 - `#define nome testo`
 - `#define nome (parametri) testo`
 - rimpiazza ogni occorrenza di nome con il testo specificato
 - è possibile specificare del testo parametrico

Pre-processing: `#define` (2)

Esempi di macro senza parametri

– `#define DEBUG`

- Si definisce un nome (senza valore) per segnalare un evento (serve per la compilazione condizionale vedi poi)

– `#define SIZE 400`

– `#define SOCKETNAME "/tmp/permsock"`

- Associa ad un nome un valore, serve a parametrizzare il codice C rispetto ad una costante
- Il pre-processor rimpiazza ogni occorrenza di `SIZE` e `SOCKNAME` con il testo specificato
- *In C si usano spesso macro per le costanti (interi, stringhe etc)*

Macro con parametri

- È possibile definire macro con parametri!

```
#define nome (a1, ..., an) testo
```

- in questo caso il testo da sostituire può essere specializzato ad ogni occorrenza della macro es.

```
#define PRODOTTO (X, Y) (X) * (Y)
```

...

```
a = PRODOTTO (a+1, b) ;
```

```
c = PRODOTTO (x+y, k) ;
```

...

Macro con parametri (2)

- Dopo la passata del pre-processor ...

```
/* la linea della define non c'è  
più */
```

...

```
a = (a+1) * (b) ;
```

```
c = (x+y) * (k) ;
```

...

- Le parentesi nella definizione sono importanti per avere la precedenza corretta degli operatori!

Macro con parametri (3)

- Una definizione scorretta sarebbe.

```
#define PRODOTTO (X,Y) X*Y
```

```
...
```

```
a = PRODOTTO (a+1 ,b) ;
```

```
c = PRODOTTO (x+y ,k) ;
```

```
...
```

Macro con parametri (4)

– Risultato dopo il pre-processore...

```
/* la linea della define non c'e'  
piu' */
```

...

```
a = a+1*b;
```

```
c = x+y*k;
```

...

Macro e Funzioni

- Attenzione alla differenza fra macro con parametri e funzioni !
 - (1) Le macro sostituiscono testo per testo
 - (2) una macro viene espansa a tempo di compilazione e non genera chiamate in fase di esecuzione
 - (3) il parametro di una macro viene valutato più volte nel suo corpo (una per ogni occorrenza)
 - possibili problemi con parametri che leggono dato da stream es: `scanf()`

Macro e funzioni (2)

- Una definizione della macro quadrato con chiamata scorretta.

```
#define QUAD (X) (X) * (X)
```

```
...
```

```
a = QUAD (scanf ("%d" , &b) ) ;
```

```
...
```

Macro e funzioni (3)

- Una definizione della macro quadrato con chiamata scorretta. Dopo il preprocessore

```
/* la linea della define non c'è e'  
   piu' */
```

...

```
a = (scanf("%d", &b)) *  
     (scanf("%d", &b));
```

```
... /* la scanf() viene eseguita due  
     volte e si perde un intero */
```

Macro e funzioni (4)

- Ancora una differenza fra macro e funzioni
 - (4) Le macro non sono tipate! Funzionano indipendentemente dal tipo degli argomenti

```
#define PRODOTTO(X,Y) ((X) * (Y))  
double a,b;  
int x,y,k,c;  
...  
a = PRODOTTO(a+1,b) ;  
c = PRODOTTO(x+y,k) ;
```

Macro su più righe ...

- Attenzione : il testo da sostituire va da X fino alla fine della riga

```
#define PRODOTTO (X, Y) X*Y
```

...

- Per fare macro su più righe usare backslash(\)

```
#define PRODOTTO (X, Y) \  
X*Y
```

Ancora sulle macro C

- Alcune macro sono predefinite: `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`,
- Es: `fprintf(stderr, "Error at line %d of file %s\n", __LINE__, __FILE__);`
- E' possibile definire macros con argomenti variabili (variadic macros)
 - sono parte del C99 standard
 - Es.:

```
#define dbgprintf(...) \
    mydbgprintf(__FILE__, __LINE__, __VA_ARGS__);
void mydbgprintf(const char *file, int line, const char *format, ...);
```
- Stringification (esempio):

```
#define LIST_NODE(type) \
    struct list_##type { \
        type val; \
        struct list_##type *next; \
    }
```


Compilazione condizionale

- `#if #ifdef #ifndef #endif`:
 - sono direttive al preprocessore che permettono di includere selettivamente alcuni pezzi del codice durante la compilazione
 - `#if... #endif`

`#if espr_cond`



`#endif`

Questa zona viene ricopiata dal pre-processor nel file da compilare solo se `espr_cond` è verificata (cioè se è diversa da 0)

Compilazione condizionale (2)

- `#if #ifdef #ifndef #endif`:
 - es: un modo rapido per commentare un'area che ha già dei commenti `/*...*/`
 - il C non ammette commenti annidati

```
#if 0
```



```
#endif
```

Questa zona non viene mai copiata

Compilazione condizionale (3)

- **#if #ifdef #ifndef #endif**:
 - **#ifdef nome** testa se **nome** è già stato definito con una **#define**

#ifdef nome



#endif

Questa zona viene inclusa solo se *nome* è stato definito con una **#define**

Compilazione condizionale (4)

- **#if #ifdef #ifndef #endif**:
 - **#ifndef nome** testa se **nome** non è già stato definito con una **#define**

#ifndef nome



#endif

Questa zona viene inclusa solo se **nome** non è già stato definito con una **#define**

Compilazione condizionale (5)

- **#if #ifdef #ifndef #endif :**

- quando si usano ?

- Es. escludere condizionalmente il codice di debugging

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

```
    fprintf(stderr, ..)
```

```
#endif
```

- se ho finito il debugging non definisco DEBUG, ad esempio **#undef DEBUG**

- Per il debugging è buona norma usare il più possibile **assert(condizione)** che termina il programma se la condizione è falsa. Tutte le assert sono rimosse dal compilatore se si compila con **-DNDEBUG**

Compilazione condizionale (6)

- Viene usata per proteggersi da inclusioni multiple :

– Struttura del file **stdio.h**

```
#ifndef _STDIO_H  
#define _STDIO_H
```

Tipi e prototipi stdio

```
#endif
```

- La prima volta che viene incluso il file, **_STDIO_H** non e' definita ed il contenuto del file viene copiato, le successive inclusioni non hanno effetto

Compilazione condizionale (7)

- Uso in combinazione di gcc -D
 - Il gcc permette di specificare una define da inserire in ogni file da compilare

```
gcc -Dnome=val file.c
```

inserisce all'inizio di **file.c** (prima del preprocessing)

```
#define nome val
```
 - in questo caso (1) durante il debugging uso **-D**
(2) quando il debugging è finito compilo senza **-D** ed il codice di debug non viene generato