Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Il preprocessore

Catia Prandi - catia.prandi2@unibo.it

Credit: **Pietro Di Lena**

```c
// Russian roulette
#define return if(random()%6==0) exit(1); return
```

Il preprocessore

- Il **preprocessore** è un programma che elabora il contenuto del file sorgente prima della compilazione.
- Essenzialmente opera delle *sostituzioni tipografiche* sul codice prima che questo venga passato al compilatore.
- Il preprocessore non verifica la sintassi C. Può quindi essere utilizzato anche su un qualsiasi file di testo.
- Anche se il lavoro del preprocessore è formalmente distinto dal lavoro del compilatore, il preprocessore è integrato nel compilatore.

Il preprocessore

- Il **preprocessore** è un programma che elabora il contenuto del file sorgente prima della compilazione.
- Essenzialmente opera delle *sostituzioni tipografiche* sul codice prima che questo venga passato al compilatore.
- Il preprocessore non verifica la sintassi C. Può quindi essere utilizzato anche su un qualsiasi file di testo.
- Anche se il lavoro del preprocessore è formalmente distinto dal lavoro del compilatore, il preprocessore è integrato nel compilatore.
- Tutte le righe di codice in un file sorgente che iniziano con # sono **direttive al preprocessore**.
- Le direttive al preprocessore permettono di
  - **includere** altri file all'interno del file sorgente (direttiva `#include`);
  - **ridefinire** il significato di identificatori, tramite sostituzione tipografica (direttiva `#define`);
  - **disabilitare** condizionalmente parti di codice in fase di compilazione (direttive `#if` `#ifdef`).

La direttiva #include

- Il preprocessore sostituisce ogni riga di codice della forma

```
#include <filename>
```

  oppure

```
#include "filename"
```

  con il contenuto del file `filename`.
- Se il nome del file è specificato tra le parentesi angolari <>, il file viene cercato in una o più directory standard, note al compilatore.
- Se il nome del file è specificato tra virgolette " ", il file viene cercato nella directory corrente.

La direttiva #define

- La direttiva `#define` permette di definire delle **macro definizioni**, abbreviate generalmente in **macro**.

```
#define <IDENTIFICATORE> [<Testo>]
```

- Le macro sono utilizzate per effettuare sostituzioni tipografiche nel file sorgente: ogni occorrenza di `IDENTIFICATORE` nel sorgente viene sostituita con `Testo`.
- La nuova stringa di testo `Testo` va dallo spazio dopo il nome della macro fino a fine riga. Può contenere degli spazi.
- Per convenzione, i nomi delle macro sono definiti utilizzando caratteri maiuscoli.

La direttiva #define

- La direttiva `#define` permette di definire delle **macro definizioni**, abbreviate generalmente in **macro**.

  `#define <IDENTIFICATORE> [<Testo>]`

- Le macro sono utilizzate per effettuare sostituzioni tipografiche nel file sorgente: ogni occorrenza di `IDENTIFICATORE` nel sorgente viene sostituita con `Testo`.
- La nuova stringa di testo `Testo` va dallo spazio dopo il nome della macro fino a fine riga. Può contenere degli spazi.
- Per convenzione, i nomi delle macro sono definiti utilizzando caratteri maiuscoli.
- La definizione di un nome può essere annullata con la direttiva `#undef IDENTIFICATORE`.
- Un esempio di utilizzo delle macro è quello di definire nomi simbolici per costanti numeriche

```
1 #define PI 3.14159 // Costante P-greco
2
3 double r    = 1.3;
4 double circ = 2*PI*r;
5 double area = PI*r*r;
```

  Dopo aver definito la macro PI, possiamo modificare la precisione della costante P-greco semplicemente modificando la macro senza dover modificare ogni singola occorrenza della costante nel codice.
- Nota: le costanti limite nei file `limits.h` e `float.h` sono definite tramite macro.

La direttiva #define: macro molto lunghe

- Abbiamo detto che la stringa di testo che verrà sostituita dalla macro va dallo spazio dopo il nome della macro fino a fine riga.
- E' possibile definire macro molto lunghe utilizzando righe differenti in modo da rendere più leggibile il codice.
- E' sufficiente *spezzare* la macro su più righe utilizzando il costrutto \ a fine riga:

```
#define STRINGA "Questa stringa molto probabilmente \
risulta essere troppo lunga per poter essere digitata \
comodamente su una unica riga."
```

- Attenzione a non inserire uno spazio dopo \.

Le macro parametriche

- Le macro possono *accettare* dei parametri. Possono quindi essere utilizzare per realizzare delle **pseudo-funzioni**.

```
1 #define PI 3.14159     // Costante P-greco
2 #define SQUARE(x) x*x // Quadrato di x
3
4 double r    = 1.3;
5 double area = PI*SQUARE(r);
```

il codice sopra viene trasformato dal preprocessore in

```
1 double r    = 1.3;
2 double area = 3.14159*r*r;
```

Le macro parametriche

- Le macro possono *accettare* dei parametri. Possono quindi essere utilizzare per realizzare delle **pseudo-funzioni**.

```
1 #define PI 3.14159     // Costante P-greco
2 #define SQUARE(x) x*x // Quadrato di x
3
4 double r    = 1.3;
5 double area = PI*SQUARE(r);
```

il codice sopra viene trasformato dal preprocessore in

```
1 double r    = 1.3;
2 double area = 3.14159*r*r;
```

- Rimarichiamo che le macro effettuano delle semplici sostituzioni:

```
1 #define SQUARE(x) x*x
2
3 double x = SQUARE("pippo");
```

$\Longrightarrow$

```
1 double x = "pippo"*"pippo";
```

```
1 #define SQUARE(x) x*x
2
3 double x = SQUARE(1 + 3);
```

$\Longrightarrow$

```
1 double x = 1 + 3*1 + 3;
```

Nel secondo esempio è chiaro che, per un corretto uso, è necessario *proteggere* l'espressione `x*x` nella macro con delle parentesi: `#define SQUARE(x) ((x)*(x))`.

Esempi: utilizzo di una macro parametrica

```c
#include <stdio.h>

// Restituisce l'i-esimo bit di x
#define BIT(x,i) ((x) & (1<<(i)) ? 1 : 0)

int main() {
 short int x = 31;
 unsigned int on, off;

 on  = BIT(x,0) +BIT(x,1) +BIT(x,2) +BIT(x,3) +
       BIT(x,4) +BIT(x,5) +BIT(x,6) +BIT(x,7) +
       BIT(x,8) +BIT(x,9) +BIT(x,10)+BIT(x,11)+
       BIT(x,12)+BIT(x,13)+BIT(x,14)+BIT(x,15);

 off = 8*sizeof(short int)-on;

 printf("Number %u has %u bits on and %u bits off.\n",x,on,off);

 return 0;
}
```

- Calcola il numero di bit ad 1 e a 0 in uno `short int`.
- Senza l'utilizzo di una macro il codice sarebbe stato molto meno leggibile.

Alcune macro speciali

- Il preprocessore definisce automaticamente alcune macro *speciali* che possono essere molto utili in fase di *debugging* del codice.

```
1 #include <stdio.h>
2
3 int main() {
4  printf("File: %s Line: %d Func: %s\n",__FILE__,__LINE__,__func__);
5  return 0;
6 }
```

Se il file sorgente si chiama `main.c`, il programma sopra produce il seguente output

```
File:  main.c Line:  4 Func:  main
```

Alcune macro speciali

- Il preprocessore definisce automaticamente alcune macro *speciali* che possono essere molto utili in fase di *debugging* del codice.

```
1 #include <stdio.h>
2
3 int main() {
4  printf("File: %s Line: %d Func: %s\n",__FILE__,__LINE__,__func__);
5  return 0;
6 }
```

Se il file sorgente si chiama `main.c`, il programma sopra produce il seguente output

```
File: main.c Line: 4 Func: main
```

- Il preprocessore mette a disposizione un limitato numero di macro speciali.

Macro name	Descrizione
`__FILE__`	stringa costante contenente il nome del file in cui è usata la macro
`__LINE__`	intero corrispondente alla linea di codice su cui è posta la macro
`__DATE__`	stringa costante contenente la data di compilazione
`__TIME__`	stringa costante contenente l'ora di compilazione
`__func__`	variabile di tipo stringa che contiene il nome della funzione in cui è posta la macro

Le direttive #if, #ifdef, #ifndef

- Le direttive #if, #ifdef e #ifndef permettono di escludere parti di codice dalla compilazione in base al verificarsi o meno di particolari condizioni.
- Ogni direttiva di questo tipo deve essere *chiusa* da una direttiva #endif.

```
1 #if espressione-costante-intera
2   /* considerato solo se
3      l'espressione e' diversa da 0
4   */
5 #endif
```

```
1 #ifdef macro
2   /* considerato solo se macro
3      e' gia' stata definita
4   */
5 #endif
```

- La direttiva #ifndef si comporta come #ifdef, tranne per il fatto che il codice viene attivato solo se la macro non è stata definita.

Le direttive #if, #ifdef, #ifndef

- Le direttive #if, #ifdef e #ifndef permettono di escludere parti di codice dalla compilazione in base al verificarsi o meno di particolari condizioni.
- Ogni direttiva di questo tipo deve essere *chiusa* da una direttiva #endif.

```
1 #if espressione-costante-intera
2   /* considerato solo se
3      l'espressione e' diversa da 0
4   */
5 #endif
```

```
1 #ifdef macro
2   /* considerato solo se macro
3      e' gia' stata definita
4   */
5 #endif
```

- La direttiva #ifndef si comporta come #ifdef, tranne per il fatto che il codice viene attivato solo se la macro non è stata definita.
- Queste direttive sono utili in fase di debugging.

```
1 #include <stdio.h>
2 #define DEBUG
3
4 int main() {
5 #ifdef DEBUG
6   printf("File: %s Line: %d Func: %s\n",__FILE__,__LINE__,__func__);
7 #endif
8   return 0;
9 }
```

Le informazioni sul file, numero di riga e funzione non sono stampate se #define DEBUG è commentata.

La direttiva #else

- La direttiva `#else` ci permette di fornire una *alternativa* nel caso in cui le condizioni per i rami `#if`, `#ifdef`, `#ifndef` non siano verificate.

```
1  #include <stdio.h>
2
3  #define DEFAULT
4
5  int main() {
6    int x, y;
7
8  #ifdef DEFAULT
9    x=y=0;
10 #else
11   printf("Inserisci x:");
12   scanf("%d",&x);
13   printf("Inserisci y:");
14   scanf("%d",&y);
15 #endif
16
17   // Codice del programma
18
19   return 0;
20 }
```

- Se `DEFAULT` non è definito, allora l'eseguibile verrà compilato in modo da richiede all'utente di inserire i valori iniziali per le variabili `x` e `y`. Diversamente, l'eseguibile verrà compilato in modo da inizializzare a `0` le due variabili.