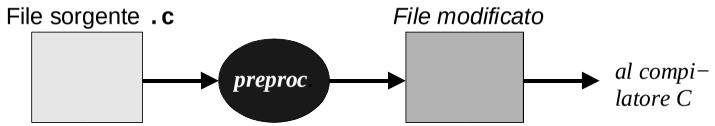
ANCORA SUL PRE−PROCESSORE C

Il pre−processore C *modifica il testo del programma* prima che esso raggiunga il compilatore vero e proprio.



IL PRE−PROCESSORE C

Cosa può fare?

- **includere altre porzioni di testo, prese da altri file**
- **effettuare *ricerche e sostituzioni* (più o meno sofisticate) sul testo**
- ***inserire o sopprimere parti del testo* a seconda del verificarsi di certe condizioni da noi specificate.**

LA DIRETTIVA #define

Sintassi:

`#define` *testo1* *testo2*

Effetto:

definisce una r*egola di ricerca e sostitu−zione:* ogni occorrenza di t*esto1* verrà sostituita da t*esto2*

Scopo:

definire c*ostanti simboliche (*per conven−zione, t*esto1* è m*aiuscolo)*

ESEMPIO

Prima del pre−processing:

```
#define RADICEDI2 1.4142F
main() {
  float lato = 18;
float diagonale = lato * RADICEDI2;
}
```

Dopo il pre−processing:

```
main() {
  float lato = 18;
float diagonale = lato * 1.4142F;
}
```

ESEMPIO

Prima del pre−processing:

```
#define RADICEDI2 1.4142F
main() {
  float lat
float diago
}
```

Regola di ricerca e sostituzione: ogni occorrenza di RADICEDI2 deve essere rimpiazzata da 1.4142F

Dopo il pre−proce

Programma risultante inviato al compilatore

```
main() {
  float lato = 18;
float diagonale = lato * 1.4142F;
}
```

CONTRO−ESEMPIO

Prima del pre−processing:

```
#define RADICEDI2 1.414paperino
main() {
  float lato = 18;
float diag = lato * RADICEDI2;
}
```

Dopo il pre−processing:

```
main() {
  float lato = 18;
float diag = lato * 1.414paperino;
}
```

IL PRE−PROCESSORE C

Attenzione:

- **nell'effettuare ricerche e sostituzioni, il pre−processore *non sa quello che fa***
- **si limita a sostituire testo con altro testo**
- ***non effettua controlli di nessun tipo*, né può farli: non è un compilatore, e dunque *non conosce la sintassi del C!***
- **Quindi, regole sbagliate possono produrre *risultati assurdi!***

CONTRO−ESEMPIO

Prima del pre−processing:

```
#define RADICEDI2 1.414paperino
main() {
  float lato
float diag =
}
```

La regola di ricerca e sostituzione è *perfettamente lecita* in sé….

Dopo il pre−proce

.. ma il programma risultante è *sintatticamente errato*!!

```
main() {
  float lato = 18;
float diag = lato * 1.414paperino;
}
```

LE MACRO

La regola di ricerca e sostituzione introdotta dalla direttiva `#define` si chiama *macro.*

Regole semplici, come le precedenti:

```
#define MAX 10
#define RADICEDIDUE 1.4142F
```

definiscono *macro semplici.*

La direttiva `#define` permette però anche di definire regole più complesse, che vanno sotto il nome di *macro parametriche.*

MACRO PARAMETRICHE

La regola di ricerca e sostituzione può comprendere dei *parametri:*

```
#define DOUBLE(X) 2*X
```

il parametro che segue il nome DOUBLE si usa per determinare il testo da inserire in sostituzione di DOUBLE(X).

- così, la frase DOUBLE(3) sarà sostituita dalla frase 2*3
- analogamente, la frase DOUBLE("a") sarà sostituita dalla frase 2*"a"

MACRO PARAMETRICHE

Attenzione:
il meccanismo di sostituzione del testo può portare, a volte, a *risultati inattesi* e per certi versi *sorprendenti.*

Ad esempio, la macro:
`#define DOUBLE(X) 2*`
usata nel seguente modo:
`b = DOUBLE(a+1)`
produce come testo
`b = 2*a+1`

Nelle intenzioni voleva significare "*il doppio di a+1*"...

... ma l'espressione risultante *ha un altro significato!*

MACRO PARAMETRICHE

Siamo in presenza di una *diversa lettura* della macro

- l'utente intendeva il parametro *come un* "*tutt'uno*"
- *ma la regola di sostituzione non ha rispettato questa* "*visione*", in quanto l'operatore * ha priorità superiore al +

È una macro sbagliata.

MACRO PARAMETRICHE

Soluzione: inserire una coppia di parentesi per assicurare che ciò che è "*concettual−mente un tutt'uno*" rimanga anche *pratica−mente* un tutt'uno.

#define DOUBLE(X) 2*(X)

- **così, la frase DOUBLE(a+1) sarà sostituita dalla frase 2*(a+1) che esprime correttamente quanto desiderato.**

MACRO PARAMETRICHE

È un'altra forma di *diversa lettura* della macro:

- **l'utente intendeva la macro *come un* "*tutt'uno*"**
- ***ma la regola di sostituzione non ha rispettato questa* "*visione*", in quanto l'ha espansa come espressione (X)+(Y), e la priorità degli operatori ha creato il guaio.**

Occorre riscrivere la macro.

MACRO PARAMETRICHE

Consideriamo quest'altra macro:

#define ADD(X,Y) (X) + (Y)

- **I singoli parametri sono fra parentesi, in modo da evitare il problema precedente**
- **Tuttavia, un altro problema è in agguato.**

Se la macro viene usata come:

b = 2 * ADD(3, 4)

i*l testo risultante è*

b = 2*(3)+(4)

che non è affatto "il doppio di 3+4" !!

MACRO PARAMETRICHE

Soluzione: inserire una coppia di parentesi intorno alla macro nel suo complesso per assicurare non già l'atomicità *dei singoli parametri,* quanto l'atomicità *della macro nel suo complesso.*

#define ADD(X,Y) ((X)+(Y))

- **così, la frase 2*ADD (3,4) sarà sostituita dalla frase 2*((3)+(4)) che esprime correttamente quanto desiderato.**

MACRO CON EFFETTI COLLATERALI

Una macro *non dovrebbe mai avere effetti collaterali,* perché le espansioni multiple potrebbero giocare *brutti scherzi.*

Esempio:

```
#define MAX(X,Y)  ( ((x) > (Y)) ? (X) : (Y) )
```

- **nessun problema con frasi come MAX (3,8) o MAX(z,x+1)** …
- **.. ma *molti problemi con MAX (++x,4) !!***

MACRO vs. FUNZIONI

- **una funzione è un'entità computazionale invocata a run−time**, il cui mondo (il record di attivazione) viene *generato a tempo d'esecuzione* a fronte di ogni chiamata
  - il compilatore può effettuare controlli di correttezza!
- **una macro invece è solo una regola di espansione di testo**, applicata dal pre−processore *prima ancora* che il programma venga *compilato.*
  - nessun controllo è possibile

MACRO CON EFFETTI COLLATERALI

Una macro *non dovrebbe mai avere effetti collaterali,* perché le espansioni multiple potrebbero giocare *brutti scherzi.*

Esempio:

```
#define MAX(X,Y)  (
```

- **nessun problema cor**
  o MAX(z,x+1) …
- **.. ma *molti problemi con MAX (++x,4) !!***

Se x vale 5, nelle inten−zioni doveva calcolare max(6,4), cioè 6 ...

Morale: max(6,4) dà 7

… ma in realtà X viene *espanso due volte* e quindi x è *incrementato due volte!*