Capitolo 13 – Il preprocessore

Outline

Obiettivi

- In questo capitolo, impareremo a:
  - utilizzare `#include` per lo sviluppo di grandi programmi.
  - Utilizzare #define per la creazione di macro.
  - Capire la compilazine condizionata.
  - Visualizzare messaggi di errore durante la compilazione.
  - Utilizzare le asserzioni per testare se i valori di una espressione sono corretti.

Introduzione

- Preprocessing
  - Si verifica prima di compilare un programma
  - Inclusione di altri file
  - Definizione di costanti simboliche e macro
  - Compilazione condizionata
  - Esecuzione condizionata delle direttive
- Formato delle direttive
  - Linee che iniziano con #
  - Solo caratteri di spazio prima delle direttive su una linea

La direttiva `#include`

- `#include`
  - Copia uno specifico file dove compare la direttiva
  - `#include <filename>`
    - Inclusione di librerie standard
  - `#include "filename"`
    - Inclusione di file definiti dall'utente
  - Utilizzata per:
    - Programmi con più file sorgenti da compilare assieme
    - File Header – hanno delle definizione e delle dichiarazioni comuni (classi, strutture, prototipi di funzione)
      - `#include` in ogni file

La direttiva #include: costanti simboliche

- #define
  - Direttiva utilizzata per creare costanti simboliche e macro
  - Costanti simboliche
    - Quando un programma viene compilato, tutte le occorrenze della costante simbolica vengono rimpiazzate nel testo
  - Formato

    `#define` *identifier replacement-text*
  - Esempio:

    ```
    #define PI 3.14159
    ```
  - Ciò che è a destra dell'identificatore rimpiazza il testo

    ```
    #define PI = 3.14159
    ```
    - Rimpiazza "`PI`" con "`= 3.14159`"
  - Non si possono ridefinire le costanti simboliche una volta create

La direttiva #include: le macro

- Le macro
  - Operazioni definite in `#define`
  - Una macro senza argomenti viene trattata come una costante simbolica
  - Una macro con argomenti ha i suoi argomenti sostituiti al testo degli argomenti, quando espansa
  - Effettua la sostituzione del testo – nessun data type checking
  - La macro

    ```
    #define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
    ```

    causa

    ```
    area = CIRCLE_AREA( 4 );
    ```

    per diventare

    ```
    area = ( 3.14159 * ( 4 ) * ( 4 ) );
    ```


La direttiva #include: le macro

- Uso di parentesi
  - Senza parentesi la marco

    ```
    #define CIRCLE_AREA( x )  PI * ( x ) * ( x )
    ```

    causerebbe

    ```
    area = CIRCLE_AREA( c + 2 );
    ```

    Per diventare

    ```
    area = 3.14159 * c + 2 * c + 2;
    ```
- Argomenti multipli

  ```
  #define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
  ```

  causerebbe

  ```
  rectArea = RECTANGLE_AREA( a + 4, b + 7 );
  ```

  Per diventare

  ```
  rectArea = ( ( a + 4 ) * ( b + 7 ) );
  ```


La direttiva #include: le macro

- #undef
  - Elimina la definizione di una costante simbolica o di una macro
  - Se una costante simbolica o una macro non era definita può essere ridefinita

Compilazione condizionata

- Compilazione condizionata
  - Controllano le direttive e la compilazione
  - Espressioni di cast, `sizeof`, enumerazioni di costanti non possono essere valutate nelle direttive
  - Struttura simile alla `if`

```
#if !defined( NULL )
   #define NULL 0
#endif
```

    - Determina se la costante simbolica `NULL` è stata definita
      - se `NULL` è definita, `defined( NULL )` vale `1`
      - se `NULL` non è definita, questa funzione definisce `NULL` a `0`
  - Ogni `#if` deve terminare con `#endif`
  - `#ifdef` è una abbrevazione per `#if defined( name )`
  - `#ifndef` è una abbrevazione per `#if !defined( name )`

Compilazione condizionata

- Altre istruzioni
  - `#elif` – è equivalente a `else if` in una istruzione `if`
  - `#else` – è equivalente a `else` in una istruzione `if`
- Codice "Comment out"
  - Non si può utilizzare `/*` ... `*/`
  - Utilizzare

```
#if 0
   code commented out
#endif
```

  - Per attivare il codice, cambiare `0` con `1`

Compilazione condizionata

- Debugging

```
#define DEBUG  1
#ifdef DEBUG
     cerr << "Variable x = " << x << endl;
  #endif
```

  - Definendo `DEBUG` a `1` attiva il codice
  - Dopo aver corretto il codice, rimuovere l'istruzione `#define`
  - Le istruzioni di debugging ora vengono ignorate

Le direttive `#error` e `#pragma`

- `#error` tokens
  - I tokens sono sequenze di caratteri separate da spazi
    - `"I like C++"` ha `3` token
  - Visualizza un messaggio che include i token specificati come messaggio d'errore
  - Si ferma e previene la compilazione del programma
- `#pragma` tokens
  - provoca un'azione definita dall'implementazione (consult compiler documentation)
  - una direttiva pragmas non riconosciuta dal compilatore sarà ignorata

Gli operatori # e

- #
  - Causa un rimpiazzamento di testo convertito in stringa fra virgolette
  - L'istruzione

```
#define HELLO( x ) printf( "Hello, " #x  "\n" );
```

causa

```
HELLO( John )
```

Per diventare

```
printf( "Hello, " "John" "\n" );
```

  - Le stringhe separate da spazi bianchi vengono concatenate quando si usa la `printf`

Gli operatori # e

- ##
  - Concatena due token
  - L'istruzione

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

causa

```
TOKENCONCAT( O, K )
```

Per diventare

```
OK
```


Numeri di linea

- `#line`
  - Rinumera le successive linee di codice, iniziando col valore intero
  - Possono essere inclusi i nomi di file
  - `#line 100 "myFile.c"`
    - Le linee sono numerate da 100 iniziando dal successivo file sorgente
    - I messaggi del compilatore faranno riferimento a `"myfile.c"`
    - Rende gli errori più significativi
    - I numeri di linea non appaiono nel file sorgente

Costanti simboliche predefinite

- Quattro costanti simboliche predefinite
  - Non si possono usare in `#define` o `#undef`

Symbolic constant	Description
`__LINE__`	The line number of the current source code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file is compiled (a string of the form **"Mmm dd yyyy"** such as **"Jan 19 2001"**).
`__TIME__`	The time the source file is compiled (a string literal of the form **"hh:mm:ss"**).

Asserzioni

- `assert` macro
  - Header `<assert.h>`
  - Valore di test di una espressione
  - Se `0` (`false`) stampa un messaggio di erroree invoca `abort`
  - Esempio:

```
assert( x <= 10 );
```

  - se `NDEBUG` è definita
    - Tutte le successive istruzioni assert vengono ignorate

```
#define NDEBUG
```