Compilazione Condizionale

Esistono direttive del preprocessore che consentono la compilazione condizionata, vale a dire la complazione di parte del codice sorgente solo sotto certe condizioni.

L'uso e' molteplice:

- Gestione degli errori
- Portabilità del codice
- Inclusione di codice scritto da altri
- Debug del codice sorgente

Comandi per la Compilazione Condizionale

#if include nella compilazione un qualche testo in dipendenza del valore di un'espressione costante

#ifdef include un qualche testo nella compilazione se un nome di macro é definito

#ifndef include un qualche testo nella compilazione se un nome di macro non é definito

#else include un qualche testo, se i precedenti test in #if , #ifdef , #ifndef o #elif sono falliti

#elif permette la scrittura di if - else annidati, l'ultimo deve essere #else

#endif conclude l'espressione condizionale

Comandi per la Compilazione Condizionale

SINTASSI

#if espressione costante
gruppo di linee di codice 1
#else
gruppo di linee di codice 2

SEMANTICA

Se *espressione costante* ≠ 0 verrà compilato il *gruppo di linee di codice 1* e l'altro viene scaricato altrimenti verrà compilato il il *gruppo di linee di codice 2* e l'altro scaricato

Comandi per la Compilazione Condizionale

espressione costante: viene valutata a tempo di compilazione

espressione costante: di tipo intero, coinvolgendo solo costanti intere, costanti di tipo char. Tutta l'aritmetica è fatta usando il tipo long o unsigned long. Non ammette l'uso dell'operatore **sizeof** o un cast

Un *gruppo di linee di codice* può contenere un qualsiasi numero di linee di testo, anche altre direttive di preprocessore o anche nessuna linea

Comandi per la Compilazione Condizionale

SINTASSI

#if espr. cost.1
linee di codice 1
#elif espr. Cost.2
linee di codice 2
...
#elif espr.cost.n
linee di codice n
#else
linee di codice

SEMANTICA

Se *espr cost1* ≠ 0 compila il *linee di codice 1* e tutto il resto viene scaricato altrimenti verrà valutata l' *espr.cost2* del successivo #elif e se diversa da 0 vengono compilate le *linee di codice 2* e il resto scaricato e così via. Se nessuna delle *espr. cost.i* dà un valore diverso da 0, allora verrà compilato il *gruppo inee di codice* e verrà ignorato il resto

Comandi per la Compilazione Condizionale

SINTASSI

#ifdef identific.
Linee di codice 1

SEMANTICA

Compila le *linee di codice 1* solo nel caso in cui *identific* è stato precedentemente definito mediante una direttiva #define senza che sia stato eliminato da una direttiva #undef

#ifndef identific.
Linee di codice

Compila il codice *linee di codice* solo nel caso in cui l'identific non e' stato definito

Uso ed Esempi

Usare printf per il debug

```
#define DEBUG 1
#if DEBUG
     printf("debug: a = %d\n",a);
#endif
```

Visto che DEBUG è diverso da 0 le linea del printf viene compilata ed aiuta nel debug del programma. Successivamente ommettendo `#define DEBUG 1` le linee non verrano compilate (e quindi non verranno eseguite).

Uso ed Esempi

Costruzione alternativa

```
#define DEBUG
#ifdef DEBUG
     printf("debug: a = %d\n",a);
#endif
```

Visto che DEBUG è definito, la linea del printf viene compilata ed aiuta nel debug del programma. Successivamente, ommettendo `#define DEBUG` le linee non verrano compilate e quindi eseguite.

Uso ed Esempi

Prevenire sovrapposizioni di MACRO: #undef

```
#include "progetto_esterno.h"
#undef MACRO
#define MACRO
```

Non conoscendo totalmente progetto_esterno.h, prima di definere una propria macro, conviene usare undef per prevenire una doppia definizione

Uso ed Esempi

Test durante lo sviluppo di un programma

```
Blocco istruzioni 1
Blocco istruzioni 2
Blocco istruzioni 3

/*
 Blocco istruzioni 2
*/
```

Problemi: Il codice eliminato contiene a sua volta dei commenti

Durante un test potrebbe essere desiderabile eliminare temporaneamente alcune parti di codice.

Alternativa

```
Blocco istruzioni 1
#if 0
Blocco istruzioni 2
#endif
Blocco istruzioni 3
```

Comando **defined**

Nello standard ANSI C è disponibile l'operatore **defined**. L'espressione

`defined identificatore` è equivalente a

`defined(identificatore)` e

vale 1 se l'identificatore è stato definito.

Nota. Può essere usata in una espressione di #if.

```
#define HP9000
#if defined(HP9000)
...
#endif
```

Un esempio

Scriviamo un programma che ordina un vettore.

Lo testiamo usando le direttive del preprocessore

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 6000
#define DEBUG

void insSort(int * a,  int n);
/* ordina in ordine crescente un vettore di interi
* Precondizione :  (a != NULL && n > 0)
* Postcondizione : a[i] <= a[i+1], per 0<=i<n.*/

void stVett(int *, int);
/* visualizza sul monitor gli elementi del vettore */
```

Un esempio

```
main()
{ int app, num, i, vett[SIZE];
printf("Inserisci il numero di elementi del vettore (<= %d).\n",SIZE);
scanf("%d",&num);
srand(app);
#if  defined(DEBUG)
     printf("Inizializziamo un vettore\n");
#endif
for (i = 0;i < num;i++) vett[i] = rand() ;
#ifdef DEBUG
     printf("\nGli elementi scelti a caso e inseriti  nel vettore sono:\n");
     stVett(vett,num);
#endif
insSort(vett,num);
#ifdef DEBUG
     printf("\nGli elementi ordinati  nel vettore sono:\n");
    stVett(vett,num);
#else
    printf("Abbiamo finito il debugging di insSort\n");
#endif
return 0;}
```

Un esempio: Output

Fase di Debugging: DEBUG attivo

Inserisci il numero di elementi del vettore (<= 6000).

15

Inizializziamo un vettore

Gli elementi scelti a caso e inseriti nel vettore sono:

16838 5758 10113 17515 31051 5627 23010 7419 16212 4086
 2749 12767 9084 12060 32225

Gli elementi ordinati nel vettore sono:

 2749 4086 5627 5758 7419 9084 10113 12060 12767 16212
16838 17515 23010 31051 32225

Un esempio: Output

Fine Fase di Debugging: #undef DEBUG

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 6000
#define DEBUG
#undef DEBUG

void insSort(int * a,  int n);
/* ordina in ordine crescente un vettore di interi
* Precondizione :  (a != NULL && n > 0)
* Postcondizione : a[i] <= a[i+1], per 0<=i<n.*/

void stVett(int *, int);
/* visualizza sul monitor gli elementi del vettore */
```

Un esempio: Output

Inserisci il numero di elementi del
vettore (<= 6000).

10

Abbiamo finito il debugging di insSort

Evitare possibili errori di sintassi

Quando si usa la compilazione condizionata, il codice escluso non solo non genera codice oggetto ma non è nemmeno verificato sintatticamente, questo può nascondere degli errori che solo in un secondo momento sarebbero individuati.

Alternativa alla compilazione condizionale

Usando normali istruzioni condizionali si ottiene il risultato che queste verranno verificate anche se per esse molti compilatori non genereranno codice.

rimpiazzato da

```
#define DEBUG
#ifdef  DEBUG
   ...
#endif
#if defined(DEBUG)
  ...
#endif
#undef DEBUG
```

```
enum { DEBUG = 1}
if (DEBUG) {...}
enum { DEBUG = 0}
```

Compilazione di più files

Introduzione all´uso di make

Un programma può esere formato da piu file .h e .c contenuti in una directory di lavoro

Il comando make legge per default il file makefile che ci permette di specificare e descrivere le dipendenze tra i vari moduli e file di cui è composto il programma.

In particolare contiene le istruzioni per compilare e ricompilare il programma

Compilazione di più files

Esempio:
main.c e sum.c includono entrambi l'header sum.h

si vuole che il programma eseguibile sia scritto sul file sum

Contenuto di Makefile

```
sum: main.o sum.o
     cc –o sum main.o sum.o
main.o:  main.c sum.h
     cc –c main.c
sum.c sum.c sum.h
     cc-c sum.c
```

Inizia con una tabulazione

dipendenze

Riga di comando

Sintassi di un makefile

Un makefile è costituito da una sequenza di **REGOLE** che Specificano **DIPENDENZE** ed **AZIONI**.

Una regola iniza con una sequenza di nomi di file, detti **file target**, separati da spazi e seguiti da un ":", a loro volta seguiti da una sequenza di file che rappresentano i **PREREQUISITI** e detti file sorgente

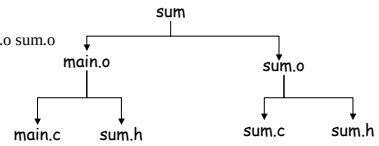
```
sum: main.o sum.o
```

Tutte le righe successive che iniziano con una tabulazione rappresentano azioni

```
sum: main.o sum
        cc –o sum main.o sum.o
```

Albero dipendenze e regole predefinite

```
sum: main.o sum.o
        cc –o sum main.o sum.o
main.o:  main.c sum.h
        cc –c main.c
sum.o: sum.c sum.h
        cc-c sum.c
```



Regola predefinita: file .o dal corrispondente file .c

Versione semplificata

```
sum: main.o sum.o
        cc –o sum main.o sum.o
main.o: sum.h
        cc –c main.c
sum.o: sum.h
        cc-c sum.c
```

Regole predefinite e semplificazioni

```
sum: main.o sum.o
        cc –o sum main.o sum.o
main.o: sum.h
        cc –c main.c
sum.o: sum.h
        cc-c sum.c
```

Nell'esempio precedente main.o e sum.o dipendono entrambi da sum.h. Possiamo semplificare il precedente makefile nella forma:

```
sum: main.o sum.o
        cc –o sum main.o sum.o
main.o  sum.o: sum.h
        cc –c $*.c
```

$*.c viene espanso in main.c nel caso in cui debba essere creato main.o e in sum.c nel caso debba essere creato sum.o

Makefile: altre funzionalità

- Permette l'introduzione di commenti

```
# Questo è un commento
```

- Permette la definizione di MACRO

```
DIRLAVOR = /home/Pippo/
```

- Permette di richiamare una macro

```
$(DIRLAVORO)
```

- Permette di evitare di scrivere righe di comando sullo schermo

```
@echo "linking..."
@cc –c
```

Un esempio

```
#Esempio di file Makefile per ordinamento
BASE    =       /home/blufox/base
CC      =       gcc
CFLAGS=         -O - Wall
EFILE   =       $(BASE)/bin/compare_sorts
INCLS   =       -I $(LOC)/include
LIBS    =       $(LOC)/lib/g_lib.a \
                $(LOC)/lib/u_lib.a
LOC     =       /usr/local
OBJS    =       main.o      another_qsort.o   chk_order.o\
                compare.o   quicksort.o

$(EFILE):       $(OBJS)
        @echo "linking ..."
        @$(CC)  $(CFLAGS) –o $(EFILE)  $(OBJS)  $(LIBS)

$(OBJS):  compare_sorts.h
        $(CC) $(CFLAGS) $(INCLS) –c $*.c
```