21 Preprocessore (I)

Preprocessore: parte di compilatore che elabora il testo del programma prima dell'analisi lessicale e sintattica

- **Può includere nel testo altri file**
- **Espande i simboli definiti dall'utente secondo le loro definizioni**
- **Include o esclude parti di codice dal testo che verrà compilato**

Queste operazioni sono controllate dalle *direttive per il preprocessore* (il primo carattere è #)

Inclusione di file header. Due possibilità:

- **Percorso a partire da cartelle standard**
  #include <file.h>
- **Percorso a partire dalla cartella in cui avviene la compilazione o percorso assoluto**
  #include "file.h"

Esempio:

```
#include "subdir\file.h"
#include "c:\subdir\file.h"
```


21 Preprocessore (II)

Macro

Simbolo che viene sostituito con una sequenza di elementi lessicali corrispondenti alla sua definizione

```
#define CONST 123
#define LINUX
#define MAX(A,B) ((A)>(B) ? (A) : (B))

int main()
{
   int z = 10;
   X = CONST;              // X = 123
   Y = MAX(4, (z+2))       // Y = ((4) > (z+2) ? (4) : (z+2))
}
```

ATTENZIONE alle parentesi!!!!

```
#define MAX(A,B) (A)>(B) ? (A) : (B)

int main()
{
   Y = 1 + MAX(1, 3)       // Y = 1 + (1) > (3) ? (1) : (3)
}
```

Diventerebbe

```
int main()
{
   Y = 1 + (1) > (3) ? (1) : (3)  // 3 invece di 4, per via della
                                  // maggiore priorita' dell'op. +
                                  // rispetto all'op. >
}
```

21 Compilazione condizionale (I)

Compilazione condizionale #if, #elif, #else, #endif

```
#if    constant-expression
       (code if-part)
#elif  constant-expression
       (code elif-parts)
#elif  constant-expression
       (code elif-parts)
#else
       (code else-part)
#endif
```

I valori delle espressioni costanti vengono interpretati come valori logici ed in base a essi vengono compilati o no i frammenti testo (text) seguenti.

Esempio:

```
#define DEBUG_LEVEL 1 // all'inizio del file

int main(){
#if DEBUG_LEVEL == 2
    cout<<i<<j;  // debug di i e j
#elif  DEBUG_LEVEL == 1
    cout<<i;      // debug della sola variabile i
#else
   // DEBUG_LEVEL == 0
   (nessuna cout)
#endif
  return 0;
}
```


21 Compilazione condizionale (II)

FORME CONCISE

#ifdef identifier	per	**#if defined identifier**
#ifndef identifier	per	**#if !defined identifier**

ESEMPIO

```
// file main.cpp

#define LINUX            // commentare questa riga e
// #define WINDOWS      //  scommentare questa
                        // per compilare sotto Windows

#include <cstdlib>
#include <iostream>
using namespace std;

int main()
  #ifdef LINUX     // equivale a '#if defined LINUX'
     system(" CLEAR");
  #elif defined WINDOWS
     system("CLS");
  #else
     cout<<"Sistema operativo non supportato"<<endl;
     exit(1);
  #endif
     return 0;
}
```

DEFINE A RIGA DI COMANDO

Alternativamente, gli identificatori per il processore si possono definire invece che in main.cpp direttamente alla chiamata del compilatore:

```
// Per compilare sotto LINUX
g++ -DLINUX main.cpp –o main.exe
// Per compilare sotto WINDOWS
g++ -DWINDOWS main.cpp –o main.exe
```

21 Compilazione condizionale (III)

Altro utilizzo della compilazione condizionale:

evitare che uno stesso file venga incluso più volte in una unità di compilazione. Ogni file di intestazione, per esempio *complesso.h*, dovrebbe iniziare con le seguenti direttive

---

```
// file complesso.h
#ifndef COMPLESSO_H
#define COMPLESSO_H
  // qui va il contenuto vero e proprio del file
  class complesso{
     double re, im;
  public:
    // funzioni della classe complesso
  };
#endif
```

```
// file main.cpp
#include "complesso.h"
#include <iostream>
#include "complesso.h"

int main(){
    complesso c;
    return 0;
}
```

l'aggiunta erronea di questo secondo include ora non causa più il problema di ridefinizione della classe *complesso*, perché la definizione della classe *complesso* viene inclusa solo la prima volta

---