Laboratorio di programmazione

Lezione XI

Tatiana Zolo

zolo@cs.unipr.it

MAKEFILE: COS'È?

Molti file sorgenti $\Longrightarrow$ compilazione snervante!

Conviene individuare i file che sono stati modificati dall'ultima compilazione e ricompilare solo quest'ultimi.

Il Makefile o makefile serve proprio ad agevolare la compilazione del programma. Si tratta di un file che può contenere:

- ➜ il compilatore da usare;
- ➜ le istruzioni su come utilizzare il compilatore;
- ➜ quali file compilare;
- ➜ come creare il file eseguibile;
- ➜ la directory dove trovare le librerie necessarie;
- ➜ . . .

Una volta creato il Makefile appropriato basterà digitare il comando make per ricreare l'eseguibile.

MAKEFILE: CENNI

Sintassi essenziale del makefile

```
#commento
target ... : dependencies ...
        command
        ...
        ...
```

- ➜ `target`: tipicamente è il nome dell'eseguibile o del file oggetto da ricompilare, ma può essere anche un'azione (es. "clean").
- ➜ `dependencies`: usate come input per generare l'azione `target` (in genere più di una). Vengono citati i file e le azioni da cui dipende la compilazione di `target`.
- ➜ `command`: il comando da eseguire. Può essere più di uno e di solito si applica alle `dependencies`.

es. vedi la struttura del programma "bilancia_parentesi" ed il suo Makefile.

COME USARE MAKE

Il comando make prende in pasto un makefile e lo esegue:

- ➜ ricompila tutti i file sorgenti che sono stati modificati dall'ultima compilazione;
- ➜ se sono cambiati gli header file verranno ricompilati anche tutti i file che li includevano;
- ➜ se qualche file è stato modificato tutti i file oggetto, nuovi o vecchi, verranno linkati assieme per produrre l'eseguibile.

Uso (da riga di comando):

`make`

`make all` (`all` è l'etichetta di default).

`make etichetta`: si ricompila solo il sottoinsieme dipendente da etichetta.

MAKEFILE: CONCLUDENDO...

- Scrivi programmi di una certa complessità e dimensione? L'utilità di make è evidente!!
- E le potenzialità di questa utility? Attraverso make si può rendere automatico non solo il lavoro di ricompilazione, ma qualsiasi altro processo che richieda di essere aggiornato (provate a guardare nei makefile che accompagnano i programmi freeware...).

Genericità in C++: TEMPLATE

- Problema della genericità:
  - ➜ scrivere codice generico
  - ➜ realizzare strutture dati generiche

  validi *indipendentemente* dal tipo degli oggetti coinvolti.

GENERICITÀ IN C++: TEMPLATE

- Problema della genericità:
  - ➜ scrivere codice generico
  - ➜ realizzare strutture dati generiche

  validi *indipendentemente* dal tipo degli oggetti coinvolti.

- Il C++ mette a disposizione i TEMPLATE: permettono di definire funzioni e classi in termini di un tipo indicato come parametro. Il tipo esatto da usare verrà poi precisato al momento dell'uso della classe o della funzione.

TEMPLATE DI FUNZIONE

- C++: linguaggio fortemente tipizzato $\Longrightarrow$ ostacolo all'implementazione di funzioni altrimenti immediate.

```
int min(int a, int b) {
  return a < b ? a : b;
}

double min(double a, double b) {
  return a < b ? a : b;
}
```

- Template di funzione: fornisce un algoritmo usato per generare automaticamente istanze particolari di una funzione che si differenziano per il tipo.

Template di funzione

```
template <class T>
T min(T a, T b) {
  return a < b ? a : b;
}

int main() {
  // int min(int, int).
  min(10, 20);

  // double min(double, double).
  min(10.0, 20.0);

  return 0;
}
```

TEMPLATE DI CLASSE

- Utilità delle classi generiche: classe che sfrutta una logica che può essere generalizzata (es. pila, coda, ...).

TEMPLATE DI CLASSE

- Utilità delle classi generiche: classe che sfrutta una logica che può essere generalizzata (es. pila, coda, ...).
- Può eseguire operazioni definite dal programmatore su qualunque tipo di dati; sarà compito del *compilatore* generare l'esatto tipo di oggetto in base al tipo specificato nella creazione dell'oggetto.

TEMPLATE DI CLASSE

➜ Generica pila di oggetti della classe T:

```
template <class T>
class Pila {
public:
  Pila();
  ~Pila();
  void push(T x);
  T pop();
  bool empty() const;
  unsigned int capacita() const;
  unsigned int num_elementi() const;
private:
  T* A;
  int top;
};
```

TEMPLATE DI CLASSE

➜ Dichiarazione di classe generica:

```
template <class T>
class Pila {
  ...
};
```

TEMPLATE DI CLASSE

➜ Dichiarazione di classe generica:

```
template <class T>
class Pila {
  ...
};
```

➜ Definizioni di metodi all'esterno (con operatore ::) di classi generiche:

```
template <class T>
T
Pila<T>::Pop() {
  ...
};
```

TEMPLATE DI CLASSE

➜ Dichiarazione di classe generica:

```
template <class T>
class Pila {
   ...
};
```

➜ Definizioni di metodi all'esterno (con operatore ::) di classi generiche:

```
template <class T>
T
Pila<T>::Pop() {
   ...
};
```

➜ Istanziazione di un template di classe:

```
Pila<int> P1;
Pila<char> P2;
```

ESERCIZI

1. Creare una funzione *generica* `scambia()` che non ritorna nulla e che scambia i valori delle due variabili con cui viene chiamata. Fare un piccolo main di prova (`scambia_template.cc`).

2. Riprendiamo l'esercizio 2. della lezione X: riscrivere la classe "Pila" aggiungendo la possibilità di raddoppiare la capacità della pila qualora venga fatta un'operazione di push su pila piena. Realizzare quanto chiesto con un metodo privato della classe, `raddoppia()`, invocato all'interno del metodo `push()`. Fare la suddetta pila "generica" (cioè con i template) e fare un piccolo main di prova. Usare anche le eccezioni (`pila_template.cc`).