TEMPLATE

TEMPLATE

- Partiamo da un esempio:
- ```
  int min( int a, int b ) {
      return (a < b) ? a : b;
  }
  ```
  ```

TEMPLATE

- Partiamo da un esempio:
- ```
  int min( int a, int b ) {
      return (a < b) ? a : b;
  }
  ```

- ```
  min(3, 5);        // 3
  ```

TEMPLATE

- Partiamo da un esempio:
- ```
  int min( int a, int b ) {
      return (a < b) ? a : b;
  }
  ```
- ```
  min(3, 5);        // 3
  ```
- Ma
- ```
  min(3.9, -18.72) ?
  ```
  ```

TEMPLATE

- Potremmo scrivere due funzioni diverse, una per ogni tipo:
- ```
  int min_int( int a, int b ) {
      return (a < b) ? a : b;
  }
  ```

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

TEMPLATE

- Potremmo scrivere due funzioni diverse, una per ogni tipo:
- ```
  int min_int( int a, int b ) {
      return (a < b) ? a : b;
  }
  ```
- ```
  double min_double( double a, double b ) {
      return (a < b) ? a : b;
  }
  ```
- ```
  size_t min_size_t( size_t a, size_t b ) {
      return (a < b) ? a : b;
  }
  ```
  ```

TEMPLATE

- Potremmo scrivere due funzioni diverse, una per ogni tipo:
- ```
  int min_int( int a, int b ) {
      return (a < b) ? a : b;
  }
  ```
- ```
  double min_double( double a, double b ) {
      return (a < b) ? a : b;
  }
  ```
- ```
  size_t min_size_t( size_t a, size_t b ) {
      return (a < b) ? a : b;
  }
  ```
- ```
  float min_float( float a, float b ) {
      return (a < b) ? a : b;
  }
  ```

TEMPLATE

- Possiamo riassumere i problemi di questo approcio:

- Più copie della stessa funzione (che fanno la stessa cosa!) per ogni tipo utilizzato
- Scrivere una nuova funzione per ogni nuovo tipo introdotto nel codice
- Specificare il tipo ogni volta che la funzione viene richiamata

TEMPLATE

- Possiamo riassumere i problemi di questo approcio:

- Più copie della stessa funzione (che fanno la stessa cosa!) per ogni tipo utilizzato
- Scrivere una nuova funzione per ogni nuovo tipo introdotto nel codice
- Specificare il tipo ogni volta che la funzione viene richiamata

- Una soluzione potrebbe essere l'overload di funzione, ma devo comunqe scrivere una funzione per ogni tipo:

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

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

TEMPLATE

- Template permettono il riutilizzo della stessa funzione su tipi diversi:

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

- Ma cos'è un tempate?
  ```

TEMPLATE

- In origine, abbiamo specificato la funzione min su variabili di tipo int:

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

- E per definirla sui double è bastato cambiare il tipo:

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

- Possiamo istruire il compilatore con questa regola costruttiva sotto forma di una funzione template, specificando cosa debba essere cambiato tra una definizione e l'altra. Il compilatore sostituirà il parametro per noi!

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

TEMPLATE

- In fase di compilazione, ogniqualvolta il compilatore si accorge che la funzione template viene usata su un nuovo tipo (fino a quel momento non ancora definito), genererà una nuova funzione per il tipo corretto.
- Questo processo è detto **template instantiation**

- Per utilizzarle, e sufficiente specificare il tipo:

```
int a = 4, b = 100;
int c = min<int>(a, b);
```

```
double f = 4.1, l = 10.0;
double h = min<double>(f, l);
```

TEMPLATE

- Possiamo riassumere i problemi di questo approcio:

- ~~Più copie della stessa funzione (che fanno la stessa cosa!) per ogni tipo utilizzato~~
- ~~Scrivere una nuova funzione per ogni nuovo tipo introdotto nel codice~~
- Specificare il tipo ogni volta che la funzione viene richiamata

TEMPLATE

- Possiamo riassumere i problemi di questo approcio:

- ~~Più copie della stessa funzione (che fanno la stessa cosa!) per ogni tipo utilizzato~~
- ~~Scrivere una nuova funzione per ogni nuovo tipo introdotto nel codice~~
- Specificare il tipo ogni volta che la funzione viene richiamata

- In realtà, il compilatore potrebbe dedurre il tipo a seconda dell'uso...

```
int a = 4, b = 100;
int c = min(a, b);
```

-
```
double f = 4.1, l = 10.0a;
double h = min(f, l);
```

TEMPLATE

- Template possono fare molto di più
- Tecnica di metaprogrammazione in cui le template sono usate dal compilatore per generare codice sorgente temporaneo che viene unito al codice sorgente del programma in compilazione
- Il linguaggio di scrittura delle template è Turing completo. Eseguito interamente dal compilatore.
- Paradigma simil-funzionale: tutte le variabili sono non-mutable; utilizzo della ricorsione