20 - Compilazione dei Template (lezione 29-11-2024)

La compilazione del template avviene in due fasi:

1. **definizione del template**: il compilatore analizza il codice del template in modo generico, senza conoscere i tipi o i valori specifici che verranno utilizzati in fase di istanziazione. Può effettuare solo controlli di tipo sintattico e di semantica statica per le parti di codice indipendenti dai parametri del template.
2. **istanziazione del template**: qui il compilatore analizza il template utilizzando i tipi o i valori concreti forniti per i parametri. È in questa fase che il compilatore verifica se tutte le operazioni dipendenti dal tipo sono valide per i tipi specificati.
   nella prima fase il compilatore lavora con informazioni incomplete:

```
template <typename T >
void incr (int& i, T& t ) {
    ++i; // espressione indipendente dai parametri del template dove è possibile
effettuare tutti i controlli di sicurezza
    ++t; // espressione dipendente dai parametri del template
}
```

Conseguenze:

- la definizione di un template deve essere disponibile in tutti i punti del programma che richiedono l'istanziazione;
- dal momento che l'istanziazione può avvenire in più unità di traduzione, per non violare il principio DRY (Don't Repeat Yourself) le definizione del template si trova all'interno dell'header file;
- la ODR (One Definition Rule) ammette che le funzioni templatiche siano definite più di una volta (come le funzioni inline) a patto che:
  - si trovino in unità di traduzione diverse,
  - siano definite con la stessa sintassi,
  - siano definite con la stessa semantica.

Esistono tre modi per organizzare i codici sorgente quando si definiscono le classi templatiche:

1. **includere definizioni e dichiarazioni nello stesso file**: includere tutto il codice dei template prima di ogni loro uso in un file header (la definizione completa del template è visibile prima che il compilatore incontri il codice che lo utilizza);
   - utilizzato più frequentemente;
   - "modello di compilazione dei template per inclusione";

```
// File Stack.hh
template <typename T>
class Stack {
  void push(const T& value);
};

template <typename T>
void Stack<T>::push(const T& value) {
```

```
  // implementazione
}
```

2. **separare dichiarazioni e definizioni**: le dichiarazioni del template vengono incluse in un file `.hh`, mentre le definizioni vengono inserite in un file `.cpp` separati. È comunque necessario includere il file delle definizioni ogni volta che si usa il template.
   - variante del primo metodo;
   - utilizzato solo se necessario (ad esempio nel caso di funzioni templatiche ricorsive).

```
// File Stack.h (dichiarazioni)
template <typename T>
class Stack {
  void push(const T& value);
};

// File Stack.cpp (definizioni)
template <typename T>
void Stack<T>::push(const T& value) {
  // implementazione
}
```

3. **istanziazioni esplicite** utilizzare le istanziazioni esplicite di template (dichiarare e definire i template in file separati, garantendo che le istanziazioni vengano generate in un'unica unità di traduzione):
   - includere solo le dichiarazioni dei template e le dichiarazioni di istanziazione esplicita prima di ogni loro uso nell'unità di traduzione
   - assicurarsi che le definizioni dei template e le definizioni di istanziazione esplicita siano fornite una sola volta in un'altra unità di traduzione
     Più complicato, meno flessibile, utilizzato per ridurre i tempi di compilazione.

```
// File Stack.h (dichiarazioni)
template <typename T>
class Stack {
  void push(const T& value);
};

// File Stack.cpp (definizioni)
template <typename T>
void Stack<T>::push(const T& value) {
  // implementazione
}

// Istanziazioni esplicite
template class Stack<int>;
template class Stack<double>;
```

Keyword `typename`

Quando un template fa riferimento a un tipo dipendente da un parametro template, il compilatore può confondersi tra un valore e un tipo. Per chiarire che si tratta di un tipo, si utilizza la keyword `typename`.
Problema: esempio non templatico

```
struct S {
  using value_type = int;
};

void foo(const S& s) {
  S::value_type* ptr; // ok
}
```

Il compilatore capisce immediatamente che `S::value_type` è un tipo. Tuttavia, se templatizziamo la classe `S`:

```
template <typename T>
struct S {
  using value_type = T;
};

template <typename T>
void foo(const S<T>& s) {
  S<T>::value_type* ptr; // errore
}
```

Il compilatore non sa se `S<T>::value_type` è un tipo o un valore, e quindi segnala un errore.
Soluzione: aggiungendo la keyword `typename`, indichiamo esplicitamente al compilatore che si tratta di un tipo:

```
template <typename T>
void foo(const S<T>& s) {
  typename S<T>::value_type* ptr; // ok
}
```

In alcuni casi però il problema può non essere immediatamente evidente:

```
int p = 10;

template <typename T>
void foo(const S<T>& s) {
  S<T>::value_type* p;   // compila senza errori
}
```

Il compilatore interpreta l'istruzione come un'operazione binaria (`S<T>::value_type * p`), dove `p` è la variabile globale dichiarata come intero. Questo comportamento può portare a errori logici difficili da individuare.