# C++ Templates

## from zero to hero

Nicola Gigante

C++ User Group Udine

21 ottobre 2014

# Di cosa parleremo

*A book titled "Everything you need to know about ..." is certain to be elementary.*

*(John D. Cook)*

# C++ Templates

## from zero to hero (parte 1 di n)

# Di cosa parleremo

1. Programmazione generica
2. Template: funzionamento generale
3. Caso concreto: la Standard Template Library
4. Andando oltre

# Programmazione generica

- Scrittura di codice che funziona indipendentemente dal tipo dei dati su cui opera.
- Non è duck-typing: il tipo è *generico* ma *noto* a compile-time.
- Pionieri del paradigma: ML e Ada (anni '70)
- Introdotta nel C++ da Alexander Stepanov nei primi anni '90.
- Adottata successivamente in misure diverse da vari linguaggi (es. Generics in C# e Java)

# Vantaggi

- Riuso del codice
- Type-safety
- Astrazione
- Velocità

# C'erano una volta C, Pascal, Fortran...

```
int abs(int x) {
    return x < 0 ? -x : x;
}

double fabs(double x) {
    return x < 0 ? -x : x;
}

double fabsf(float x) {
    return x < 0 ? -x : x;
}
```

# C'erano una volta C, Pascal, Fortran...

```
float x = accurate_calculation();
// ...
// My businness-critical code
// ...

// Ops...
float we_cant_even_lose_a_penny = abs(x);
```

# Primo passo: overloading

```
int abs(int x) {
    return x < 0 ? -x : x;
}

double abs(double x) {
    return x < 0 ? -x : x;
}

double abs(float x) {
    return x < 0 ? -x : x;
}
```

# Primo passo: overloading

```
float x = accurate_calculation();
// ...
// My businness-critical code
// ...

// OK!
float we_wont_even_lose_a_penny = abs(x);
```

# Prima soluzione: macro

```
#define abs(x) ((x) < 0 ? -(x) : (x))
```

# Pare di no...

```
int my_strange_number = abs(i++);

int we_have_time_to_waste = abs(slowfunction());
```

# Seconda soluzione: type oblivion

```
void
qsort(void *base, size_t nel, size_t width,
      int (*compare)(const void *, const void *));
```

# Pare di no...

```
int intcmp(const void *pa, const void *pb) {
    int a = *(int*)pa, b = *(int*)pb;
    return a < b ? -1 :
           a > b ?  1 : 0;
}

// 20 lines of code later...
int *myarray = source();
int arraylen = please_dont_get_this_wrong();

qsort(myarray, arraylen, sizeof(int), &intcmp);
```

# Pare di no...

Dovrebbe essere:

```
sort(myarray);
```

# Piccole funzioni crescono...

```
$ wc -l linear_algebra/*.c
     10042 int_gauss_elimination.c
     10042 float_gauss_elimination.c
     10042 double_gauss_elimination.c
     10042 complex_gauss_elimination.c
```

# Object Oriented Programming

La programmazione orientata agli oggetti fornisce delle soluzioni parziali a questi problemi.

- Il dynamic dispatch *nasconde* il tipo degli oggetti in gioco, impedendo al compilatore e al programmatore di ragionare su quello che accade nel codice.
- Il layout dei dati in memoria è inadatto agli algoritmi che li manipolano (vedi l'intervento successivo, Data Oriented Design di Matteo Bertello).
- Il dogma "tutto è un oggetto" è spesso innaturale, vedi le diffusissime classi `XyzManager`, le classi singleton, ecc...

# Soluzione: programmazione generica

```
template<typename T>
T abs(T x) {
    return x < 0 ? -x : x;
}
```

- abs è una *funzione template*.
- Il tipo T, introdotto dalla parola chiave `typename`, è un parametro del template.
- Dentro al template, T è un tipo noto e fissato.

# Funzioni template: utilizzo

```
int answer = abs(-42);
float pi = abs(-3.14);
```

- L'utilizzo è totalmente trasparente.
- Il compilatore *deduce* i parametri del template dal tipo degli argomenti.
- Al momento della chiamata il template viene *istanziato*, creando una funzione per i tipi specifici richiesti.

# Code to the interface

Il corpo stesso della funzione stabilisce i vincoli a cui il tipo deve soddisfare.

```
int x = abs("hello");

$ clang++ template.cpp
template.cpp:3:18: error: invalid argument type
                          'const char *' to unary expression
```

# Performance

- I template vengono istanziati *dal compilatore*, a compile-time.
- Dopo l'istanziazione quella che viene chiamata è una normale funzione.
- All'interno dell'istanza del template il tipo è precisamente *noto*.
- Nessun overhead a runtime.[1]
- La staticità del tipo consente maggiori ottimizzazioni.

[1] Ma attenzione al code-bloat

# Visibilità

- Per poter istanziare il template, la definizione deve essere *visibile* nel punto della chiamata.
- Conseguenza: la funzione template deve essere specificata nello stesso file o in un header.
- Il template è soggetto a *inlining*, ulteriore fonte di ottimizzazione.

# Deduzione dei tipi

- I parametri template di una funzione template vengono *dedotti* automaticamente a partire dai tipi degli argomenti della funzione.

# Deduzione dei tipi

- Meccanismo simile ad una unificazione, o un pattern matching.

```
template<typename T>
T sum(const T *a, int len) {
   T s = 0;
   for(int i = 0; i < len; ++i)
      s += a[i];
   return s;
}

int   *iarray = ...;
float *farray = ...;
int   isum = sum(iarray, 42); // T = int
float fsum = sum(farray, 42); // T = float
```

# Automatic type deduction

A volte la genericità rende difficile sapere che tipo nominare.

```
// Impossibile in C++98
template<typename T1, typename T2>
???? sum(T1 v1, T2 v2) {
  return v1 + v2;
}
```

# Automatic type deduction

Soluzione: automatic type deduction

```
// Possibile in C++11
template<typename T1, typename T2>
auto sum(T1 v1, T2 v2) -> decltype(v1 + v2) {
  return v1 + v2;
}
```

# Automatic type deduction

Soluzione: automatic type deduction

```
// Banale in C++14
template<typename T1, typename T2>
auto sum(T1 v1, T2 v2) {
  return v1 + v2;
}
```

# Classi template

- Dichiarazione di template di intere classi invece che di sole funzioni.
- Permettono di dichiarare *strutture dati generiche*.

```
std::vector<int>   vi = {2, 3, 5};
std::vector<float> vf = {1.41, 1.73, 2.24};
```

# Esempio: un generico stack

```
#include <vector>

template<typename T>
class stack {
public:
   stack() = default;
   stack(stack const&) = default;

   void pop() {
      _data.pop_back();
   }

   void push(const T &v) {
      _data.push_back(v);
   }

   T top() const {
      return _data.last();
   }

private:
   std::vector<T> _data;
};
```

# Utilizzo del nostro stack

```
stack<int> s;
s.push(42);

std::cout << s.top() << "\n";

s.pop();
```

# Argomenti di default per parametri template

```
template<typename T, typename Container = std::vector<T>>
class stack {
public:
   // ...

private:
   Container _data;
};

stack<T>                  vector_stack;
stack<T, std::list<T>> list_stack;
```

# Definizione out-of-line di membri di una classe template

- In una normale classe è spesso necessario definire out-of-line le funzioni membro:

```
// header file
class MyClass {
  void myfunc();
};


// implementation file
void MyClass::myfunc() {
  // ...
};
```

# Definizione out-of-line di membri di una classe template

- Una classe template non fa differenza, ma la definizione deve comunque essere visibile.

```
// header file
template<typename T>
class MyClass {
  void myfunc();
};

// same file (or visible anyway)
template<typename T>
void MyClass<T>::myfunc() {
  // ...
};
```

# Specializzazioni

A volte si può voler cambiare il funzionamento di un template per un tipo o per un insieme specifico di tipi.

- Perchè l'implementazione generica non è adatta
- Perchè esiste un implementazione specializzata più efficiente

# Specializzazioni

Specializzazione di una classe template:

```
namespace std {
  template<typename T>
  class vector { ... };

  template<>
  class vector<bool> { ... };
}
```

# Specializzazioni parziali

Specializzazione parziale:

```
template<typename T1, typename T2>
class MyClass { ... };

template<typename T1, typename T2>
class MyClass<T1 *, T2 *> { ... };

template<typename T>
class MyClass<T, T> { ... };
```

# Nella pratica

Don't panic. Alcuni consigli della prima ora:

- Se si implementano strutture dati o algoritmi, considerare di implementarli genericamente.
- Se si sta organizzando una gerarchia di classi, chiedersi se il dynamic dispatch a runtime è davvero necessario.
- Usare clang per ottenere messaggi di errore decenti.
- Usare le utility della libreria standard quando possibile.
- Assicurarsi di utlizzare documentazione aggiornata al C++11/14.

# La Standard Template Library

- Libreria di classi template che implementano strutture dati generiche di uso frequente.
- Collezione di algoritmi base che operano genericamente su qualsiasi container.
- Esempio chiave di programmazione generica.

# STL: Container e algoritmi

Collezioni di elementi di tipo omogeneo.

- Container sequenziali:
  `std::vector<T>`, `std::list<T>`, `std::deque<T>`,
  `std::array<T, N>`, ecc...
- Container associativi:
  `std::map<K,T>`, `std::set<T>`, ecc...
- Algoritmi base di uso molto frequente:
  `std::copy`, `std::find`, `std::sort`, `std::accumulate` (fold),
  `std::transform` (map/zip), ecc...

# Consigliati dalla redazione

Alcuni container forniscono in modo molto efficiente funzionalità di base importanti:

- `std::vector<T>`:
  Vettore di dimensione variabile, elementi contigui in memoria, accesso diretto.
- `std::array<T, N>`:
  Array contiguo di dimensione fissata a *compile-time*. Astrae gli array grezzi del C.[2]
- `std::map<K, T>`:
  Container associativo, chiavi di tipo `K`, elementi di tipo `T`.
- `std::set<T>`:
  Insieme di elementi di tipo `T`.

[2] Notare il non-type template parameter

# Gestire la complessità

$M$ container e $N$ algoritmi.

- $M \times N$ implementazioni?
- No.
- Funzioni generiche che operano su qualsiasi container indistintamente.

# Iteratori

- Elemento di astrazione base della libreria
- Una sequenza di elementi è identificata da un *range* di iteratori $[\mathrm{begin}, \mathrm{end})$
- Tutti gli algoritmi operano su range generici di iteratori.
- I container forniscono gli iteratori ai propri elementi.

# Esempio

```
#include <vector>
#include <algorithm>

std::vector<int> v = // ...;

std::sort(v.begin(), v.end());
```

# Iteratori

L'interfaccia degli iteratori imita quella dei puntatori.
Se `it` è un iteratore:

- `*it` ottiene l'elemento puntato dall'iteratore.
- `++it`/`--it` fa puntare l'iteratore all'elemento successivo/precedente.

In effetti, gli iteratori di `std::vector<T>` non sono altro che puntatori `T*`.

# Esempio

```
namespace std {
  template<typename Iterator>
  Iterator max_element(Iterator begin, Iterator end)
  {
    Iterator max = begin++;

    for(Iterator it = begin; it != end; ++it) {
      if(*it > *max)
        max = it;
    }

    return max;
  }
}
```

# Esempio

```
std::list<int> l = // ...;

int m = *std::max_element(l.begin(), l.end());

std::cout << "My␣beautiful␣result:␣" << m << "\n";
```

# Di che tipo sono gli iteratori?

Ogni container fornisce il proprio tipo di iteratore:

```
template<typename T>
class list {
public:
  typedef /* unknown */ iterator;
  typedef /* unknown */ const_iterator;
};

template<typename T>
class vector {
public:
  typedef T       * iterator;
  typedef T const* const_iterator;
};
```

# Di che tipo sono gli iteratori?

Quindi basta riferirsi al tipo innestato:

```
std::vector<int> v = // ...;

for(std::vector<int>::iterator it = v.begin();
    it != v.end();
    ++it)
{
  std::cout << *it << "\n";
}
```

# Di che tipo sono gli iteratori?

Oppure dedurre il tipo in automatico (C++11):

```
std::vector<int> v = // ...;

for(auto it = v.begin();
    it != v.end();
    ++it)
{
  std::cout << *it << "\n";
}
```

# Di che tipo sono gli iteratori?

Oppure usare il range-for (C++11):

```
std::vector<int> v = // ...;

for(int x : v)
{
  std::cout << v << "\n";
}
```

# Piccolo cavillo sintattico

Per nominare un tipo innestato all'interno di un altro, bisogna usare la parola chiave **typename** se il tipo è *type-dependent*.

```
void example1(std::vector<int> v) {
  std::vector<int>::iterator it = v.begin();
  // code...
}

template<typename T>
void example2(std::vector<T> v) {
  typename std::vector<T>::iterator it = v.begin();
}
```

# Oh, oh, oppa functional style

Le espressioni lambda introdotte nel C++11 giocano bene assieme alla STL.

```
#include <cmath>

const float pi = 3.1412659;
std::vector<float> v = {0, pi/2, pi, 3/2*pi, 2*pi};

std::transform(v.begin(), v.end(),
  [](float x) {
     return sin(x) + cos(x);
  });
```

# Dietro le quinte

```
namespace std {
  template<typename Iterator, typename Callable>
  void transform(Iterator begin, Iterator end, Callable f)
  {
     for(Iterator it = begin; it != end; ++it)
        *it = f(*it);
  }
}
```

# Iterator categories

La genericità non deve andare a discapito delle performance.

- Gli iteratori di `std::vector<T>` supportano l'accesso diretto, perchè puntano a elementi contigui in memoria.
- Gli iteratori di `std::list<T>` non possono supportare efficientemente la stessa operazione.
- Entrambi possono andare sia in avanti che indietro. Gli iteratori di `std::forward_list<T>` non possono andare indietro.

# Iterator categories

Gli iteratori sono divisi in *categorie* diverse a seconda delle operazioni che supportano.

InputIterator Minimo comune denominatore

ForwardIterator Spostamento in avanti

BidirectionalIterator Spostamento in entrambe le direzioni

RandomAccessIterator Accesso casuale

OutputIterator Gli elementi puntati sono modificabili

# Iterator categories

Ogni container definisce la propria categoria di iteratori

- `std::vector<T>` fornisce dei RandomAccessIterator
- `std::list<T>` fornisce dei ForwardIterator

Ogni algoritmo standard richiede iteratori di una specifica categoria:

- `std::sort()` richiede RandomAccessIterator
- `std::transform()` si accontenta di ForwardIterator

# Distinguere tra categorie

```
namespace std {
 template<typename It>
 auto distance(It begin, It end) {
  return distance(begin, end,
    typename iterator_traits<It>::iterator_category());
 }

 template<typename It>
 auto distance(It begin, It end, random_access_iterator_tag) {
   return end - begin;
 }

 template<typename It>
 int distance(It begin, It end, input_iterator_tag) {
   int n = 0;
   for(It it = begin; it != end; ++it, ++n);
   return n;
 }
}
```

# Andando oltre

L'introduzione del meccanismo di specializzazione delle classi template ha avuto una inaspettata conseguenza:

```
template<int I>
struct fact {
  static const int value = I * fact<I - 1>::value;
};

template<>
struct fact {
  static const int value = 1;
};

std::cout << "fact(5)=" << fact<5>::value << "\n";
```

# Type-level computation

Abbiamo a disposizione un piccolo, rudimentale linguaggio puramente funzionale, interpretato dal compilatore durante il type-check del programma.

```
struct EmptyList;

template<int Head, typename Tail>
struct List {
  static const int head = Head;

  typedef Tail tail;
};

typedef List<23, List<30, List<18, EmptyList>>> MyList;
```

# Type-level computation

```
template<typename L>
struct Length {
  static const int value =
      1 + Length<typename L::tail>::value;
};

template<>
struct Length<EmptyList> {
  static const int value = 0;
};

std::cout << Length<MyList>::value << "\n";
```

# Type-level computation

La possibilità di eseguire manipolazioni complesse dei tipi a compile-time apre varie possibilità:

- Forzare tramite il type system gli invarianti del codice.
- Generare codice specifico e ottimizzato in funzione di come viene richiamata la propria interfaccia.
- Esempi notevoli (in ordine di improbabilità):
  - Boost.Unit
  - Sql++11
  - Eigen / Armadillo
  - TCalc
  - Boost.Spirit

# Per approfondire

Alcuni riferimenti obbligatori:

- A Tour of C++, Bjarne Stroustrup
- The C++ Programming Language, 4th Edition, Bjarne Stroustrup
- C++ Templates: The Complete Guide, David Vandervoorde e Nicolai M.Josuttis

Materiale online:

- Stack Overflow, tag `c++faq`
- C++ FAQs: `http://www.parashift.com/c++-faq/`
- `http://cppreference.com`

E ovviamente gli incontri del C++ User Group di Udine `;-)`