

Template



Funzione che ritorna il minimo tra due variabili di un tipo che ammette il test booleano `operator<`

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

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

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

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

Una soluzione attraente ma subdolamente pericolosa potrebbe essere la definizione di **macro** per il preprocessore.

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

il preprocessore sostituisce:

```
min(10,20)
```

con

```
10 < 20 ? 10 : 20
```

```
min(3.5,2.1)
```

con

```
3.5 < 2.1 ? 3.5 : 2.1
```

Non abbiamo ottenuto una “vera funzione”

```
min(++i,--j)
```

verrebbe espansa dal preprocessore con:

```
++i < --j ? ++i : --j
```

e provocherebbe una doppia applicazione di ++ o --.

```
int i=3, j=6;  
cout << min(++i,--j);  
// stampa 5  
// NON stampa 4
```


Generic programming

From Wikipedia, the free encyclopedia

Not to be confused with [Genetic programming](#).


In the simplest definition, **generic programming** is a style of [computer programming](#) in which algorithms are written in terms of [types to-be-specified-later](#) that are then *instantiated* when needed for specific types provided as [parameters](#). This approach, pioneered by ML in 1973,^{[1][2]} permits writing common [functions](#) or [types](#) that differ only in the set of types on which they operate when used, thus reducing [duplication](#). Such software entities are known as *generics* in [Ada](#), [Delphi](#), [Eiffel](#), [Java](#), [C#](#), [F#](#), [Objective-C](#), [Swift](#), and [Visual Basic .NET](#); *parametric polymorphism* in [ML](#), [Scala](#), [Haskell](#) (the Haskell community also uses the term "generic" for a related but somewhat different concept) and [Julia](#); *templates* in [C++](#) and [D](#);

Template (C++)

From Wikipedia, the free encyclopedia

Templates are a feature of the **C++** programming language that allows functions and classes to operate with **generic types**. This allows a function or class to work on many different **data types** without being rewritten for each one.

Templates are of great utility to programmers in C++, especially when combined with **multiple inheritance** and **operator overloading**. The **C++ Standard Library** provides many useful functions within a framework of connected templates.

Major inspirations for C++ templates were the parameterized modules provided by **CLU** and the generics provided by **Ada**.^[1] 

Definizione della funzione `min` come template:

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

Quindi:

```
int main() {
    int i,j,k;
    orario r,s,t;
    ...
    // istanziamento implicito del template
    k = min(i,j);
    t = min(r,s);
    // oppure: istanziamento esplicito del template
    k = min<int>(i,j);
    t = min<orario>(r,s);
}
```


- I parametri di un template possono essere:
 - **Parametri di tipo**: si possono istanziare con un tipo qualsiasi
 - **Parametri valore di qualche tipo**: si possono istanziare con un valore costante del tipo indicato
- Un template **non è codice compilabile**: istanziazione **implicita** o **esplicita** di un template di funzione
- **Processo di deduzione degli argomenti** di un template nella istanziazione implicita (dove il **tipo di ritorno non si considera mai**)



Attenzione: nell'istanziamento implicita il **tipo di ritorno** dell'istanza del template non viene **mai considerato nella deduzione degli argomenti** (essendo opzionale l'uso del valore ritornato)

```
int main() {  
    double d; int i,j;  
    ...  
    d = min(i,j); // istanzia int min(int,int)  
                  // e quindi usa la conversione  
                  // int => double  
}
```

L'algoritmo di deduzione degli argomenti di un template procede esaminando tutti i parametri attuali passati al template di funzione da sinistra verso destra. Se si trova uno stesso parametro **T** del template che appare più volte come parametro di tipo, l'argomento del template dedotto per **T** da ogni parametro attuale deve essere **esattamente** lo stesso.

```
int main() {  
    int i; double d, e;  
    ...  
    e = min(d,i);  
    // NON COMPILA  
    // Si deducono due diversi argomenti del  
    // template: double e int  
}
```

L'istanziazione dei parametri di tipo **deve essere univoca**.

Nell'algoritmo di deduzione degli argomenti sono ammesse **quattro tipologie di conversioni** dal tipo dell'argomento attuale al tipo dei parametri del template:

- (1) conversione da lvalue in rvalue, i.e. da **T&** a **T**;
- (2) da array a puntatore, i.e. da **T[]** a **T***;
- (3) conversione di qualificazione costante, i.e. da **T** a **const T**;
- (4) conversione da rvalue a riferimento costante, i.e. da rvalue di tipo **T** a **const T&**

```
template <class T> void E(T x) {...};
template <class T> void F(T* p) {...};
template <class T> void G(const T x) {...};
template <class T> void H(const T& y) {...};

int main() {
    int i = 6; int& x = i;
    int a[3] = {4,2,9};
    E(x);           // (1): istanzia void E(int)
    F(a);           // (2): istanzia void F(int*)
    G(i);           // (3): istanzia void G(const int)
    H(7);           // (4): istanzia void H(const int&)
}
```


Istanziazione esplicita degli argomenti dei parametri del template di funzione. Nell'istanziazione esplicita è possibile applicare *qualsiasi conversione implicita* di tipo per i parametri attuali del template di funzione.

```
int main() {  
    int i; double d, e; ...  
    e = min<double>(d,i);  
    // compila!  
    // istanzia: double min(double,double)  
    // e quindi converte implicitamente i  
    // da int a double  
}
```



Un parametro di una funzione può essere un *referimento ad un array statico*. In questo caso, la dimensione costante dell'array è parte integrante del tipo del parametro e il compilatore controlla che la dimensione dell'array parametro attuale coincida con quella specificata nel tipo del parametro.

```
int min(int (&a)[3]) { // array di 3 int
    int m = a[0];
    for (int i = 1; i < 3; i++)
        if (a[i] < m) m = a[i];
    return m;
}

int ar[4] = {5,2,4,2};
cout << min(ar); // non compila!
```

```
template <class T, int size>
T min(T (&a)[size]) {
    T vmin = a[0];
    for (int i = 1; i < size; i++)
        if (a[i] < vmin) vmin = a[i];
    return vmin;
}
```

Parametro valore



```
int main() {
    int ia[20];
    orario ta[50];
    ...
    cout << min(ia);
    cout << min(ta);
    // oppure
    cout << min<int,20>(ia);
    cout << min<orario,50>(ta);
}
```

Modello di compilazione del template



Modello di compilazione del template

- 1) La definizione del template deve essere visibile all'utilizzatore del template?
- 2) Il file header del template cosa deve includere?
 - Solo la dichiarazione del template?
 - Sia la dichiarazione che la definizione?
- 3) Cosa si compila effettivamente?

Compilazione per inclusione

Definizione del template in un “header file” che deve essere sempre incluso dal codice che necessita di istanziare il template. Non vi è quindi il concetto di compilazione separata di un template.

Problema 1: No information hiding

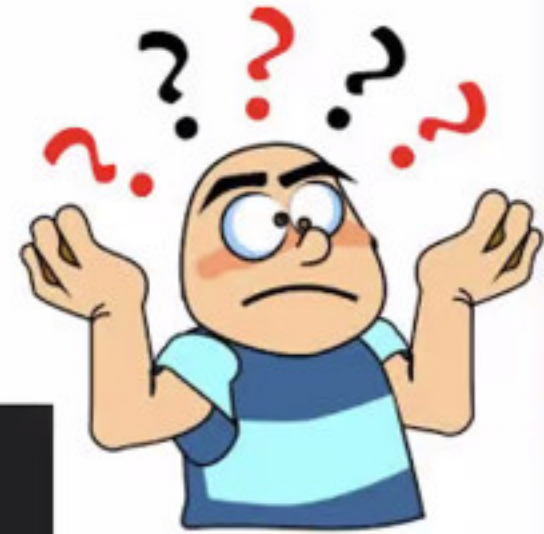
Problema 2: Istanze multiple del template



Compilazione per inclusione

Problema 1: No information hiding

Nessuna soluzione (pazienza)



Compilazione per inclusione

Problema 1: No information hiding

Nessuna soluzione



Problema 2: Istanze multiple del template

Dichiarazioni esplicite di istanziazione



Dichiarazione esplicita di istanziazione del template di funzione:

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

al tipo `int` ha la forma:

```
template int min(int,int);
```

Forza il compilatore a generare il codice dell'istanza del template relativa al tipo `int`.

Dichiarazione esplicita di istanziazione del template di funzione:

`-fno-implicit-templates` Never emit code for **non-inline templates** which are instantiated implicitly (i.e. by use); only emit code for explicit instantiations. See Template Instantiation, for more information.

```
g++ -fno-implicit-templates
```

Dichiarazione esplicita di istanziazione del template di funzione:

```
// file min.h
template <class Tipo>
Tipo min(Tipo a, Tipo b) {
    return a < b ? a : b;
}
```

```
// file usedTemplates.cpp
#include "min.h"
template int min(int,int);
template orario min(orario,orario);
```

```
// file main.cpp
#include "min.h"
#include "orario.h"
int main() {
    cout << min(9,3) << min(3*16,50-2);
    cout << min(orario(4), orario(4,5,6));
}
```

```
g++ -fno-implicit-templates -c main.cpp
g++ -fno-implicit-templates -c usedTemplates.cpp
g++ main.o usedTemplates.o
```

Compilazione per separazione

Dichiarazione del template separata dalla sua definizione.

Parola chiave **export** non supportata dai compilatori. Compilatori Comeau ed icc (Intel C++ Compiler) tra i pochissimi che supportavano **export**.

Il comitato C++11 ha deciso di rimuovere **export** dallo standard. Rimane una keyword riservata.

Si veda: H.Sutter and T.Plum: “Why we can’t afford export”





There are fundamental drawbacks to the use of templates:

1. Historically, some compilers exhibited poor support for templates. So, the use of templates could decrease code portability.
2. Many compilers lack clear instructions when they detect a template definition error. This can increase the effort of developing templates, and has prompted the development of [Concepts](#) for possible inclusion in a future C++ standard.
3. Since the compiler generates additional code for each template type, indiscriminate use of templates can lead to [code bloat](#), resulting in larger executables.
4. Because a template by its nature exposes its implementation, injudicious use in large systems can lead to longer build times.
5. It can be difficult to [debug code](#) that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.
6. Templates of Templates (nesting) are not supported by all compilers, or might have a max nesting level.
7. Templates are in the headers, which require a complete rebuild of all project pieces when changes are made.
8. No information hiding. All code is exposed in the header file. No one library can solely contain the code.

An incomplete list of C++ compilers

Modified August 20, 2014

I ([Bjarne Stroustrup](#)) am often asked to recommend a C++ compiler. However, I don't make recommendations; that would be too much like taking sides in commercial wars. Also, I don't know every C++ compiler; there are simply too many "out there".

I recommend that people take Standard conformance very seriously when considering a compiler. If you can, avoid any compiler that doesn't closely approximate the ISO standard or fails to supply a solid implementation of the standard library. The recent releases from all the major C++ vendors do that.

Some compilers that can be downloaded for free (do check their conditions/licenses before attempting commercial use):

- [Apple C++](#). It also comes with OS X on the developer tools CD.
- [Bloodshed Dev-C++](#). A GCC-based (Mingw) IDE.
- [Clang C++](#). A relatively new and very active development.
- [Cygwin \(GNU C++\)](#)
- [Digital Mars C++](#)
- [Mentor Graphics - Lite edition](#).
- [MINGW - "Minimalist GNU for Windows"](#). Another GCC version for Windows
- [DJ Delorie's C++ development system for DOS/Windows \(GNU C++\)](#)
- [GNU CC source](#)
- [IBM C++](#) for IBM power, System Z, Bluegene, and Cell.
- [Intel C++ for non-commercial development](#)
- [Microsoft Visual C++ Express edition](#).
- [Oracle C++](#).



Clang

From Wikipedia, the free encyclopedia

This article is about the compiler. For the phenomenon of rhyming word association, see [Clanging](#).

Clang /ˈklæŋ/^[4] is a [compiler front end](#) for the [programming languages C, C++, Objective-C, Objective-C++, OpenMP,](#)^[5] [OpenCL](#), and [CUDA](#). It uses the [LLVM](#) compiler infrastructure as its [back end](#) and has been part of the LLVM [release cycle](#) since LLVM 2.6.

It is designed to act as a drop-in replacement for the [GNU Compiler Collection](#) (GCC), supporting most of its compilation flags and unofficial language extensions.^{[6][7]} Its contributors include [Apple](#), [Microsoft](#), [Google](#), [ARM](#), [Sony](#), [Intel](#) and [Advanced Micro Devices](#) (AMD). It is [open-source software](#),^[8] with [source code](#) released under the [University of Illinois/NCSA License](#), a [permissive free software licence](#).

The Clang project includes the Clang [front end](#) and the Clang [static analyzer](#) and several code analysis tools.^[9]

Clang



Original author(s) [Chris Lattner](#) and others

Developer(s) [Apple Inc.](#) and others