Per non avere problemi nell'operatore di stampa per i template, sempre dichiarato come *friend*, si seguano tre indicazioni.

1) Forward declaration della classe di interesse nel template<T>
2) Dichiarazione sotto il template T dell'operatore di stampa
3) Creazione della classe, indicazione dell'operatore di stampa come friend con typename specificato a vuoto (per evitare che il compilatore dia come errore *friend declaration declares a non template function.* Esempio sotto:

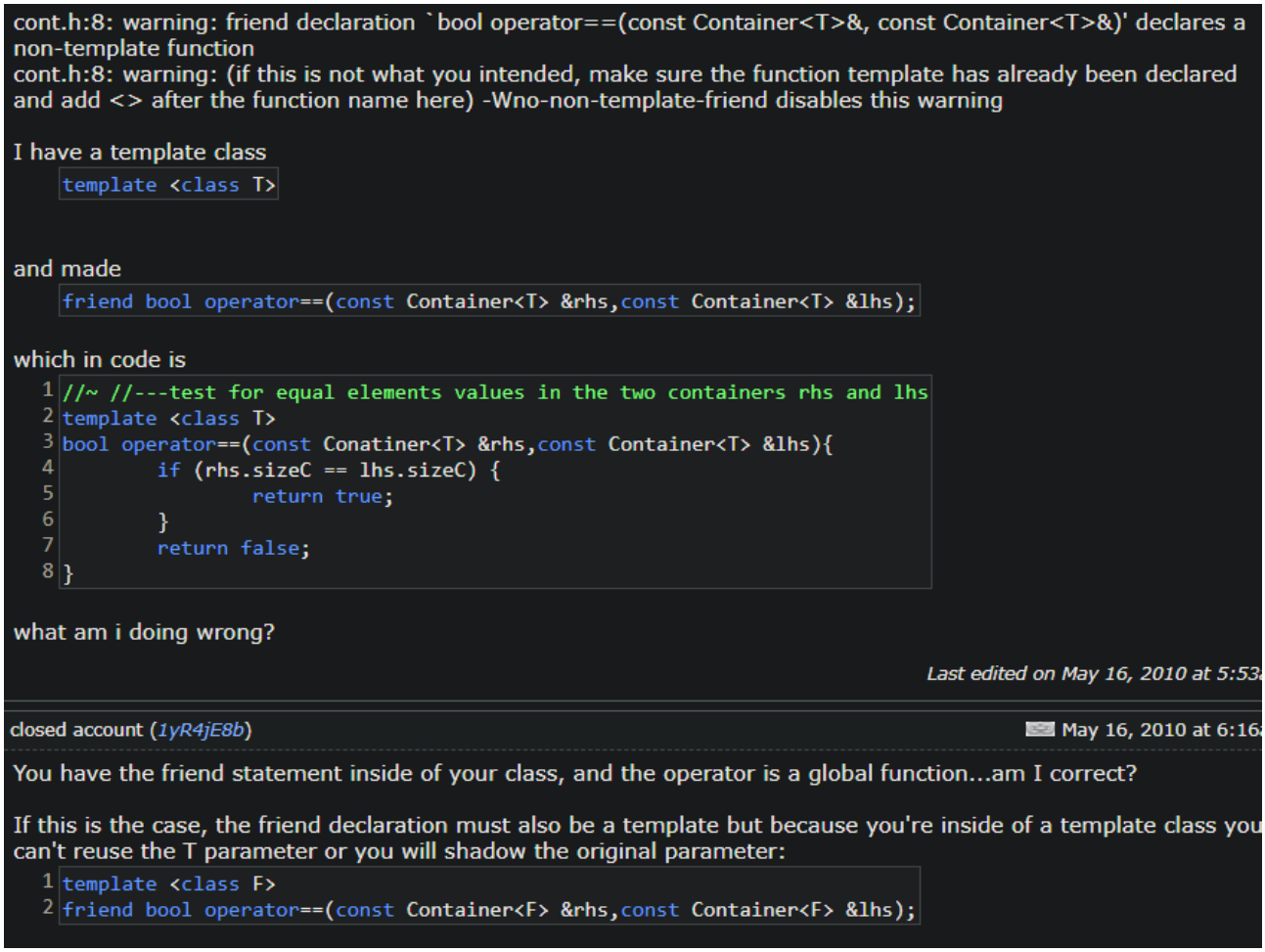
```
template <class T>
class Array;

template <class T>
ostream& operator<<(ostream&, const Array<T>&);

template <class T>
class Array{
    friend ostream& operator<< <>(ostream, const Array<T>&);
```

Un esempio subito utile e identico tra amicizie di template:

You can achieve this by forward declaring both the class and the function.

```
#include <iostream>

template<typename T>
class  cl;

template<typename T>
void non_template_friend(cl<T> m);

template<typename T>
class  cl
{
private :
    T val;
public:
    cl()= default;
    explicit cl(T v) : val(std::move(v)) {}

    friend void non_template_friend<T>(cl m); //Now we can refer t
};

template <typename T>
void non_template_friend(cl<T> m)  { std::cout << m.val << std::en


int main()
{

    cl<int> c(10);
    non_template_friend(c);
    return 0;
}
```

Alcuni esempi in merito all'errore prima:



cont.h:8: warning: friend declaration `bool operator==(const Container<T>&, const Container<T>&)' declares a non-template function
cont.h:8: warning: (if this is not what you intended, make sure the function template has already been declared and add <> after the function name here) -Wno-non-template-friend disables this warning

I have a template class

```
template <class T>
```

and made

```
friend bool operator==(const Container<T> &rhs,const Container<T> &lhs);
```

which in code is

```
1 //~ //---test for equal elements values in the two containers rhs and lhs
2 template <class T>
3 bool operator==(const Conatiner<T> &rhs,const Container<T> &lhs){
4         if (rhs.sizeC == lhs.sizeC) {
5                 return true;
6         }
7         return false;
8 }
```

what am i doing wrong?

Last edited on May 16, 2010 at 5:53

closed account (*1yR4jE8b*) May 16, 2010 at 6:16

You have the friend statement inside of your class, and the operator is a global function...am I correct?

If this is the case, the friend declaration must also be a template but because you're inside of a template class you can't reuse the T parameter or you will shadow the original parameter:

```
1 template <class F>
2 friend bool operator==(const Container<F> &rhs,const Container<F> &lhs);
```

In questo caso, essendo già dentro alla classe template, il parametro T viene oscurato (*shadowed*) da quello già presente, pertanto per fare in modo si affermi che è una funzione parte del template ma funzionante con altri parametri si deve ridichiarare l'operatore di uguaglianza e affermare sia friend.

In particolare, citando la documentazione Oracle, "i template devono essere dichiarati prima di essere usati. Una friend declaration costituisce un uso del template e non una dichiarazione". Quindi bisogna dare prima una definizione della classe di uso, una per il metodo che prende i generici parametri di una determinata classe friend ma all'interno dello stesso template, specificando typename vuoto per indicare quali parametri sta effettivamente prendendo.

Ad esempio:

```
array.h
// generates undefined error for the operator<< function
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
        operator<<(std::ostream&, const array<T>&);
};
#endif

array.cc
#include <stdlib.h>
#include <iostream>

template<class T> array<T>::array() {size = 1024;}

template<class T>
std::ostream&
operator<<(std::ostream& out, const array<T>& rhs)
    {return out <<'[' << rhs.size <<']';}

main.cc
#include <iostream>
#include "array.h"

int main()
{
    std::cout
      << "creating an array of int... " << std::flush;
    array<int> foo;
    std::cout << "done\n";
    std::cout << foo << std::endl;
    return 0;
}
```

Note that there is no error message during compilation because the compiler reads the following as the declaration of a normal function that is a `friend` of the `array` class.

```
friend ostream& operator<<(ostream&, const array<T>&);
```

Because `operator<<` is really a template function, you need to supply a template declaration for prior to the declaration of `template class array`. However, because `operator<<` has a parameter of `type array<T>`, you m precede the function declaration with a declaration of `array<T>`. The file `array.h` must look like this:

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

// the next two lines declare operator<< as a template function
template<class T> class array;
template<class T>
    std::ostream& operator<<(std::ostream&, const array<T>&);

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
      operator<< <T> (std::ostream&, const array<T>&);
};
#endif
```

Vediamo un ulteriore esempio:

```
base.h:24: warning: friend declaration ‘std::ostream& operator<<(std::ostream&, Base<T>*)’
base.h:24: warning: (if this is not what you intended, make sure the function template has
```

I've tried adding <> after << in the class declaration / prototype. However, then I get it `does not match any template declaration`. I've been attempting to have the operator definition fully templated (which I want), but I've only been able to get it to work with the following code, with the operator manually instantiated.

base.h

```
template <typename T>
class Base {
  public:
    friend ostream& operator << (ostream &out, Base<T> *e);
};
```

base.cpp

```
ostream& operator<< (ostream &out, Base<int> *e) {
    out << e->data;
return out;
}
```

I want to just have this or similar in the header, base.h:

```
template <typename T>
class Base {
  public:
    friend ostream& operator << (ostream &out, Base<T> *e);
};

template <typename T>
ostream& operator<< (ostream &out, Base<T> *e) {
    out << e->data;
return out;
}
```

I've read elsewhere online that putting <> between << and () in the prototype should fix this, but it doesn't. Can I get this into a single function template?

La risposta corretta è:

It sounds like you want to change:

```
friend ostream& operator << (ostream& out, const Base<T>& e);
```

To:

```
template<class T>
friend ostream& operator << (ostream& out, const Base<T>& e);
```

Per avere la funzione correttamente templatizzata, si dovrebbe avere una dichiarazione apposita della classe di riferimento, non essendo la stessa classe ma un'altra

```
template <typename T>
class Base {
  public:
    template<class U> friend ostream& operator << (ostream &out, Base<U> const &e){
       return out;
    };
};

int main(){
   Base<int> b;
   cout << b;
}
```

In alcuni casi, è comunque conveniente per evitare questa pesantezza del C++, evitare di dichiarare friend una funzione oppure un pezzo di codice solo per usarlo in maniera templatizzata.
Per esempio:

I have a problem to overload the `<<` stream operator and I don't find the solution :

```
template<class T, unsigned int TN>
class NVector
{
    inline friend std::ostream& operator<< (
        std::ostream &lhs, const NVector<T, TN> &rhs);
};

template<class T, unsigned int TN>
inline std::ostream& NVector<T, TN>::operator<<(
    std::ostream &lhs, const NVector<T, TN> &rhs)
{
    /* SOMETHING */
    return lhs;
};
```

It produces the following error message:

> warning : friend declaration 'std::ostream& operator<<(std::ostream&, const NVector&)' declares a non-template function [-Wnon-template-friend]
>
> error: 'std::ostream& NVector::operator<<(std::ostream&, const NVector&)' must take exactly one argument

How to solve that problem ?

E la soluzione riflette due problematiche; la seconda più evidente, essendo funzione esterna, non posso mettere il tipo Nvector, ma in particolare, definire la funzione friend dell'operatore direttamente dentro alla classe, scriverla lì e fine oppure fare come sopra.
In particolare:

There are two different issues in your code, the first is that the `friend` declaration (as the warning clearly says, maybe not so clear to understand) declares a single non-templated function as a friend. That is, when you instantiate the template `NVector<int,5>` it declares a non-templated function `std::ostream& operator<<(std::ostream&,NVector<int,5>)` as a friend. Note that this is different from declaring the template function that you provided as a friend.

I would recommend that you define the friend function inside the class definition. You can read more on this in this answer.

```
template <typename T, unsigned int TN>
class NVector {
   friend std::ostream& operator<<( std::ostream& o, NVector const
      // code goes here
      return o;
   }
};
```

Alternatively you can opt for other options:

1. declare the `operator<<` template as a friend (will grant access to any and all instantiations of the template),
2. declare a particular instantiation of that template as a friend (more cumbersome to write) or
3. avoid friendship altogether providing a public `print( std::ostream& )` member function and calling it from a non-friend templated `operator<<`. I would still opt to befriend the non-template function an provide the definition inside the templated class.

The second issue is that when you want to define an operator outside of the class of the left hand side argument, the operator is a *free function* (not bound to a class) and thus it should not be qualified:

```
template<class T, unsigned int TN>
inline std::ostream& operator<<(std::ostream &lhs, const NVector<T
{
    /* SOMETHING */
    return lhs;
};
```