

nir-operators are infix relational operators (<, >, =) of n operands that make it possible to write things like if (a<b<c) {...}. Here we examine a C++ implementation of nir-operators.

Nir-operators in C++

Nicola Di Nisio

The paper shows various techniques to improve the use of relational operators in C++ programs, by using smart pointer, template and virtual functions.

A certain knowledge of the C++ language is required.

Relational operators are functions which make a tie between two or more expressions. The best known are the binary ones, that take two operands. For the we usually adopt an infix notation: the operands are placed one at the left and the other at the right side of the symbol of the operator. In C/C++ these operators are <, <=, >, >=, == and !=.

From here on we will use the term *nir-operator* to mean “n-ary infix relational operator”.

In the ANSI/ISO C++ the *bool* type has been introduced and this should be the result type of any relational operator (see [doc96]). Whichever is the type of the relational operator an implicit conversion to *bool* is made, while in the AT&T version of the C++ the type-casting is toward *int*.

In any case it is possible to overload the infix relational operator to make it return any data type and this is the possibility that we will exploit to make a binary relational operator n-ary. There are at least two techniques that we can use for this aim: the first makes use of inheritance and virtual functions and for this it is intrusive toward the classes we want to endow with nir-operators; the second, instead, makes use of *smart pointers* and does not force to modify our classes.

The strategy

The heart of our approach to the implementation of nir-operators is the following question: for the relational operator it is possible to return the value of its right hand operand together to the truth value of the comparison made at its left. This truth value may be immediately considered as the truth value of the expression, or it may be considered as an intermediate result for subsequent comparisons. This last event happens in expressions of the type $a_1 < a_2 < \dots < a_n$ in which the result is *false* even if only one comparison fails.

Let us suppose that the operands, of type T, had, among the others, an attribute *truthValue* which default value 1. Let us consider the expression $a < b < c$. If $a < b$ returns *b* with *b.truthValue* equal to the truth value of the comparison $a < b$, then after a first step the expression is equal to

```
(b.truthValue) && (b<c)
```

At the second step on $b < c$ we have

```
(b.truthValue) && (c.truthValue)
```

At the end $a < b < c$ returns *c.truthValue*. It is therefore necessary to give an *operator bool()* for the ANSI/ISO C++ and an *operator int()* for the AT&T's one, which returns only the attribute *truthValue*. Indeed, this is the conversion operator called by the compiler for any relational expression.

The intrusive approach

Let us consider the *operator<()* and a class T that we want to endow with the nir-operator<. The first thing to do is to provide a base class *NiropBase*, which provide the attribute *truthValue*. From this class we will derive the classes with nir-operators, with the constraint that these classes do not implement any infix relational operator, indeed those operator should be implemented by *NiropBase*.

Here follows the implementation of the base class:

```
class NiropBase {
public:
    NiropBase() {
        truthValue=1;
        isShallowCopy=0;
    }

    #if __cplusplus >= 199711L || (defined(__BCPLUSPLUS__) && __BCPLUSPLUS__>=0x520)
    operator bool() {
        int tmp=truthValue;
        if (isShallowCopy) {
            isShallowCopy=0;
        }
    }
    #endif
};
```

```

        truthValue=1;
    }
    return tmp;
}
#else
// I compilatori non ANSI/ISO
// non hanno l'operator bool()
operator int() {
    int tmp=truthValue;
    if (isShallowCopy) {
        isShallowCopy=0;
        truthValue=1;
    }
    return tmp;
}
#endif

friend NiropBase& operator< (NiropBase &op1, NiropBase &op2) {
    op2.isShallowCopy=1;
    if (!op1.truthValue)
        op2.truthValue=0;
    else
        op2.truthValue=(op1.isLessThan(op2));
    if (op1.isShallowCopy) {
        op1.isShallowCopy=0;
        op1.truthValue=1;
    }
    return op2;
}

private:
    int truthValue;
    int isShallowCopy;
    virtual int isLessThan(const NiropBase &op) const=0;
    virtual int isEqualTo(const NiropBase &op) const=0;
};

```

This class provides two pure virtual functions (*isLessThan* and *isEqualTo*), to compute all the relational operators implemented by its derived classes (see later for more detailed considerations). As aforesaid, *truthValue* is initialised to 1 (i.e. *true*), because this is the neutral value with respect to the logical operator AND, which chains the singleton boolean expressions present in expressions like $a < b < c$, which in truth is $(a < b) \text{ AND } (b < c)$.

Another public member is *operator<*, which propagates possible *false* truth values, or evaluates the comparison $op1 < op2$ by using *isLessThan()*. The *operatorX* (where X may be equal to $<$, $<=$, $>$, $>=$, $==$, $!=$) must all be implemented in the base class *NiropBase*. In the example provided, for brevity, there is only the *operator<*.

Among the private members there are *truthValue* and *isShallowCopy*, which is needed to count the number of shallow copies made for each operand. These shallow copies are destroyed either by the relational operators or by the operator *bool()*. Whenever *isShallowCopy* reaches 0, *truthValue* is set to 1.

Now follows the implementation of a string class, derived from *NiropBase* to make its relational operator *nir-operators*:

```

#include <string.h>
#include "NiropBas.h"

class StringDerived: public NiropBase {
public:
    StringDerived(const char *s1=0): NiropBase() {
        s=new char[80];
        if (s1)
            strcpy(s,s1);
        else
            s[0]='\0';
    }

    ~StringDerived() {delete[] s;}

    operator char*() {return s;}

private:
    char *s;
    int isLessThan(const NiropBase &s2) const {

```

```

        return strcmp(s, ((StringDerived*) &s2)->s)<0;
    }
    int isEqualTo(const NirropBase &s2) const {
        return strcmp(s, ((StringDerived*) &s2)->s)==0;
    }
};

```

By using this class it is now possible to write something like the following code:

```

StringDerived a("aa"), b("bb"), c("cc");
if( a<b<c )
    cout<<"it runs!"<<endl;

```

In closing, the steps to endow a class T of nir-operators are:

1. Derive T from NirropBase
2. Remove from T the implementation any infix relational operator
3. Endow T with the private member functions *isLessThan()* and *isEqualTo()*.

This approach to nir-operators, forces us to modify the source code of our class T and this is not always possible, for example we do not have the source code of T. In these cases we can adopt a *smart* approach.

The smart approach

Another solution to the problem of nir-operator is based on *smart pointers*. A smart pointer is an object of a class parameterised on a class T, which gives the illusion to be a pointer to an object of type T. The illusion is obtained by overloading *operator*()* and *operator->()*, by which we retrieve a pointer to the *pointed* object.

To add the nir-operators, the first step is to prepare a template which has all we need to realise a smart pointer, that is constructors from T& and T* and the overloaded functions *operator*()* and *operator->()*. To this standard equipment we must add the attribute *truthValue*, to keep track of the truth values of the comparisons, the *operator bool()*, to return *truthValue* to who tries to interpret the smart pointer as a boolean value, and the *operatorX()* (see above). These last will consider the truth value of the left side operand and will call the right *operatorX()* of the derived class T, returning a copy of the right side smart pointer operand, with the right truth value. Here follows such a template:

```

template <class T> class Nirrop {
public:
    Nirrop(T* _ptr=0): ptr(_ptr) {
        truthValue=1;
        isShallowCopy=0;
    }

    Nirrop(T& _obj): ptr((T*)&_obj) {
        truthValue=1;
        isShallowCopy=0;
    }

#ifdef __cplusplus >= 199711L || (defined(__BCPLUSPLUS__) && __BCPLUSPLUS__>=0x520)
    operator bool() {
        int tmp=truthValue;
        if (isShallowCopy) {
            isShallowCopy=0;
            truthValue=1;
        }
        return tmp;
    }
#else
    // I compilatori non ANSI/ISO
    // non hanno l'operator bool()
    operator int() {
        int tmp=truthValue;
        if (isShallowCopy) {
            isShallowCopy=0;
            truthValue=1;
        }
        return tmp;
    }
#endif
};

```

```

void operator=(T* _p) {ptr=_p;}
void operator=(T& _obj) {ptr=((T*)&_obj);}

T& operator*() {return *ptr;}
T* operator->() {return ptr;}

operator T() {return *ptr;}

Nirop<T>& operator<(Nirop<T> &op2) {
    op2.isShallowCopy=1;
    if (truthValue==0)
        op2.truthValue=0;
    else
        op2.truthValue=(*ptr<>(*op2.ptr));
    if (isShallowCopy) {
        isShallowCopy=0;
        truthValue=1;
    }//if
    return op2;
};

private:
    T* ptr;
    int truthValue;
    int isShallowCopy;
};

```

This template has got all you need to endow a generic class T with the *nir-operator*<. The most relevant fact is that the class T does not have anything in particular to do, but implementing the *operator*<(). Depending on the implementation of the *operatorX*() in the template class we should be required to furnish T with all the relational operators or only one of them which is logically complete, like *operator*<=>() (see next paragraph for more detailed considerations).

Here follows a string class which does not have any attribute or method useless to its strict purposes of string class:

```

#include <string.h>

class MyString {
public:
    MyString(const char* s1=0) {
        s=new char[80];
        if (s1)
            strcpy(s,s1);
        else
            s[0]='\0';
    }

    ~MyString() {delete[] s;}

    operator char*() {return s;}

    int operator< (MyString &s2) const {
        return strcmp(s,s2) <0;
    }
private:
    char *s;
};

```

Now it is possible to write a piece of code like this:

```

MyString aa("aa"),bb("bb"),cc("cc");
Nirop<MyString> a(aa),b(bb),c(cc);
if (a<b<c)
    cout<<"funziona!"<<endl;

```

The presence of the constructor from T& and of the assignment operator from T& (together to that from T*) enables us to use Nirop<T> even as a object and this will be indeed the most frequent case.

Which operators use in the template or in the base class?

In the this last approach we have noted that the template class could require the implementation of all the infix relational operators in the class T. This is our case no. 1 and is indeed the most efficient.

On the other hand, in the intrusive approach we have seen a base class that requires the implementation of only two functions in its derived classes *isLessThan()* and *isEqualTo()*, which defines a total order in the class T. This is our case no. 2.

The last extreme possibility, valid in both the approaches, is to require the implementation of only one relational operator which is logically complete, like *operator<=()*. This is our case no. 3.

Well, these possibilities are in a decreasing order of both complexity and computational and spatial efficiency. In fact in the first case the author of the class T must realise all the infix relational operator and this for every class T, while the average number of comparisons made for a general relational computation is 1, in that for every comparison only one infix relational operator is called.

In the second case only two relational operators/functions must be realised in the class T, but the average number of infix relational operations made for every relational operation is:

Expression	Number of computations
<code>a==b</code>	1
<code>a!=b</code> \Leftrightarrow <code>!(a==b)</code>	1
<code>a<b</code>	1
<code>a<=b</code> \Leftrightarrow <code>(a<b)&&(a==b)</code>	2
<code>a>b</code> \Leftrightarrow <code>!(a<b a==b)</code>	2
<code>a>=b</code> \Leftrightarrow <code>!(a<b)</code>	1
Average	1.33333

We have neglected the cost of simple boolean computations, because they may be very small in comparison to the costs of infix relational operators for a generic class T.

The last case is the most simple for the generic author of the class T, but the average number of infix relational operations made for every relational operation increases:

Expression	Number of computations
<code>a==b</code> \Leftrightarrow <code>a<=b && b<=a</code>	2
<code>a!=b</code> \Leftrightarrow <code>!(a<=b && b<=a)</code>	2
<code>a<b</code> \Leftrightarrow <code>a<=b && !(a<=b && b<=a)</code>	3
<code>a<=b</code>	1
<code>a>b</code> \Leftrightarrow <code>!a<=b</code>	1
<code>a>=b</code> \Leftrightarrow <code>(a<=b && b<=a) !a<=b</code>	3
Average	2

The second case is undoubtedly the most convenient for most classes.

Smart Object

The overloading of the operator “.” is explicitly denied (see [doc96]), because otherwise it could become impossible to access the attributes of the object. I would have preferred to be able to re-define the *operator.()* with a special semantic: if *p* is of type `SmartPtr<T>` and *attr* is an attribute of *p*, then *p.attr* returns back the value of the attribute *attr* of *p*, otherwise we execute the *operator.()* of the class `SmartPtr<T>`.

If this all was it possible, then a “Smart Object” could have been implemented, instead of a smart pointer. On this point a strong difference exists between the C++ and Smalltalk. This last is all *late binding*. The messages are sent to the objects, which interpret them at run-time and only at this time we can be sure that all our code is good.

The C++, instead, has an *early binding* nature: we do not send messages to the objects, but only recall some methods of theirs and all it is known at compile time. This does not mean that the C++ does not offers a late binding mechanism, but only that it seem not be in the nature of the language.

Differences among compilers

To write something like

```
Nirop<MyStringa> a(MyStringa("aa"));
```

does not warrant that at the following row *a* contains something useful: the temporary string “aa” could have been destroyed and it is exactly what happens in ANSI/ISO compilers. The Turbo C++ 3.0, which is not ANSI/ISO, accept without any problem the declaration of *a* initialised with a temporary, while in

the C++ Builder 1.0, which is ANSI/ISO compliant, at the following line of code *a* points to garbage and the compiler does not give any warning on this fact.

The on-line manual of the Visual Age C++ 3.0 for OS/2 (compiler not fully ANSI/ISO compliant, in the sense that it does not respect [doc96], but an older version, the X3J16/92-0060 of Sep/17/92) says explicitly that the temporary survive as long as the block in which they have been created survive. Follows that temporaries are not destroyed immediately after their evaluation (this can lead to inefficiency on some cases).

The declaration of temporary strings to be assigned to some `Nirop<MyString>` could be tiresome, because it doubles the amount of code to write for the declaration and the initialisation of variables. In the intrusive approach this problems does not appear, hence any existing program can be left without any change in its code, except for the fact to derive any class *T* to be endowed with nir-operators from the class `NiropBase`.

More on smart pointers

Another problem of the smart approach is that smart pointers limit the access to methods of objects pointed by *ptr*.

Let us suppose to have defined in the class `MyString` two overloaded versions of the `operator<()>`, one for `MyString` and the other for `char*`. If *p* is of type `Nirop<MyString>`, then the expression `p<<"aa"` cannot be compiled, because `Nirop<MyString>` has defined only the operator

```
operator<(Nirop<T>& _obj)
```

with `T=MyString`. It is only for this operator that it is recalled the `operator<(MyString& s)>` of the class `MyString`. The ideal should be to be able to use template member functions, but they are not yet implemented by the most common C++ compilers. For example, adding an `operator<()>` like the following

```
template <class Q>
Nirop<T> operator< (Q& _obj) {...}
```

we can solve this problem, because the compiler generates at run-time a different instance of the operator `operator<(Q&)>` for each type *Q* different from *T* used in every right-hand side of expressions like

```
aNiropString < anExpOfTypeQ
```

A compromise solution is that of specifying all the possible instances of `operator<(...)>` we could need, but it is clear that this limit the universality of the solution.

Another alternative is to use the notation

```
(*aNiropString) < anExpOfTypeQ
```

that is, unfortunately, usable only in a binary fashion, because in this way we overcome the relational operators in `Nirop<T>`.

Operators' precedence

The need which nir-operators want to satisfy is that of being able to write relational expressions like human beings do in the mathematical language, in which all the operators do have the same priority. A chain like the following

```
a==b < c < d==e
```

is interpreted with the only left associativity. In C++, instead, `operator<()>` precedes `operator==()>`, hence the same expression should be rewritten in this way

```
(a==b) < c < (d==e)
```

to be interpreted in the same way of the preceding.

Let us consider the case in which `a="bb"`, `b="bb"`, `c="cc"`, `d="dd"` and `e="dd"`. In the expression

```
a==b<c<d==e
```

the first subexpression interpreted is `b<c<d`, which returns the string "dd" with truth value *true*. At this point it is like to evaluate the expression `a==d==e`. For the left associativity of `operator==()>`, the

subexpression `a==d` it is first evaluated, that is `"bb"=="dd"`, which is false and makes the whole expression false. In the mathematical language, instead, the expression

```
"bb"=="bb"<"cc"<"dd"=="dd"
```

is true. As the priority of operators is resolved by the parser, the only way to remedy is to use parenthesis.

In C++, the operators `<`, `<=`, `>`, `>=` precede the operators `==` e `!=`. The critical case are that in which there are operators of different priority. The operator `&&` and `||` (And and Or) have inferior priority with respect the others mentioned and `&&` precedes `||`. This all implies that it is possible to write things like `a<b<c || a<c<b` meaning `(a<b<c) || (a<c<b)`, even if in this case the parenthesis increase the readability of the expression.

The case `T==int`

We have provided two constructors in `Nirop<T>`, one from pointer to T and the other from instances of T. For symmetry it would be appreciated to get a similar mechanism in reading, that is a conversion function to T, like the following one:

```
operator T() {return *ptr;}
```

In only one circumstance problems show up: on AT&T compliant compilers each time that `T==int`. The reason is that the instantiation of `operator T()` will override that of `operator int()` which already exists to address problems others than that of the type conversion.

This problem can be overcome by the *template specialisation*, that is explicitly treating the case `T==int`. But this solution would not solve another problem: in fact by using the old `operator int()` logic, every conversion of a `Nirop<int>` to an `int` could lead to an integer equal to 1, which is the default truth value of every `Nirop<T>`. We could then decide to return the number itself in place of 1, indeed 5 means *true* as 1 does, but what if the number was 0? The consequence is that the case `T==int` should be avoided in C++ compilers not ANSI/ISO compliant.

Another problem lead by the conversion operator to T shows up whit the C++ Builder 1.0 compiler (which has the same engine of the Borland C++ 5.02). In fact this compilers not able to choose between `operator bool()` and `operator T()` whenever T is a basic data type (int, float, double, ...) or even `void*` or `char*`! The problem can be solved in two ways: the first is to remove the `operator T()`, the other is to force the compiler to use the `operator bool()`:

```
if ((bool) (a<b<c)) {...}
```

Comparison between the two approaches

The advantages here seen for the intrusive approach are:

1. The class T must be derived from `NiropBase` and this is not possible if we do not have the source code of T
2. The class T must have no binary infix relational operator and this leads some maintenance work if the class T already exists and has these operators
3. The class T needs of the methods `isLessThan()` and `isEqualTo()` (or an equivalent set of functions), whose code can be retrieved by the operators removed at the point no. 2

The advantages of the smart approach here seen are:

1. It is not possible to initialise a `Nirop<T>` with a temporary of type T: it is always needed a permanent copy in the scope of interest. This implies that we are required to write more lines of code and the existing code should be modified in a more heavy way (we do not touch the class T, but we are forced to change and check all the code that uses it)
2. Some methods in T may be hidden by the smart pointer, unless we force the casting.
3. Sometime we are forced to deal with the syntax of pointers and this could result in changings in the existing code, together to a decreased readability of the source code.

Depending on our work conditions one or the other approach could be preferred, but the author believes that, above all in great projects, it is more believable that the intrusive approach give the best results in terms of the minimisation of overall number of lines of code changed.

It is true that on a brand new project the music is another and the weigh bridge could begin to hang down toward the smart approach, even because another advantage of his own, not yet cited, is its

speed, thanks to the fact that no virtual methods calls are used (this is the general advantage of using templates instead of inheritance).

Here follows a table which shows timings reported on a Pentium 133 with 16 Megs of RAM by a test with 10 millions if computation if the expression $a < b < c$, with $a = \text{"aa"}$, $b = \text{"bb"}$ and $c = \text{"cc"}$.

The reported timing are to be considered under a qualitative prospective, even because only the C++ Builder 1.0 and the Visual Age 3.0 compilers have produced Pentium code and hence are comparable, while the g++ has produced 486 code and the Turbo C++ 3.0 has produce 286 code.

Anyway, apart from the absolute timings of each version of the test, what matters is that the smart version is always faster than the intrusive one. The last column reports the percentage of difference in speed between the two approach on the same platform.

Operating system	Compiler	ANSI/ISO	Intrusive	Smart	+speed
OS/2 Warp 4	Visual Age 3.0	No	5.4	3.5	54 %
Windows 95	C++ Builder 1.0	Yes	6.5	4.0	62 %
Linux 2.0.27	g++ 2.7.2.1-2	Yes	8.8	6.2	41 %
MS-DOS	Turbo C++ 3.0	No	14.4	13.0	11 %

Resume

The nir-operators help us to write more readable code. This is particularly true in situations where it is frequent the verification of the appartenance to an interval. Instead of writing

```
if (date1<date2) &&  
    (date2<date3) &&  
    (date3<date4)
```

we can write

```
if (date1<date2<date3<date4)
```

The two techniques shown are two very different approaches to the problem: the first uses the heredity to endow the derived classes with nir-operators and virtual functions to physically perform the comparison between instances, while the second delegates (*by proxy...*) all the responsibility to a template parameterised over the class T to endow with nir-operators. This second technique follows the Proxy pattern [Irs97] and confirmates the quality of this pattern, notwithstanding the limitations that we have encountered.

Bibliography

[doc96] Doc No:X3J16/96-0225 WG21/N1043, Dec 2nd, 1996. Proposal of standardisation - C++ ANSI/ISO, Draft No. 2.

[Irs97] Graziano Lo Russo - "*Pattern chiari, amicizia lunga*", Computer Programming, May 1997

Nicola Di Nisio has a B.Sc. in Computer Science, interests in cryptography, neural networks and bayesian analysis. works as a software developer.

email: nicola@dinisio.net

HP: www.dinisio.net/nicola