



Milano 26 Maggio

# Inprise/Borland Forum 2000

Sviluppi su Delphi, Java/JBuilder, C++/C++ Builder

GRUPPO  
EDITORIALE  
INFOMEDIA

con la partecipazione di:





# C++ Builder Aderenza allo Standard e Qualità del Codice

**Carlo Pescio**

**([pescio@eptacom.net](mailto:pescio@eptacom.net))**

**Eptacom Consulting & Editorial Board Computer Programming.**

**Prerequisiti: programmazione C++**

**Livello: intermedio**

# Core Language

E' utile parlare di aderenza allo standard?

Molti utilizzatori di ambienti RAD sono, naturalmente, interessati alle facility di programmazione rapida/visuale.

Il codice scritto con strumenti RAD, per sfruttare a fondo le potenzialita' del tool, e' di norma non portabile.

Ha quindi senso parlare di aderenza allo standard?

Ha senso valutare la qualita' del codice generato?

A mio avviso, si.

Chi utilizza il C++ e' di solito molto attento all'efficienza del codice. Inoltre, ogni programma non banale ha un nucleo di codice distinto dalla GUI, che puo' essere molto utile mantenere portabile su altri compilatori / piattaforme.

# Core Language

## Cosa vedremo

Lo standard ANSI/ISO ha arricchito il C++ con parecchie funzionalità, sia semplici che avanzate.

Ho preso in esame solo le estensioni più significative e recenti. Non ho verificato l'aderenza allo standard sulle parti più consolidate del linguaggio.

Ho preso in esame principalmente il core language e non l'implementazione della libreria standard, salvo alcuni casi (razionale: almeno la libreria è sostituibile).

Ho esaminato sia l'esistenza e la correttezza delle nuove funzionalità che la presenza di alcune importanti macro ottimizzazioni.

Sono comunque test non esaustivi!

# Aderenza allo Standard

(C) Carlo Pescio

# Are Verificate

4 settori principali:

1) Minuzie :-) (bool, explicit, typename, ecc)

2) Eccezioni

3) Namespace

4) Template

Nei casi piu' complessi (es.Koenig Lookup) ho eseguito solo una verifica parziale (le situazioni piu' comuni, piu' alcune situazioni convolute).

# bool

Risultato: OK

Il compilatore riconosce bool, true, false e si comporta nel modo corretto nel loro utilizzo.

Mini esempio:

```
void check_bool()
{
    bool b1 = true ;
    bool b2 = false ;
    bool b3 = b1 == b2 ;
    cout << b1 << " " << b2 << " " << b3 << "\n" ;
}
```

bool e' un tipo distinto dagli altri (possibile overloading).

# explicit

Risultato: OK (messaggio di errore non ottimale)

Il compilatore riconosce i costruttori explicit e non li utilizza per effettuare conversioni implicite.

Mini esempio:

```
struct C
{
    explicit C( int )
    {
    }
} ;

void f( const C& )
{
}

void check_explicit()
{
    // must compile
    C c( 3 ) ;
    // must compile
    f( c ) ;
    // must compile
    f( static_cast< C >( 3 ) ) ;
    // must not compile
    f( 3 ) ;
}
```

il messaggio di errore dice “costruttore non trovato”...

# Scope delle variabili nel for

Risultato: OK

Il compilatore restringe lo scope delle variabili dichiarate nella condizione del ciclo for al solo body del ciclo.

Mini esempio:

```
void check_for_scope()  
{  
    int i = 0 ;  
    for( int i = 0; i < 10; ++i )  
        ;  
    cout << "expecting 0; result: " << i << "\n" ;  
}
```

Si puo' disabilitare tramite switch di compilazione (legacy).

# Dichiarazioni dentro if

Risultato: OK

Il compilatore accetta dichiarazioni dentro la condizione di un if, e lo scope e' correttamente limitato al body.

Mini esempio:

```
int f42()
{
    return( 42 ) ;
}

void check_if_declaration()
{
    if( int i = f42() )
    {
        cout << "expecting 42; result: " << i << "\n" ;
    }
}
```

# Covarianza su return type

Risultato: OK

Il compilatore consente di variare il tipo del risultato di una funzione virtuale in modo covariante.

Mini esempio:

```
struct Base
{
    virtual Base* f() ;
    virtual void f1() ;
} ;

struct Derived : public Base
{
    virtual Derived* f() ;
    virtual void f1() ;
} ;

void check_covariant_return()
{
    Derived d ;
    Derived* p = &d ;
    Derived* d1 = p->f() ;
    d1->f1() ;
}
```

# new solleva bad\_alloc

Risultato: OK

Il compilatore, correttamente, solleva l'eccezione bad\_alloc se l'operatore new fallisce (anzichè restituire NULL).

Mini esempio:

```
void check_new_throws()
{
  try
  {
    int* i = new int[ 2000000000L ] ;
    cout << "no exception!\n" ;
  }
  catch( std::bad_alloc& )
  {
    cout << "bad_alloc caught!\n" ;
  }
  catch( ... )
  {
    cout << "some exception caught!\n" ;
  }
}
```

# Espressione void

Risultato: OK

Il compilatore consente di restituire un valore di tipo void in modo esplicito.

Mini esempio:

```
void check_void_return()  
{  
    return void() ;  
}
```

Si tratta di una funzionalità molto utile per avere una reale uniformità nelle istanze dei template.

# Costanti compile-time

Risultato: OK

E' possibile definire una costante statica direttamente nella dichiarazione di una classe. Diventa una costante compile-time.

Mini esempio:

```
struct C
{
    static const int Size = 5 ; // in header file
    char buf[ Size ] ;
} ;

const int C :: Size ; // In implementation file

void check_static_constant()
{
    C obj ;
    cout << "Expecting " << C::Size << ": " ;
    cout << sizeof( obj.buf ) << "\n" ;
}
```

# static initializers

Risultato: OK

La sintassi T(), dove T e' un tipo con costruttore "banale" (es. tipi built-in) e' ammessa ed inizializza a zero l'oggetto, come avverrebbe se l'oggetto stesso fosse statico.

Mini esempio:

```
void check_static_init()
{
    int x ;
    int y = int() ;
    cout << "can be any value:" << x << "\n" ;
    cout << "must be zero:" << y << "\n" ;
}
```

Nuovamente, e' molto utile per uniformita' nelle istanze dei template.

# auto\_ptr

## Risultato: OK

Il template standard `auto_ptr` e' implementato correttamente, in particolare cede il possesso azzerando il precedente proprietario. Inoltre supporta uno pseudo-polimorfismo nel costruttore di copia. Mini esempio:

```
struct Base
{
};

struct Derived : public Base
{
};

void check_auto_ptr()
{
    auto_ptr< Derived > d( new Derived ) ;
    auto_ptr< Base > b( d ) ;
    cout << "Expecting 0: " << d.get() << "\n" ;
    cout << "Expecting not 0: " << b.get() << "\n" ;
}
```

# uncaught\_exception

Risultato: FAIL

Il compilatore riconosce, ma non gestisce, la funzione standard `uncaught_exception`.

Mini esempio:

```
struct C
{
    ~C()
    {
        if( uncaught_exception() )
            cout << "U.E.\n" ;
        else
            cout << "N.U.E.\n" ;
            // further evidence:
            //      throw 3 ;
    }
};

void check_uncaught_exception()
{
    cout << "expecting U.E.\n" ;
    C x ;
    throw 1 ;
}
```

# Function try block

Risultato: FAIL

Il compilatore non riconosce la sintassi di un function try block ed emette un errore di compilazione.

Mini esempio:

```
struct Base
{
    Base( int )
    {
        throw 1 ;
    }
};

struct Derived : Base
{
    Derived() ;
};

Derived :: Derived()
try
: Base( 4 )
{
    // constructor body
    std::cout << "inside Derived" ;
}
catch( ... )
{
    std::cout << "exception caught\n" ;
}
```

# Rethrow

Risultato: OK

Il compilatore gestisce correttamente il re-throw di un oggetto eccezione.

Mini esempio (cattivo):

```
struct C
{
    C()
    {
        cout << "C\n" ;
    }
    C( const C& )
    {
        cout << "C(C&)\n" ;
    }
    ~C()
    {
        cout << "~C\n" ;
    }
};

void check_rethrow()
{
    try
    {
        throw C() ;
    }
    catch( ... )
    {
        f() ;
    }
}

void f()
{
    try
    {
        throw ;
    }
    catch( C& c )
    {
        cout <<
            "C caught\n" ;
    }
    catch( ... )
    {
        cout <<
            "other caught\n" ;
    }
}
```

# Run-Time Type Information

Risultato: OK

Il compilatore riconosce ed implementa le keyword relative al RTTI, come `dynamic_cast`.

Mini esempio:

```
class A
{
public :
virtual void f()
{}
} ;

class B
{
public :
virtual void g()
{}
} ;

class C : virtual public A,
          virtual public B
{
public :
virtual void f()
{ cout << "C::f()\n" ; }
virtual void g()
{ cout << "C::g()\n" ; }
} ;

void check_RTTI1()
{
C c ;
A* a = &c ;
B* b =
dynamic_cast< B* >( a ) ;
b->g() ; // calls C::g()
A* a2 =
dynamic_cast< A* >( b ) ;
a2->f() ; // calls C::f()
}
```

# namespace

Risultato: OK (test parziali!)

Il compilatore sembra gestire correttamente i namespace, le using declaration, le using directive, gli alias.

Mini esempio (cattivo!)

```
namespace A
{
    struct C
    {
        virtual void f()
        {
            cout << "A::C::f()\n" ;
        }
    } ;
}

void check_namespace()
{
    B::C c ;
    c.f() ;
}

namespace B
{
    struct C : A::C
    {
        virtual void f()
        {
            A::C::f() ;
            typedef A::C Parent ;
            Parent::f() ;
        }
    } ;
}
```

# Koenig Lookup

Risultato: OK (test parziali!)

Il compilatore sembra implementare correttamente la Koenig Lookup nella ricerca delle funzioni da chiamare.

Mini esempio:

```
namespace Test
{
    class A
    {
    } ;

    void f( const A& )
    {
        cout << "Test::f() called\n" ;
    }
}

void check_KoenigLookup1()
{
    Test::A a ;
    f( a ) ;
}
```

# typename

Risultato: OK (in modalita' ANSI)

Il compilatore accetta typename in luogo di class nei template, e richiede (in modalita' ANSI) l'uso di typename per i tipi dipendenti dai parametri dei template.

Mini esempio:

```
template< typename T > void f( const T& t )
{
    cout << t << "\n" ;
}

template< class T > class B
{
    public :
    // error
    typedef A<T>::aType bType ;
    // ok
    typedef typename A<T>::aType bType ;
    bType b ;
} ;

template< class T > class A
{
    public :
    typedef T aType ;
} ;
```

# Template member function

Risultato: OK

Il compilatore riconosce le template member function, anche in casi “cattivi” come quello riportato di seguito.

```
void check_template_member_function()
{
    int x = Get( cin ) ;
    double d = Get( cin ) ;
    std::string s = Get( cin ) ;

    cout << x << " " << d << " " << s ;
}

Get_Proxy Get( std::istream& s )
{
    return( Get_Proxy( s ) ) ;
}
```

continua...

# Template member function

continua...

```
class Get_Proxy
{
public :
    Get_Proxy( std::istream& s ) : is( s )
    {
    }
    template< class T > operator T()
    {
        T t ;
        is >> t ;
        return( t ) ;
    }
private :
    std::istream& is ;
} ;
```

NB: altri compilatori si confondono se il template member e' un operatore di conversione come sopra...

# Template parameter

Risultato: OK

Il compilatore riconosce e gestisce correttamente l'uso di template come parametri di template.

Mini esempio:

```
template< class T > struct C1
{
    void f1() {} ;
} ;
```

```
template< class T > struct C2
{
    void f1() {} ;
} ;
```

```
template< class T1,
         template< class T2 > class C
         > class D
{
public :
    D()
    {
        C< T1 > x ;
        x.f1() ;
    }
} ;
```

```
void check_template_parameter()
{
    D< int, C1 > d1 ;
    D< int, C2 > d2 ;
}
```

# Template export

Risultato: FAIL

Il compilatore riconosce, ma non implementa, la keyword `export` per la compilazione separata dei template.

Mini esempio:

```
export template< class T > class C
{
  // implementazione in altra translation unit
  void f() ;
} ;
```

# explicit qualification

Risultato: OK

Il compilatore consente di qualificare in modo esplicito i parametri delle funzioni template; inoltre opera la trail argument deduction correttamente.

Mini esempio:

```
template< int N > int f( int m )
{
    return( N * m ) ;
}

template< int N, class T >
int f2( const T& x )
{
    return( N * x ) ;
}

void check_explicit_arguments()
{
    int n = f< 6 >( 7 ) ;
    cout << n << "\n" ;
    int m = f2< 6 >( 7 ) ;
    cout << m << "\n" ;
}
```

# template keyword

Risultato: FAIL (parziale)

Il compilatore riconosce la keyword `template` come aiuto per risolvere problemi di ambiguita', ma non implementa ancora correttamente lo standard.

Mini esempio:

```
struct C
{
    template< int N > char* f()
        { return new char[ N ] ; }
} ;

void check_keyword_template()
{
    C c ;
    // must fail (ill-formed: < means less than)
    char* p1 = c.f<200>();
    // must compile (< starts explicit qualification)
    char* p2 = c.template f<200>();
}
```

# Specializzazione parziale

Risultato: OK

Il compilatore riconosce ed utilizza le specializzazioni parziali dei template.

Mini esempio (quasi cattivo)

```
template< int N, int M > struct C
  { enum { val = N * C< N-1, M >::val } ; } ;

template< int M > struct C< 0, M >
  { enum { val = M * C< 0, M -1 >::val } ; } ;

template<> struct C< 0, 0 >
  { enum { val = 1 } ; } ;

void check_partial_specialization()
{
  cout << "expecting 3! * 4! = 6 * 24 = 144\n" ;
  int n = C< 3, 4 >::val ;
  cout << n << "\n" ;
}
```

# Ordinamento parziale

Risultato: OK

Il compilatore utilizza l'ordinamento parziale delle template function per decidere quale chiamare in caso di overloading. Mini esempio:

```
template< class T1, class T2 > void f( T1, T2 )  
{  
    cout << "T1, T2\n" ;  
}
```

```
template< class T1, class T2 > void f( T1*, T2 )  
{  
    cout << "T1*, T2\n" ;  
}
```

```
template< class T1, class T2 > void f( T1, T2* )  
{  
    cout << "T1, T2*\n" ;  
}
```

continua...

# Ordinamento parziale

continua...

```
void check_partial_ordering()
{
    int x, y ;
    f( x, y ) ;
    f( &x, y ) ;
    f( x, &y ) ;
    f( &x, &y ) ; // Ambiguous
}
```

aggiungere questa rimuoverebbe l'ambiguita':

```
template< class T1, class T2 > void f( T1*, T2* )
{
    cout << "T1*, T2*\n" ;
}
```

# Qualita' del Codice

(C) Carlo Pescio

# Return Value Optimization

Sancita dallo standard, evita copie inutili

La copia "inutile" di oggetti e' uno dei casi piu' frequenti di inefficienza del C++. RVO elimina copia di oggetti anonimi.

```
class Vector
{
  // ...
};
```

```
Vector f1()
{
  return( Vector() ) ;
}
```

```
Vector f2( const Vector& v1, const Vector& v2 )
{
  return( Vector( v1 ) += v2 ) ;
}
```

```
void check_rvo()
{
  // expecting a single default constructor
  Vector v1 = f1() ;
  // expecting a single copy constructor, then +=
  Vector v2 = f2( v1, v1 ) ;
  // expecting two destructors
}
```

Risultato:  
OK (simple)  
FAIL (complex)

# Named Value Optimization

Sancita dallo standard, evita copie inutili

Simile alla precedente, entra in gioco quando vi e' un solo oggetto con nome che viene restituito.

```
class Vector
{
  // ...
};
```

```
Vector f1()
{
  Vector v1 ;
  return( v1 ) ;
}
```

```
Vector f2( const Vector& v1, const Vector& v2 )
{
  Vector v = v1 ;
  v += v2 ;
  return( v ) ;
}
```

```
void check_rvo()
{
  // expecting a single default constructor
  Vector v1 = f1() ;
  // expecting a single copy constructor, then +=
  Vector v2 = f2( v1, v1 ) ;
  // expecting two destructors
}
```

Risultato:

**FAIL** (simple)

**FAIL** (complex)

# Objects in Registers

Molto utile per evitare overhead da astrazione

Oggetti “piccoli” possono essere allocati nei registri, al pari dei tipi base che li compongono.

<pre>class Point { public :     // all inline... private :     int x, y ; } ;</pre>	<pre>Point f1( Point p, int i ) {     Point p1( i, i ) ;     Point p2 = p ;     p2 += p1 ;     return( p2 ) ; }</pre>
---	---

```
// I'm using a global variable to defeat
// empty optimization of check_register!
Point globalPoint( 0, 0 ) ;

void check_register()
{
    Point p( 42, 24 ) ;
    globalPoint = f1( p, 11 ) ;
}
```

Risultato:  
**FAIL**  
(see asm code)

# Empty Base Optimization

Sancita dallo standard, evita spreco di spazio

Lo standard permette di non allocare spazio per i sotto-oggetti di classe base aventi dimensione nulla.

```
// Size != 0
class Empty
{
};
```

```
// Size != 0
Empty e[ 10 ] ;
```

```
// Size != 0
class EM
{
    Empty e ;
};
```

```
// Size > sizeof( long )
class C
{
    Empty e ;
    long l ;
};
```

```
// Anomaly test!
class Base
{
};
```

```
class Derived :
public Base
{
    public :
        Base b ;
};
```

```
Derived d ;
bool e = (&d == &d.b) ;
```

```
// Size == sizeof( long )
class E : private Empty
{
    long l ;
};
```

Risultato:  
EMO illegale  
(disable option!)

EBO OK  
[anomaly]  
(disable option?)

# Conclusioni

(c) Carlo Pescio

# Conclusioni

## Un compilatore piu' che buono

Rimanendo in ambiente Windows, il C++ Builder 5 sembra uno dei compilatori meglio allineati con lo standard ANSI/ISO.

I pochi punti deboli di rilievo riscontrati nell'aderenza allo standard riguardano le eccezioni e l'esportazione dei template, mentre gli altri difetti sono "minori".

Dal punto di vista delle (macro-) ottimizzazioni vi sono ampi spazi di miglioramento, ed e' ancora compito del programmatore scrivere codice efficiente. Ad esempio, esistono numerose tecniche per evitare la copia (costruttori operazionali, classi duali, metaprogrammi template) che possono supplire alle carenze dell'ottimizzatore.

# Q & A



Carlo Pescio

# Approfondimenti

(C) Carlo Pescio

# Approfondimenti

## C++ Informer

Newsletter gratuita sul C++

Iscrizioni:

- <http://www.eptacom.net/newsletter>

Rubriche:

- ANSI/ISO C++  
es. RVO, NVO, costruttori operazionali,  
EBO discussi sui numeri 1 e 2
- No-Limits C++  
es. metaprogrammazione, overloading sul  
tipo del risultato, forName/newInstance, ecc.
- Q&A
- ecc

? Feedback, opinioni, ?  
domande, osservazioni,  
? giudizi... : ?

**Carlo Pescio**

pescio@eptacom.net

<http://www.eptacom.net>