V4.0 High Integrity C++ Coding Standard (HIC++). Un anno dopo.

Richard Corden

Il 3 Ottobre 2003, PRQA pubblicava la versione iniziale del *High Integrity C++ Coding Standard* (HIC++). Durante il decennio successivo la comunità degli sviluppatori ha scaricato questo standard di codifica ben25 mila volte.

Esattamente 10 anni dopo la pubblicazione iniziale, il 3 Ottobre 2013, abbiamo realizzato una *major revision* di HIC++.(V4.0) Il set aggiornato di regole si basa su quanto hanno insegnato i precedenti 10 anni, incorporando i *feedback* arrivati dalla comunità di utilizzatori HIC++, imparando anche dagli altri standard e, cosa più importante, affrontando i recenti cambiamenti operati allo stesso linguaggio C++ (in particolare C++ 2011).

In questo articolo verrà fornita una breve rassegna degli ultimi 12 mesi della V4.0, identificando le regole che sono state più frequentemente richiamate, tenendo in considerazione alcuni dei feedback dalla comunità degli utilizzatori, ed anche accennando ad alcune delle discussioni in corso tra gli esperti sul miglior modo di utilizzare il nuovo linguaggio, il *Modern C++*.

"Un linguaggio completamente nuovo"

In molti, incluso lo stesso Bjarne Stroustrup, guardano al Modern C++ come ad un "linguaggio completamente nuovo" (http://www.stroustrup.com/C++11FAQ.html#think). ISO C++ 2011 aveva aggiunto molte nuove funzionalità, e un obiettivo chiave di molte delle modifiche era stato quello di rendere il linguaggio più facile da usare e consentire agli sviluppatori di esprimere chiaramente il loro fine, piuttosto che come semplice "linguaggio appreso".

Di conseguenza, Modern C++ è (o dovrebbe essere) un linguaggio più sicuro del suo predecessore. La necessità di migliore comprensione e di promuovere il miglior/corretto uso delle nuove funzionalità era una delle principali spinte dietro la creazione del HIC++ V4.0.

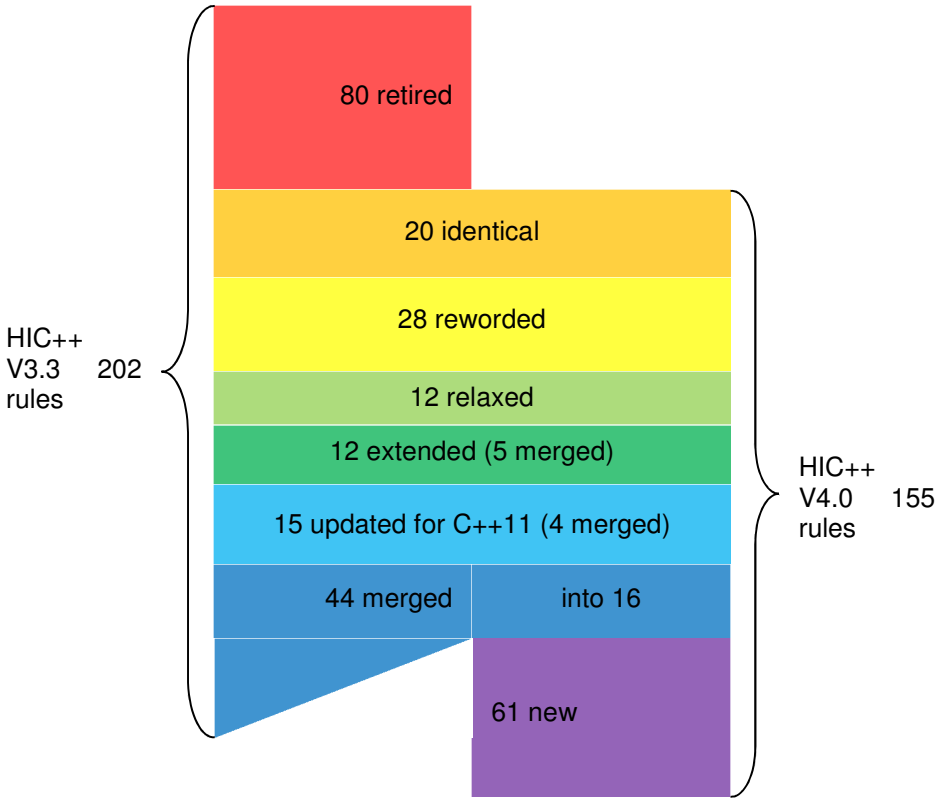
Siamo infatti anche più consapevoli dell'importanza di regole ben costruite e, in particolare, che le regole debbano essere applicabili, inequivocabili ed avere una chiara motivazione. La base logica è estremamente importante dato che questa consente di prendere decisioni informate riguardo il "*conforming to*" o "*deviating from*", di adeguarsi o derogare, da una regola all'interno di un dato contesto. Una regola vaga o non applicabile – sia per ispezione manuale che per analisi automatica – non è di alcuna utilità ma, anzi, è uno spreco del prezioso tempo degli ingegneri.

Ove possibile, le regole dovrebbero essere applicate automaticamente, liberando tempo da utilizzare per concentrarsi su livelli più elevati di progettazione e questioni strutturali. Naturalmente, ci sentiamo anche obbligati a sottolineare che non tutti gli strumenti automatici di analisi statica sono uguali (!), specialmente per cio' che riguarda la loro accuratezza – minimizzazione dei falsi positivi (rumore) e falsi negativi (un guasto identificato come situazione corretta).

Il set di regole V4.0 è stato derivato da più fonti:

- Regole adottate dalla V3.3, la precedente versione di HIC++, e migliorate (decadute, accorpate, riformulate, snellite, o estese)
- Regole adattate da standard esistenti
- Segnalazioni da parte di esperti (come Herb Sutter, Scott Meyers, Anthony Williams)
- Analisi diretta dello standard ISO C++ 2011 da parte degli esperti PRQA di linguaggio
- Controllo dei cambiamenti di linguaggio per C++ 2014 e successivi attraverso la diretta partecipazione dei *C++ Committee Meeting*

Nel complesso, il passaggio da V3.3 a V4.0 è riassunto come segue*:



HIC++ continua ad essere diverso dagli altri standard di codifica tradizionali come MISRA C++ e JSF++. A differenza di JSF++ non è proibita alcuna *major feature*, mentre sono offerte le migliori indicazioni per il corretto utilizzo di tutte le caratteristiche del linguaggio. Inoltre, HIC++ risolve problematiche utilizzando l'approccio C++ di una singola e potente regola che elimina la necessità di molte regole disgiunte. La *Rule 12.5.6* (ved. sotto) è un buon esempio di ciò, dato che evita la necessità di regole relative a "*checking for self assignment*" o di offrire "*strong exception guarantee*".

Le "*Top 5 Rules*"

L'adozione di HIC++ è realmente globale; sviluppatori provenienti ormai da 137 Paesi hanno fatto accesso al sito web HIC++ (www.codingstandard.com) dalla data di pubblicazione della V4.0. La maggior parte degli interessati proviene dagli USA e Germania (entrambi si attestano al 20% delle visite) seguiti da Russia, UK, India, Svezia, Francia, Polonia e Svizzera, e a seguire dai rimanenti 127 Stati (non serve elencarli tutti!) .

Uno degli esercizi più interessanti, ed istruttivi, è stato quello di identificare quali fossero le regole cui la comunità degli sviluppatori ha fatto riferimento più di frequente, e qui di seguito sono elencate le "*Top 5* " [$]:

Top 5	Rule	Descrizione
1	8.2.4	Do not pass std::unique_ptr by const reference
2	4.1.1	Ensure that a function argument does not undergo an array-to-pointer conversion
3	2.1.1	Do not use tab characters in source files
4	12.5.6	Use an atomic, non-throwing swap operation to implement the copy and move assignment operators
5	2.2.1	Do not use digraphs or trigraphs

* Estratto dal *PRQA Whitepaper :* "*High Integrity C++ Coding Standard V4.0 - an overview*"

[$] Risultati basati sulla combinazione di hit di pagine web e feedback diretti sulla versione PDF scaricata

Consideriamo ora ognuna di queste in maggior dettaglio.

Rule 8.2.4

Finora la *Rule 8.2.4* è stata considerata più frequentemente che qualsiasi altra in V4.0. Non c'è dubbio che ciò sia una conseguenza del recente dibattito tra Scott Meyers e Herb Stutter; questi due importanti personaggi del panorama C++ hanno discusso lungamente gli argomenti coperti da "Rule 8.2.4 Do not pass std::unique_ptr by const reference".

Gli argomenti sono convincenti da entrambe le parti, e sembra che entrambi gli approcci abbiano vantaggi e svantaggi. In definitiva, HIC++ segue le indicazioni di Herb Sutter, per il quale i *sink parameter* del tipo "*std::unique_ptr*" dovrebbe essere passati per valore. Scott Meyers, d'altro canto, si schiera a favore di una regola più generale nel dichiarare i *sink parameter* come *rvalue references (&&)*.

```
void herb_sp (std::unique_ptr<int>);
void scott_sp (std::unique_ptr<int> &&);
```

E' positivo anche il constatare che una comunità consulta HIC++ per sondare la nostra opinione su questoargomento!

Rule 4.1.1

La seconda regola più consultata sul sito web è "Rule 4.1.1 Ensure that a function argument does not undergo an array-to-pointer conversion", la cui origine è lo standard JSF C++ .

Il problema nasce dal fatto che un parametro di funzione che si presenta come un array è in realtà solo un puntatore:

```
void f1(int a[10]) { // Equivalent to void f1(int*)
    a[8] = 0;          // Out of Bounds when called from f2
}

void f2() {
  int b[5];
  f1(b);             // Not illegal code!
}
```

Le dimensioni dell'array in f1 possono facilmente ingannare un revisore umano e portarlo a credere che questo aspetto sia controllato dal compilatore, mentre nessun controllo verrà mai effettuato. In JSF C++, il passaggio di argomenti di tipo array viene completamente negato per questo motivo. L' HIC++ V4.0 tuttavia consente di autorizzare array come argomenti, ma solo quando l'informazione riguardante la dimensione non sia persa:

```
void f1(int (&a)[10]) { // Parameter is reference to
                        // array of 10 elements.
  a[8] = 0;             // Guaranteed to be legal
}

void f2() {
  int b[5];
  f1(b);                // Illegal - Compile Error
                        // Cannot convert int[5] to int[10]
}
```

Rule 2.1.1

Dobbiamo ringraziare la comunità di utilizzatori di HIC++ per tutti i loro feedback, la cui quasi totalità è stata costruttiva e positiva. I feedback tecnici tipicamente sono stati relativi a suggerimenti per le relazioni tra le regole o l'aggiunta di chiarimenti. Per esempio, "Rule 2.1.1, Do not use tab characters in source files" come attualmente formulato, richiede che siano utilizzati esclusivamente gli spazi (e non le tabulazioni) in file sorgenti. La motivazione al non permettere tabulazioni è quella di garantire che l'identificazione del codice sia coerente tra i differenti editor e IDE, per esempio:

```
if ( ... ) {
  ++i;    // 4 spaces
++j       // 1 tab - indentation incorrect
          // when tab not equal to 4 characters
}
```

Abbiamo ricevuto feedback (grazie a Jason, Larry e Dave) in merito al fatto che ciò potrebbe essere troppo restrittivo. La problematica viene anche considerata quando i tab sono utilizzati sempre per iniziare la riga:

```
if ( ... ) {  // 1 tab before if
```

```
    ++i;          // 2 tabs before ++i
    ++j           // 2 tabs before ++i
}                 // 1 tab before {
```

La nostra intenzione è quella di estrapolare piccoli miglioramenti come questo in successive revisioni dello standard.

Rule 12.5.6

Le comunità di sviluppatori sono inequivocabilmente una preziosa sorgente di informazioni sul miglior modo di utilizzare Modern C++. Spesso, molto si può apprendere dalle domande e risposte che appaiono sui forum dedicati. Un esempio è relativo a "Rule 12.5.6 Use an atomic, non-throwing swap operation to implement the copy and move assignment operators", che è trattato in molte domande e risposte sui forum quali StackOverflow. Questa regola assicura la corretta copia e spostamento di una classe ed al contempo offre una *solida safety exception.* Questa indicazione è eccellente, come lo è pure il fornire un modo semplice e compatibile per inquadrare questi membri in una *class*.

Ciò che è emerso, tuttavia, è che l'esempio di regola V4.0 può essere scritto come:

```
class A {
public:
  A (A const &) …
  A (A &&) …
  A & operator=(A rhs) & noexcept {
    swap (*this, rhs);
    return *this;
  }
};
```

Questo è un ottimo esempio di un corretto approccio che finisce per essere di gran lunga migliore e più semplice delle alternative!

Rule 2.2.1

Nel *Urbana-Champaign ISO C++ Committee Meeting* (dal 3 al 8 Novembre 2014), avrà luogo una votazione formale per rimuovere i trigrammi dal linguaggio (n3981). I trigrammi sono una sequenza di 3 caratteri che iniziano con ??. Il loro scopo è quello di consentire a C e C++ di essere scritti su sistemi che non supportano caratteri quali {, /, # etc. Sfortunatamente, i trigrammi male si adattano ad alcune funzionalità dei nuovi linguaggi quali "*raw string literals*".

```
const char * s = R"??=";
```

Una precedente indagine aveva evidenziato che i trigrammi venivano utilizzati sia involontariamente che deliberatamente quali trigrammi di test! Il verdetto è quindi di bandirli completamente dal linguaggio.

E' emerso che Rule 2.2.1 Do not use digraphs or trigraphs era una regola estremamente valida da seguire. La sola domanda restante è: succederà lo stesso con i digrammi?

In Conclusione

Un anno è trascorso e l'adozione di HIC++ V4.0 ha superato ogni nostra aspettativa; il documento e' stato scaricato altre 3.800 volte durante gli ultimi 12 mesi. Ciò dimostra la costante importanza di HIC++, e riflette il fatto che HIC++ continua ad evolversi e ad ospitare Modern C++, contribuendo a documentare le migliori pratiche ed aiutando gli sviluppatori a generare *High Integrity C++ Code*.

HIC++ V4.0 è disponibile su www.codingstandard.com, unitamente ad un whitepaper correlato.
Per qualsiasi feedback aggiuntivo non esitate a contattare l'autore all'indirizzo info@programmingresearch.com

Informazioni sull'autore

Richard Corden

Richard Corden è *Lead Software Developer* di PRQA, con responsabilità primaria su QA·C++. Richard è anche un "*Individual Expert*" nel *ISO C++ Committee* ed è uno dei principali autori del *High Integrity C++ Coding Standard*.