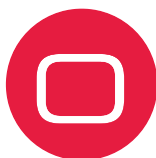
Software embedded, le linee guida per un codice di qualità

Complessità dei sistemi embedded, peso crescente della componente software e costante pressione di time-to-market sugli sviluppatori minano sempre più una buona attività di progettazione. La prima parte di questa mini-guida fornisce un'immagine dell'attuale scenario, fornendo poi alcune indicazioni utili a conservare la qualità del codice
I parte

Giorgio Fusari



ggi, la continua espansione del contenuto software e del numero di linee di codice in dispositivi e applicazioni embedded si può vedere come il risultato dall'azione combinata di due fattori chiave. Da un lato la potenza computazionale dei sistemi elettronici, in continuo aumento e dall'altro l'esigenza dei costruttori di dispositivi, e degli ingegneri sviluppatori, di integrare nei vari prodotti un numero sempre più elevato di funzionalità, capacità grafiche e opzioni di connettività. Di conseguenza, in questo scenario tecnologico per gli sviluppatori scrivere codice e software di qualità diventa un compito sempre più complesso è difficile, anche considerando gli attuali vincoli di time-to-market, che impongono la pianificazione di strategie di design con cicli di progettazione via via più compressi, ma anche sempre più 'tracciabili' in ogni fase di sviluppo. Accorgersi di un bug e scoprire i difetti di progettazione del software troppo avanti nel ciclo di produzione determina infatti un costo ormai per tutti proibitivo, sia in termini di impatto sui tempi di rilascio nel mercato, sia di danni economici e caduta d'immagine, in caso di ritiro del prodotto stesso.
Ci sono però altri aspetti che oggi rendono ulteriormente complicata la produzione di codice di qualità. In molti ambienti di lavoro e settori dedicati alla creazione del software embedded, è una pratica comune l'utilizzo di linguaggi di programmazione caratterizzati da una struttura semantica ricca e complessa, come C o C++. Inoltre, la necessità di realizzare sistemi sempre più sofisticati porta a un uso ancora più esteso anche di altri linguaggi ad alto livello (alto livello di astrazione) orientati agli oggetti, come Java o Python. Si adottano spesso ambienti IDE (integrated development environment) basati sulla piattaforma Eclipse, per integrare i vari tool e strumenti che entrano a far parte di ciclo di produzione (tool di simulazione, compiler, debugger e così via). In aggiunta, la crescente consapevolezza negli ambienti di produzione dell'importanza di accelerare l'adozione di procedure di co-design hardware e software fa sì che, nel mondo embedded, i processi di interazione e collaborazione fra i vari reparti di sviluppo e gli attori della supply chain (costruttori e fornitori di componenti) risultino tendenzialmente più intensi e complessi rispetto a quelli del mondo IT tradizionale e del software concepito per sistemi desktop.

Standard ISO/IEC 9126

La qualità del software è un principio irrinunciabile soprattutto nei sistemi embedded, dove i requisiti da soddisfare nelle diverse applicazioni vanno dalla elevata affidabilità di funzionamento, ad esempio in condizioni safety-critical, alla capacità di fornire performance di tipo deterministico (funzionamento di sistemi real-time). Di conseguenza, nei differenti domini embedded si sono consolidate pratiche di produzione del codice, mirate a mantenere comunque elevata la qualità.

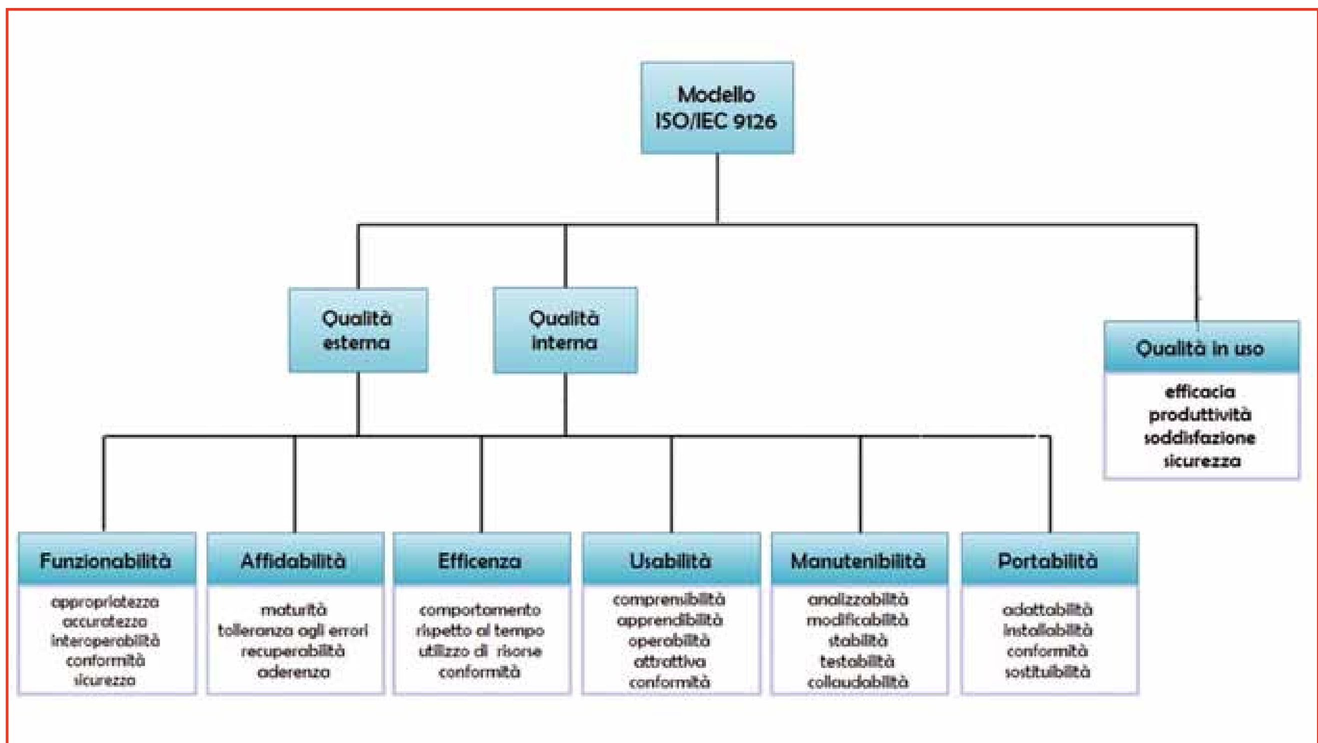


Fig. 1 - Una rappresentazione del modello ISO/IEC 9126 per la qualità del software

Si adottano strumenti per l'elaborazione di modelli matematici di analisi e ottimizzazione dell'affidabilità e delle prestazioni. Ma anche metodologie di analisi statica del codice, per la verifica di specifici requisiti di progetto (prestazioni, consumo di memoria, sicurezza), e analisi dinamica, necessaria per il collaudo di singoli componenti o del funzionamento del sistema nella sua interezza. Nelle applicazioni embedded in campo medicale, come in quelle dedicate al settore automobilistico o industriale, dove un malfunzionamento può significare un notevole danno di produzione o, addirittura, la messa in pericolo della sicurezza fisica dell'utente, la qualità del software non può certo essere oggetto di compromesso.
Per verificare la qualità del codice sorgente e dell'architettura software, sono quindi in uso diverse metriche e standard. A livello generale, un punto di riferimento è certamente lo standard ISO/IEC 9126, che definisce un modello di qualità del software.
Tale modello esprime la qualità come "l'insieme delle caratteristiche che incidono sulla capacità del prodotto di soddisfare requisiti espliciti o impliciti". Indica quindi metriche per valutare la 'qualità esterna', cioè la capacità del software, le sue prestazioni e funzionalità nell'ambiente in cui è in uso; la 'qualità interna', che si basa sulla valutazione delle proprietà intrinseche del software, misurabili direttamente sul codice sorgente; e infine la 'qualità in uso', che indica a quale livello il prodotto software sia utile per rendere efficace ed efficiente l'attività dell'utente (efficacia, produttività, soddisfazione, sicurezza nel contesto d'uso).
In sintesi, le caratteristiche che definiscono la qualità interna ed esterna di un prodotto software sono sei: funzionalità, affidabilità, usabilità, efficienza, manutenibilità e portabilità. In termini di funzionalità, si considera la capacità del software, in determinate condizioni, di svolgere i compiti per cui è stato progettato. Quindi si valuta se è dotato di funzioni appropriate per eseguire operazioni specifiche richieste dall'utente, se è preciso nel fornire i risultati, se ha una buona interoperabilità con altri sistemi. E se è sicuro, a livello di protezione dei dati e da tentativi di accesso non autorizzati. Passando all'affidabilità, cioè la capacità di mantenere determinate prestazioni sotto condizioni e tempi stabiliti, i criteri sono la robustezza (capacità di evitare arresti dell'applicazione in seguito a malfunzionamenti), la tolleranza a errori (capacità di mantenere un certo livello di prestazioni anche in caso di disfunzioni), e la velocità di ripristino dei livelli di prestazioni stabiliti, e di recupero dei dati, in caso di avaria. Per valutare l'usabilità, si analizza la qualità del software in termini di facilità di comprensione del funzionamento e modalità d'uso. Quindi praticità di apprendimento e d'uso, e gradevolezza del sistema. L'efficienza si misura osservando quante risorse vengono consumate in rapporto alle prestazioni, in determinate condizioni di funzionamento. La manutenibilità è la capacità del software di consentire modifiche con un impegno di lavoro

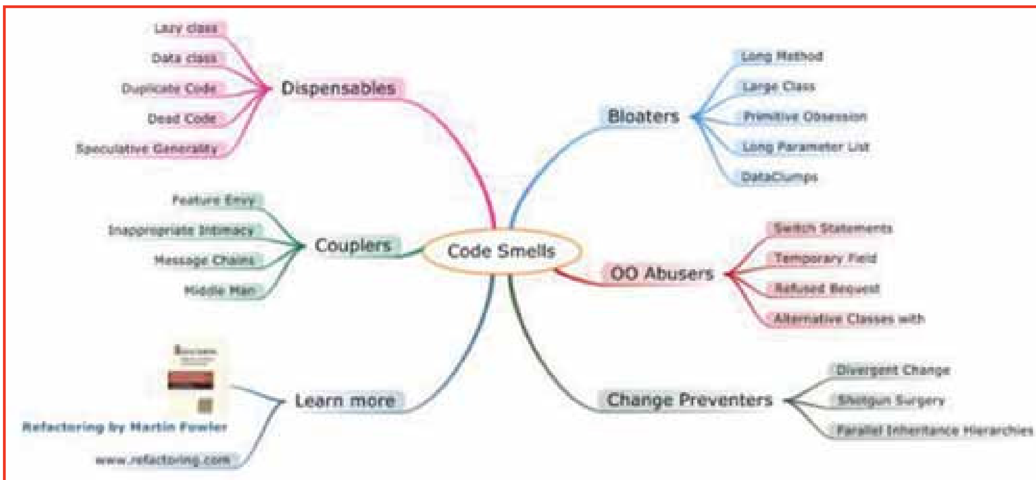


Fig. 2 – Una classificazione dei 'code smells'

limitato, ad esempio in caso di diagnosi di malfunzionamenti o carenze, o di analisi e individuazione delle modifiche da apportare. Il livello di modificabilità dipende dalla relativa facilità di correzione, quando occorre rimuovere errori o sostituire componenti. Un buona manutenibilità è legata anche a quanto il sistema si rivela stabile (rischio ridotto di comportamenti anomali e inattesi) dopo l'esecuzione delle modifiche, e alla capacità che ha di essere collaudato facilmente per validare le modifiche stesse.
Infine, la portabilità esprime la facilità con cui è possibile trasportare il software da un ambiente operativo a un altro. Quindi la sua capacità di adattamento al nuovo contesto, limitando al massimo le modifiche necessarie; la facilità di installazione in un determinato ambiente operativo; l'abilità di coesistenza e condivisione di risorse con altri software, e l'intercambiabilità, ossia la possibilità di essere sostituito a un altro software per eseguire gli stessi compiti nel medesimo ambiente.

Degrado del codice, i primi segnali

Un imperativo sempre più categorico, nell'interesse di tutti gli attori cooperanti all'ecosistema di produzione dei software, è fare in modo che la complessità degli attuali progetti, le esigenze di rispettare tutti i requisiti delle applicazioni, e la pressione del time-to-market, non incidano in modo negativo sulla qualità del software. E ciò, specie nelle realtà con ampi team di progettazione, impone l'adozione di metodologie di co-design hardware-software, tool, e strategie organizzative di monitoraggio della qualità, finalizzati a individuare ed eliminare i difetti del software il più possibile già nelle fasi iniziali del ciclo di produzione, senza dover attendere di scoprire i bug negli stadi finali di test e collaudo.

Tuttavia, più il progetto cresce, e i singoli componenti e moduli software diventano articolati e interdipendenti, più diventa difficile eseguire operazioni di manutenzione, modifica e test. Il consulente software internazionale Robert C. Martin paragona questo processo di degradazione a quello di un pezzo di carne che sta marcendo. Anche nel caso del software si possono sentire 'cattivi odori'. Fra questi 'bad smells', avvisaglie che il software sta perdendo qualità, ve ne sono cinque fondamentali, definiti "rigidità", "fragilità", "immobilità", "viscosità" e "opacità". Il software diventa rigido quando la modifica anche di un limitato pezzo di codice obbliga lo sviluppatore a effettuare ulteriori modifiche su altre parti. Fragile, quando apportare un cambiamento causa il collasso del sistema in punti apparentemente non in relazione con il problema a cui si sta lavorando. Alcuni sviluppatori navigati, ricorda Rebecca M. Riordan, progettista internazionale di sistemi computerizzati e autrice del libro Fluent C#, sono soliti dire: "Esistono sempre tre bug: quello che si conosce, quello che non si conosce e quello che si sta creando per risolvere quello che si conosceva". C'è poi la immobilità del codice. Tipicamente si riscontra quando, ad esempio, si sta modificando un'applicazione e si desidera riutilizzare un pezzo di codice, ma farlo risulta molto arduo. La viscosità è la tendenza del software a facilitare la commissione di errori da parte del programmatore, mentre l'opacità si riferisce alla difficoltà di leggere e comprendere bene il codice.

Dipendenze, buone e cattive

Un tema chiave strettamente connesso alla produzione e manutenzione di codice di qualità e quello delle dipendenze fra i vari pacchetti e moduli, specie in contesti tecnologici come quelli attuali, in cui gli aggiornamenti del software sono molto frequenti. La presenza delle dipendenze fa sì che, ad esempio in Linux, a livello di amministrazione del sistema, l'installazione di un nuovo pacchetto software possa richiedere a sua volta il soddisfacimento di varie dipendenze e quindi l'installazione a catena di moduli software aggiuntivi. A livello architetturale, un'elevata interdipendenza dei sottosistemi – in cui un singolo cambiamento provoca a cascata la necessità

di apportare una serie di cambiamenti in tutti i vari pacchetti dipendenti – può diventare pesante da gestire in termini di evoluzione e manutenzione del software, al punto da bloccare i processi di autorizzazione alle modifiche dei pacchetti stessi. A quel punto il progetto software diventa rigido, fragile, e difficile da modificare. O anche da riutilizzare, se le componenti del progetto che si desidera riusare si rivelano altamente dipendenti da altri aspetti tecnici su cui risulta troppo oneroso e complesso intervenire.
Una via di soluzione del problema è adottare tecniche di sviluppo in grado di portare verso la creazione di dipendenze definite 'buone'. Un codice robusto, manutenibile e riutilizzabile si caratterizza per la scarsità di interdipendenze. E, dove presenta dipendenze, queste sono buone, nel senso che non condizionano il raggiungimento dei requisiti richiesti per il progetto, perché si rifanno a classi stabili. Una classe è tanto più stabile quanti meno legami ha con altre classi, e in tale accezione si definisce classe indipendente. In questo campo, un sentiero di guida della progettazione object-oriented può essere rappresentato dallo Stable Dependencies Principle (SDP), secondo il quale le dipendenze fra i pacchetti, in un progetto software, dovrebbero andare nella direzione della stabilità dei pacchetti stessi. Quindi un pacchetto software dovrebbe dipendere solo da altri moduli più stabili. Allo stesso tempo, però, visto che un progetto non può essere completamente statico, è necessario conformarsi anche a un altro principio, il cosiddetto Common Closure Principle (CCP), che disciplina la creazione di pacchetti software progettati per essere più 'volatili' e adatti a subire modifiche, in modo da ridurre l'impatto dei cambiamenti sulla manutenibilità del sistema, specie quando il progetto è grande e si compone di molti pacchetti.

Ottimizzazione: non sempre va d'accordo con portabilità

Un software di qualità si contraddistingue per la flessibilità, solidità e stabilità, la facilità di riutilizzo dei componenti e la trasparenza di comprensione. Nella pratica, però, l'esigenza di rispettare determinati requisiti del sistema embedded entra in collisione con questi princìpi virtuosi: come detto precedentemente, qualità del software significa anche portabilità, ma in vari casi questo non accade. Come quando si ha a che fare con il software di sistemi embedded con architettura a 16, 32 o 64 bit, scritto in un linguaggio di programmazione ad alto livello come il linguaggio C. Se è vero che quest'ultimo consente la portabilità del codice su diverse piattaforme utilizzando vari compilatori, è vero anche che molto del software sviluppato in C per applicazioni embedded non viene scritto con questo obiettivo primario. Piuttosto, il punto diventa ottimizzare caratteristiche specifiche del sistema, come l'uso della memoria o le prestazioni. Quindi il codice viene creato 'su misura', usando un compilatore con opzioni modificate per soddisfare tali requisiti, e non risulta portabile. Così, successivamente, quando tale codice si fa migrare su un nuovo ambiente operativo, magari usando un compilatore C differente, tipicamente va in crash e si blocca.

C e C++, attenzione alle 'insidie'

L'utilizzo del linguaggio ad alto livello C++ è ampiamente diffuso nelle attività di sviluppo software per i sistemi embedded, anche quando si tratta di realizzare applicazioni safety-critical o sistemi che devono mantenere un funzionamento hard-real-time. Tuttavia occorre anche aggiungere che C++ (come anche C), proprio per la sua ricchezza semantica, in un certo senso non si posiziona esattamente come un linguaggio adatto allo sviluppo di software di categoria safety-critical. Infatti, offrendo al programmatore un'elevata gamma di sfumature e libertà espressive in fase di sviluppo, impone maggior attenzione, preparazione e responsabilità per non incorrere in errori. Se il linguaggio C, per le sue caratteristiche, si può paragonare, dal punto di vista della programmazione, a una spada con una lama tagliente, qualche sviluppatore parla di C++ come di un spada a due lame, ancora più tagliente e bisognosa di responsabilità e controllo. Le difficoltà e i 'pericoli' derivanti dall'utilizzo del linguaggio C++ risiedono nel fatto che esso è uno strumento davvero molto potente, capace di portare in errore gli sviluppatori non sufficientemente formati e disciplinati sull'applicazione delle sue regole. In altri termini, con C++ in molti casi risulta troppo facile raggiungere elevati livelli di astrazione e creare codice 'offuscato', difficile da leggere e incomprensibile nel lungo termine.

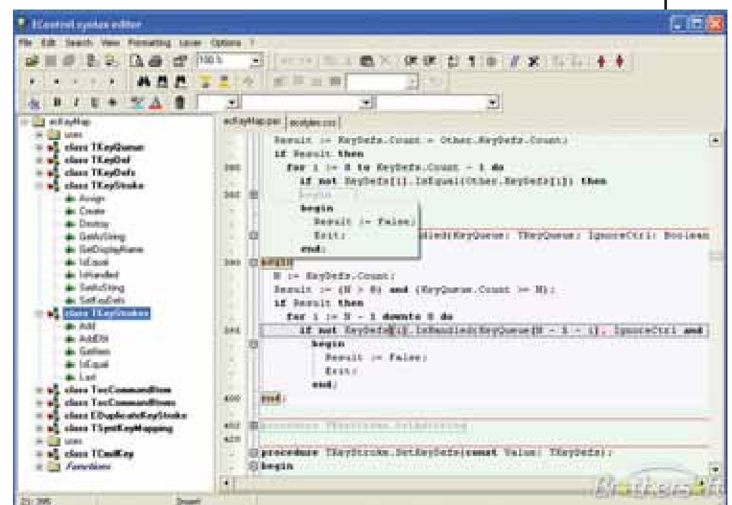


Fig. 3 – Uno strumento di analisi ed evidenziazione della sintassi del codice

Nella prossima parte di questa guida allo sviluppo di codice di qualità si analizzeranno alcuni aspetti positivi e negativi dell'uso del linguaggio C++ per lo sviluppo di sistemi embedded safety-critical, indicando alcune regole e linee guida per la creazione di codice di qualità.

C++: la 'grammatica' per un codice senza errori

II parte*

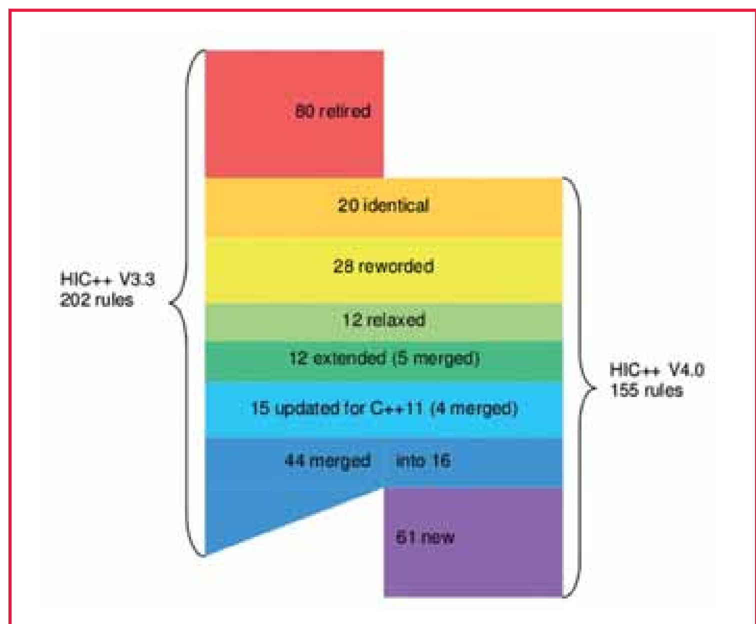
La complessità di questo linguaggio esige una conoscenza profonda delle regole sintattiche e della semantica, specie quando si sta sviluppando codice per sistemi embedded safety-critical

Giorgio Fusari

ome anticipato nella prima parte di questa mini-guida dedicata a chi intende sviluppare codice di qualità per il mondo embedded, in questo secondo articolo ci si concentrerà sull'uso del linguaggio di programmazione C++. In particolare, si analizzeranno le principali tipologie di standard di codifica, fornendo alcuni consigli e principi base per evitare gli errori classici, e andare verso la produzione di codice il più possibile limpido, comprensibile, corretto, portabile e manutenibile. Caratteristiche, queste, che certamente vanno nella direzione di favorire il lavoro collaborativo negli ambienti e nelle comunità di sviluppatori.

Già nella prima parte di questo articolo si accennava come il linguaggio C++, nel tempo, sia diventato uno strumento di sviluppo sempre più diffuso e utilizzato dai progettisti di sistemi embedded, anche di tipo safety-critical, e per applicazioni caratterizzate dalla necessità di soddisfare requisiti di funzionamento 'hard real-time'. Una delle ragioni di tale diffusione è legata alla ricchezza semantica di C++, all'elevato livello di astrazione e alla flessibilità che fornisce allo sviluppatore impegnato nella creazione di applicazioni complesse.

Tra i vari benefici, C++ supporta direttamente la programmazione ad oggetti (object oriented), principio progettuale ormai largamente adottato, per i suoi vantaggi nel mondo del software. In aggiunta, la disponibilità sul mercato di strumenti di calcolo tecnico come MATLAB, ampiamente adottati da ingegneri e ricercatori nel mondo industriale e universitario, consente ai diversi sviluppatori di generare in automatico codice C/C++. Ancora, i compilatori C++ sono reperibili per la stragrande maggioranza delle piattaforme hardware, e ciò agevola la portabilità delle applicazioni scritte in questo codice su diverse categorie di processori.



Le variazioni delle regole nello standard di codifica HIC++ V3.3 e V4.0

Non proprio adatto ai sistemi safety-critical

Dopo aver enumerato i pro, ora veniamo ai contro del linguaggio C++. La stessa ricchezza semantica, la potenza e il grado di libertà in fase di sviluppo che C++ fornisce all'utente può trasformarsi al contempo nella sua 'debolezza': un'arma a doppio taglio, che esige maggior autodisciplina e capacità di controllo da parte dello stesso sviluppatore. In altri termini, C++ non rappresenta propriamente uno strumento ideale da usare nella progettazione di applicazioni safety-critical, dove il software deve funzionare con estrema affidabilità, in sistemi 'delicati' (sistemi automotive, applicazioni militari, industriali, medicali), in cui un malfunzionamento o un comportamento inatteso del codice può determinare conseguenze anche molto gravi e tragiche. Perché C++ non sarebbe adatto? Perché la natura complessa di questo linguaggio incrementa la probabilità di introdurre nel programma errori che non possono essere identificati e segnalati da un compilatore.
Alcune strutture sintattiche di C++, e la sua semantica, lasciano talvolta spazio ad ambiguità di interpretazione (ad esempio, le ambiguità possibili nella chiamate delle funzioni). Ambiguità che poi, a livello di compilazione, si traducono e si riflettono in variazioni dei comportamenti del programma a seconda del tipo di compilatore utilizzato. Da ciò si deduce l'importanza e la necessità di acquisire una conoscenza più profonda possibile del linguaggio C++, per evitare che la notevole complessità semantica vada a ripercuotersi negativamente sul programma, con l'insorgenza di un maggior numero di errori durante l'attività del compiler. Nello specifico caso dei sistemi embedded che si rivelano critici per la sicurezza fisica – ossia le applicazioni 'safety-critical', dai sistemi avionici, a quelli medicali – occorre evitare quanto più è possibile l'uso di costrutti e forme sintattiche del codice che portino a funzionamenti imprevisti del software.

Standard e linee guida di codifica: i riferimenti chiave

Nel tempo, una strategia indirizzata a ridurre la possibilità di produrre difetti ed errori di programmazione nello sviluppo del codice, con il conseguente manifestarsi di funzionamenti inattesi nel sistema embedded, è stata l'adozione dei cosiddetti standard di codifica. Questi ultimi, in sostanza, puntano a canalizzare e concentrare le funzionalità di C++ entro un insieme più ridotto di caratteristiche e funzioni, che un programmatore può adoperare con una ragionevole sicurezza, senza la preoccupazione di generare errori e inconvenienti di vario tipo nel successivo comportamento del codice creato per una specifica applicazione.
Tra i benefici chiave dell'utilizzo degli standard di codifica non vi è soltanto un incremento della qualità e della manutenibilità del codice: a guadagnarne è anche la velocità di sviluppo, e il lavoro di gruppo, che può essere svolto con maggior efficienza, grazie a una migliorata leggibilità dei programmi.
Nello specifico settore dei sistemi embedded safety-critical, negli anni sono comparsi sulla scena diversi standard di codifica. Tra i più noti c'è, ad esempio, JSF AV C++ (Joint Strike Fighter Air Vehicle C++) - lo standard utilizzato per l'utilizzo di C++ nello sviluppo di software avionico nell'ambito del programma JSF del dipartimento della Difesa Usa. JSF C++ ha mutuato varie regole dal MISRA-C (Motor Industry Software Reliability Association), uno standard di codifica pubblicato per la prima volta nel 1998, rivisto nel 2004, e indirizzato a facilitare l'utilizzo del linguaggio C nello sviluppo di sistemi safety-critical. Tuttavia, la sempre maggiore tendenza a favorire la diffusione e l'adozione di C++ per la progettazione di sistemi safety-critical ha portato alla definizione, nel 2008, dello standard di codifica MISRA-C++.
Un altro standard di codifica che fornisce regole e raccomandazioni per la codifica sicura è lo standard CERT C++. Anch'esso punta all'obiettivo chiave di eliminare le pratiche di codifica inaffidabili e i comportamenti indefiniti del software, che possono portare alla formazione di vulnerabilità sfruttabili. Il principio base è sempre far leva su uno standard di codifica sicuro, in grado di condurre alla creazione di sistemi di più alta qualità, caratterizzati da una maggior robustezza e resistenza ad attacchi e minacce.
Ancora, tra gli standard di codifica per C++, una posizione di rilievo è occupata anche dallo standard High Integrity C++ (HIC++), pubblicato in origine nel 2003 come un insieme di 202 regole e linee guida semantiche e sintattiche, ricavate da un subset 'sicuro' del linguaggio C (ISO C++ 2003). Dopo oltre un decennio di attività, HIC++ ha ricevuto un aggiornamento importante con l'update HIC++ V4.0, rilasciato verso la fine del 2013. Quest'ultimo ha comportato il ritiro di 80 regole per varie ragioni specifiche, con l'obiettivo generale di creare un insieme di regole più applicabile e gestibile.

Principi generali

Prima di passare all'analisi di alcune linee guida specificate dai diversi standard di codifica, vale la pena fornire alcune indicazioni e consigli generali da seguire durante la scrittura del codice. Un primo criterio fondamentale, volto a favorire la leggibilità e chiarezza del codice, è imparare a usare identificatori appropriati.
Un identificatore è una sequenza di caratteri che si usa per definire un elemento del programma: ad esempio, il nome di una variabile, di una funzione, di un oggetto o di una classe. In C++ - che è un linguaggio 'case sensitive',

e fa quindi distinzione fra un carattere maiuscolo e uno minuscolo - gli identificatori sono combinazioni di caratteri alfanumerici, in cui il primo carattere deve essere una lettera dell'alfabeto (maiuscola o minuscola) o un carattere di sottolineatura (underscore '_'), mentre i rimanenti possono essere lettere, cifre o underscore. Negli identificatori, per i nomi non è ammesso utilizzare parole chiave, e nemmeno caratteri speciali. Alcuni esempi di identificatori validi sono:

nome (l'uso di lettere minuscole è consentito)
Cognome (l'uso di lettere maiuscole e minuscole è consentito)
_alpha (l'uso del carattere di sottolineatura all'inizio è permesso)
_SOMMA (l'uso del carattere di sottolineatura all'inizio è permesso anche con lettere maiuscole)
numero_1 (l'uso di underscore e numeri è permesso, assieme alle lettere)
INT (qui l'uso della parola chiave riservata 'int' di C++ è consentito, cambiando però le lettere in maiuscolo)

Esempi di identificatori invalidi, fonte di possibili errori in fase di compilazione, possono invece essere:
int (è una parola chiave riservata di C++)
numero 1 (l'uso degli spazi non è consentito)
8alpha (il primo carattere non è una lettera)
In aggiunta, una nota riguarda la chiarezza dell'identificatore: se da un lato, quando il contesto logico risulta chiaro, l'uso di identificatori brevi e concisi per funzioni e variabili favorisce la sintesi, giovando alla leggibilità, dall'altro, quando il contesto è imprecisato, è consigliabile scrivere identificatori in grado di precisare lo scopo dell'elemento identificato. Quindi adottare, quando il contesto lo richiede, identificatori con un nome più lungo, descrittivo e legato al significato logico ricoperto all'interno del programma: cosa del resto facilitata dal fatto che C++ permette di scrivere nomi anche con un notevole numero di caratteri.
Un'altra indicazione generale che si può fornire è scegliere un'unica lingua – ad esempio quella inglese – utilizzandola nell'intero ciclo di sviluppo del programma. L'uso della lingua italiana può essere consigliabile se il team di progettazione non conosce l'inglese, ma poiché quest'ultima lingua è usata come riferimento nella creazione della stragrande maggioranza delle librerie, l'adozione dell'italiano potrebbe costituire una limitazione non trascurabile per la diffusione, e il riuso del codice in un contesto internazionale di programmazione. Nei vari programmi è anche consigliabile evitare l'uso simultaneo di lingue differenti. Inoltre, la definizione degli identificatori, quindi dei nomi, per le classi e i tipi dovrebbe privilegiare nomi descrittivi, che riflettano la tipologia dei dati. È preferibile un nome che identifichi una rappresentazione astratta della classe, rispetto a una sua implementazione specifica. Nella definizione di funzioni e procedure, è consigliabile che il nome indichi, con un corretto livello di astrazione, ciò che la funzione calcola, o i vari task che la procedura esegue.
Per le variabili, è bene scegliere fin dall'inizio gli identificatori in modo opportuno, tenendo presente che sarà improbabile, in fase di debugging e manutenzione del programma, che verranno cambiati. Qui, analogamente ai discorsi fatti in precedenza, il principio è utilizzare indicatori con nomi brevi nel caso di variabili locali con un uso chiaro nel relativo contesto, e nomi più lunghi e descrittivi per le variabili con un campo di validità globale.

Codice per sistemi critici, i passi falsi da evitare

Quando si utilizza il linguaggio C++ nello sviluppo di un sistema safety-critical, ci sono alcuni specifici aspetti su cui occorre porre particolare attenzione. Qui ne evidenziamo alcuni.
Limitare l'uso del preprocessore. Quando si tratta di sistemi embedded critici per la safety, è bene rispettare alcune limitazioni a cui è soggetto l'utilizzo del preprocessore, ossia il programma che riceve ed esegue le direttive (le righe che iniziano con il carattere #) , producendo codice sorgente per il compilatore. In sostanza, l'uso del preprocessore viene limitato soltanto per l'implementazione delle direttive #include guard, evitando l'uso macro complesse e di inclusioni multiple. Come prescrive lo standard di codifica HIC++, dovrebbero essere utilizzate solo le seguenti forme di direttive #include:

```
• #include <xyz>
• #include "xyz"
```

Fonte PQRA

Ecco poi alcuni esempi di conformità, e non conformità, del codice indicati nelle linee guida HIC++:

```
// Compliant
#include <stddef>

// Non-Compliant
#define MYHEADER "stddef"
#include MYHEADER

// Non-Compliant
#define CPU 1044
#ifndef CPU
#error "no_CPU_defined"
#endif
```

Fonte PQRA

Limitare (o evitare) l'uso dell'allocazione dinamica della memoria. L'utilizzo della memoria dinamica nei sistemi safety-critical è di norma fortemente soggetto a restrizioni, e spesso sconsigliato. Il fondamento logico di questa raccomandazione è che usando l'allocazione dinamica della memoria si aumenta la probabilità che il sistema embedded possa incorrere in bug, e tenda ad assumere un comportamento non deterministico e impredicibile, con impatti sulle performance. L'allocazione statica della memoria consente invece di ottenere un pieno determinismo ed evitare bug di tal genere.

Sintassi per le istruzioni condizionali. Quando si utilizzano le istruzioni if, else, while, for, do e switch, occorre evitare costrutti che possono portare a scrivere codice non valido. Dopo ogni istruzione, ciascun blocco deve essere racchiuso tra parentesi graffe, anche se è vuoto o contiene solo una riga. Ad esempio:

```
#include <cstdint>

void doSomething ();

void foo (int32_t i)
{
  if (0 == i)
    doSomething (); // Non-Compliant
  else
    ;               // Non-Compliant

  if (0 == i)
  {                 // Compliant
    doSomething ();
  }
  else
  {                 // Compliant
  }

  switch (i)
  case 0:
    doSomething (); // Non-Compliant
}
```

Fonte PQRA

Scrivere codice conforme allo standard C++ ISO 2011. Una raccomandazione generale (implementation compliance) riportata nello standard di codifica HIC++ V4.0 spiega che l'attuale versione del linguaggio C++ è quella definita dallo standard internazionale ISO/IEC 14882:2011.
L'osservanza di questo criterio è fondamentale, perché sul mercato esistono compilatori che spesso forniscono funzionalità che oltrepassano quelle definite nello standard sopra citato, e un uso sregolato di tali caratteristiche finirebbe per ostacolare la portabilità del codice.
Adottare tool di automazione della verifica della conformità del codice. Sul mercato esistono vari prodotti e strumenti software in grado di automatizzare l'analisi del codice sorgente, per controllarne e verificarne la conformità secondo le regole definite dalle diverse versioni degli standard di codifica, come ad esempio MISRA-C++ o JSF AV C++.

Evitare il codice implementation-defined, o con comportamento non definito. Un altro criterio essenziale da rispettare è non indirizzarsi verso lo sviluppo di costrutti sintattici e codice, cosiddetto, 'implementation-defined' (definito dall'implementazione), o con un comportamento 'unspecified' (imprecisato) o 'undefined' (non definito).
Infatti, un codice di questo tipo non è portabile e viene bandito dagli standard di codifica. A differenza di un codice conforme allo standard, e compilabile da qualunque compiler fedele agli standard stessi, un costrutto del linguaggio C++, catalogato come implementation-defined, potrà essere implementato a seconda del compilatore in modi differenti, manifestando comportamenti definiti e documentati, con eventuale specifica di quali sono quelli consentiti.
Quando invece il codice è 'unspecified', significa che il comportamento dell'implementazione non è necessariamente documentato.
Infine, il comportamento 'undefined' del codice indica che gli standard non pongono alcun requisito da soddisfare per l'implementazione: quindi il compilatore potrebbe fallire durante la compilazione di porzioni di codice, andare in crash, produrre in modo silente risultati scorretti o, anche, eseguire in maniera fortuita proprio ciò che lo sviluppatore intendeva ottenere.

Naturalmente, in questi pochi punti abbiamo solo evidenziato alcuni aspetti rilevanti che riguardano le migliori pratiche di codifica del codice. Per una trattazione più approfondita e completa è possibile consultare le specifiche linee guida e documenti (ad esempio HIC++) - in alcuni casi scaricabili liberamente online – che raccolgono tutte le principali regole e best practice di programmazione per il linguaggio C++.

Nota
**La prima parte è stata pubblicata sul n. 52-giugno 2014 con il titolo "Software embedded, le linee guida per un codice di qualità"*
Online su http://elettronica-plus.it/brochure/emb/52/#64