Sarebbe bello che....

Ovvero: tutto quello che sarebbe bello riusciste a mettere in pratica per diventare informatici brillanti

Sarebbe bello che...

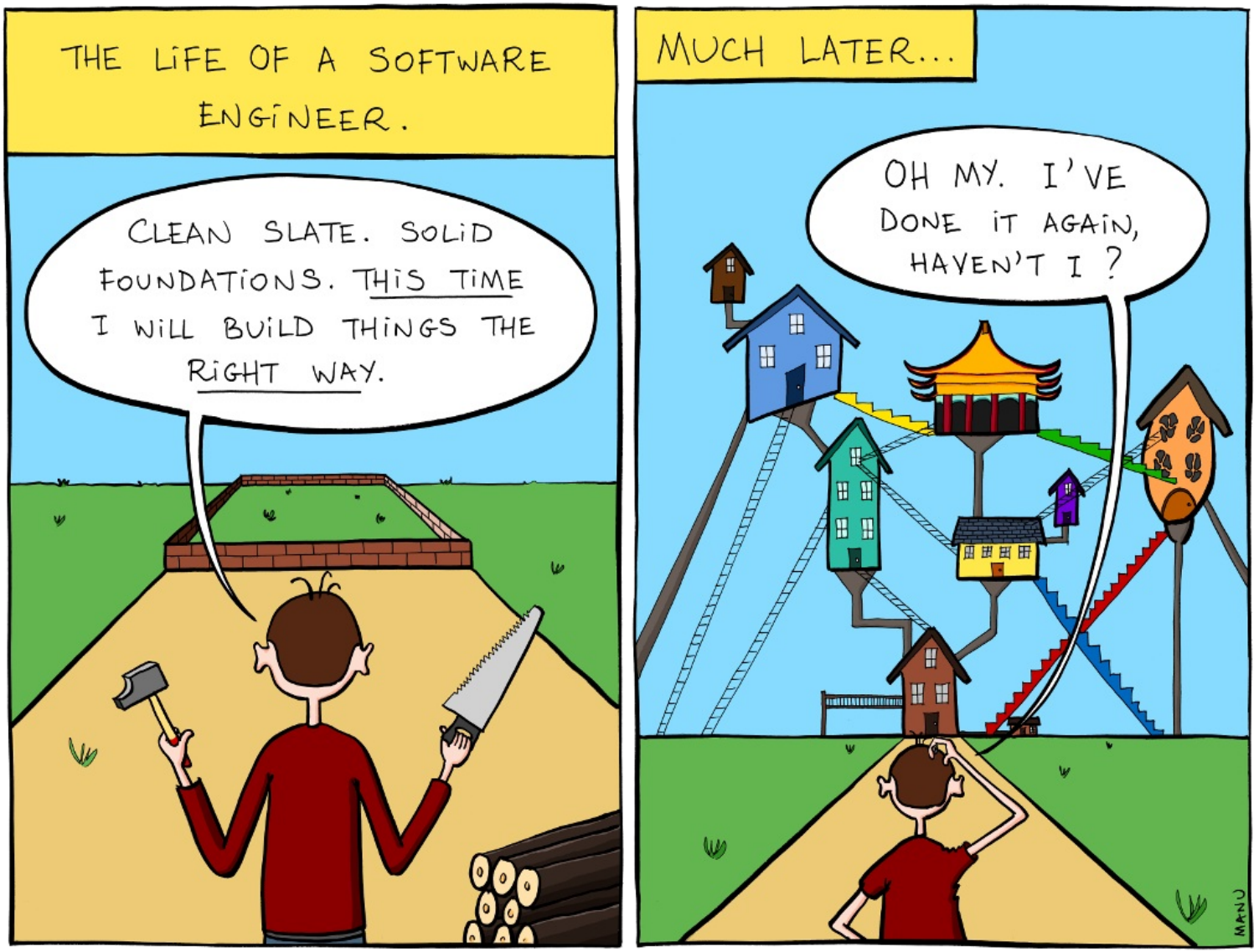
Che bello quando...

- ...si incrocia chiunque, lo si saluta;
  se si fa un sorriso, è ancora meglio!

- ...si manda una mail, la si inizia con “Buongiorno” o “Buonasera” (oppure con “Ciao” o “Cara/o ...”, ...) e la si conclude con una formula di cortesia come “Grazie” o “Buona giornata”. Ci si firma sempre per esteso: nome, cognome, eventualmente matricola

- ...si riceve una risposta a una mail, si risponde quanto prima ringraziando per l'attenzione

Capitolo 1: attitudine con cui si affronta la stesura del codice



L'ottimista cosmico

- Non gli serve leggere la specifica dell'esercizio: sa già cosa vogliono i professori (sono esseri così scontati...)
- Non gli serve usare carta e penna per ragionare sul problema e progettarne la soluzione ad alto livello, prima di implementarla: ha già tutto il codice in mente, dal primo “#include” all'ultimo “catch”, passando per tutti i “->next” !
- Ma soprattutto... **non gli serve compilare!!!** Il programma richiede 500 righe di codice? Le scrive tutte di fila, **non compilando mai**, convinto che saranno giuste.... Del resto, è un ottimista cosmico!

Rischi che corre l'ottimista cosmico

- Prima o poi, anche l'ottimista cosmico deve compilare. E allora succede la tragedia: 432 errori, 897 warning, il computer che si scalda spaventosamente....
- Capire come risolvere il problema è impossibile: gli errori sono troppi. L'ottimista cosmico cancella le sue 500 righe di codice e ricomincia da capo. Senza leggere il testo, senza progettare con carta e penna, e rigorosamente senza compilare.
- Dopo un paio di cicli “scrivo-compilo all'ultimo-butto via”, l'ottimista cosmico inizia ad avere qualche dubbio sul suo metodo...
- Dopo un altro paio di cicli di “scrivo-compilo all'ultimo-butto via”, inizia a declamare “Madre è di parto e di voler matrigna....”
- Dopo gli ultimi tre cicli di “scrivo-compilo all'ultimo-butto via” esce di casa declamando (rigorosamente in tedesco!) frammenti di “Die welt als wille und vorstellung”; viene intercettato da gentili signori in camice bianco che lo fanno accomodare su un veicolo bianco dotato di sirena e lo portano in vacanza, insieme a Napoleone e Giulio Cesare.

La comprensione dei raffinati riferimenti letterari e filosofici è lasciata per esercizio


Ottimista cosmico

Quello che crede ancora a P. Nocchio

- Quello che crede ancora a P. Nocchio è infinitamente più metodico dell'ottimista cosmico
- Legge il testo con attenzione e ci ragiona sopra
- Prende carta e penna e inizia a disegnare blocchetti, frecce, alberi, liste, progettando con cura l'algoritmo che deve implementare e scrivendone lo pseudo-codice
- Quando è convinto di avere capito il problema e la sua soluzione, inizia a implementare, compilando con regolarità ogni volta che conclude un frammento sostanzioso di codice o una funzione, e corregge gli errori di sintassi man mano che si presentano

Quello che crede ancora a P. Nocchio

- L'unico difetto di questa tipologia di informatico, è credere ancora al Teorema di P. Nocchio

Teorema di P. Nocchio

“Se compila, funziona!”

Dimostrazione

Per assurdo: con tutta la fatica che ho fatto a farlo compilare, sarebbe assurdo che non funzionasse!

Quello che crede ancora a P. Nocchio

- Al primo run di esecuzione, quello che crede ancora a P. Nocchio ottiene uno spaventoso “Segmentation fault”
- Convinto della correttezza del Teorema di P. Nocchio, non si spiega come questo sia potuto accadere e cade in uno stato di profonda confusione
- Inizia a disseminare il codice di
  - `cout << “Sono qui”;`
  - `cout << “Adesso invece sono qui”;`
  - `cout << “Sono quiiii”;`
  - `cout << “Sono quiiiiiiiiiiiiiiiii”;`
  - `cout << “Mi sa che qui non ci arrivo”;`

  ma trovare la causa dell'errore non è facile: la tentazione andare a trovare l'ottimista cosmico e Napoleone è forte...

Quello che crede ancora a P. Nocchio

Quello che non ha capito l'induzione

- Quello che non ha capito l'induzione è un'evoluzione, in meglio, di quello che crede ancora a P. Nocchio
- Legge il testo, ragiona, progetta con carta e penna, compila ogni volta in cui sviluppa una porzione di codice significativa, e fa un run del codice man mano che sviluppa nuove parti.

Quello che non ha capito l'induzione

- Anche quello che non ha capito l'induzione, però, ha un limite: crede che il principio di induzione dica

  “Sia n il numero di test che sarebbe opportuno condurre sul codice realizzato. Se il codice funziona sul primo test, allora funziona su tutti”
- Convinto di questo enunciato, testa il codice su un solo input e, se funziona su quello, consegna.
- I professori lo testano su venticinque input diversi, scoprendo che funziona solo su uno. Che sfortuna. Voto: 0

Quello che non ha capito l'induzione

Capitolo 2: dalla teoria alla pratica (metodo di sviluppo)

Problema

- Devi implementare un programma che necessita di tre funzioni f1, f2, f3, oltre alla funzione di lettura dell'input da file e al main.
- f3 richiama f1 ed f2.
- Tutte le funzioni operano su liste di interi, letti da file, e devono dare risultati diversi a seconda che la lista passata come argomento
  1. sia vuota
  2. non sia vuota
  3. contenga solo numeri pari
  4. contenga solo numeri dispari
  5. contenga più numeri pari che numeri dispari
  6. tutti gli altri casi

Cosa fate?

Risposta 1. Dimenticando i rischi che corre l'ottimista cosmico, scrivete tutto il codice **guardandovi bene dal compilare** di tanto in tanto.

Risposta 2. Ragionate, progettate, implementate e compilate, ma il primo run lo fate solo quando avete finito la stesura (e verifica della compilazione) di **tutto** il programma.

Cosa fate?

Risposta 3.

1. Ragionate, progettate, implementate e compilate per prima la **funzione che legge da file**, creando una primissima versione in cui stampate su standard output ogni elemento letto dal file, invece che inserirlo nella lista, per essere sicuri che la scansione degli elementi del file sia corretta.

2. Ragionate, progettate, implementate e compilate la **funzione di inserimento di un intero nella lista** e verificate il suo funzionamento facendone diverse chiamate dal main, con interi diversi, e verificando mediante una stampa della lista che la funzione operi correttamente.

3. **Raffinate la funzione di lettura da file** in modo che gli elementi letti vengano inseriti nella lista, usando la funzione “addElem” implementata al passo 2. Testate la funzione facendo una stampa del contenuto della lista letto.

Cosa fate?

Risposta 3, continuazione.

4. Ragionate, progettate, implementate e compilate **f1** e la testate su almeno 6 input diversi, corrispondenti ai 6 casi diversi che f1 deve prendere in considerazione. Ancora meglio se la testate su 12 o 18 input....

5. Stessa cosa con **f2**.

6. Infine Ragionate, progettate, implementate e compilate **f3**, che fa uso di f1 ed f2 (ma a questo punto, siete ragionevolmente certo che f1 ed f2 funzionino, quindi se riscontrate degli errori saranno in f3), e la testate su almeno 6 input diversi.

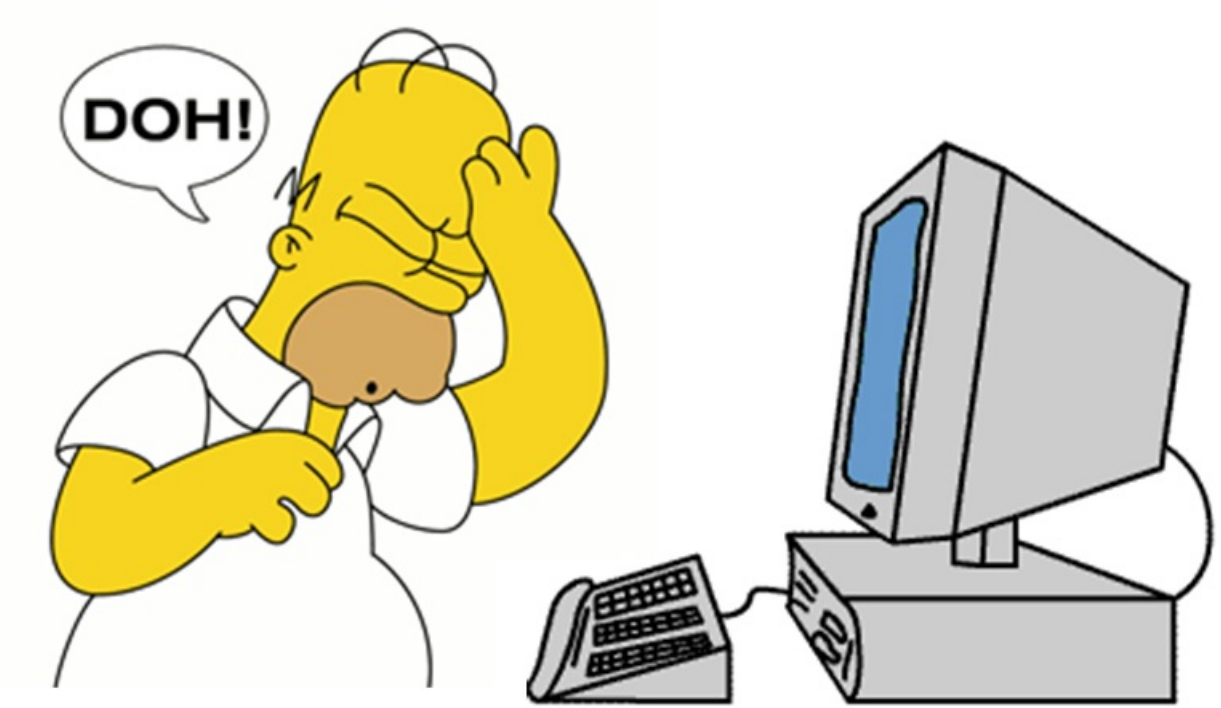
7. Implementate il **main** definitivo rimuovendo le chiamate di debug alle varie funzioni che avete implementato e lo testate adeguatamente su un numero ragionevole di casi.

Risposta 1

Risposta 2

Risposta 3

Capitolo 3: quello che sarebbe bello non vedere



In laboratorio sarebbe bello **non** vedere...

- Studenti che stanno già lavorando alla funzione xxx e il codice della funzione yyy che hanno “archiviato” è sbagliato, persino dal punto di vista sintattico (ottimisti cosmici)
- Studenti che stanno già lavorando alla funzione xxx e non hanno mai fatto un run per testare la funzione yyy: magari compila, ma è errata (quelli che credono ancora a P. Nocchio)
- Studenti che stanno già lavorando alla funzione xxx e non hanno testato la funzione yyy in modo adeguato: se xxx fa una chiamata a yyy e quest'ultima non è stata testata a sufficienza, diventerà molto difficile capire se l'errore sia in xxx o in yyy (quelli che non hanno capito l'induzione)

Assenza di fattorizzazione

- Quando scrivete un programma e vi accorgete di ripetere lo stesso gruppo di istruzioni più e più volte, magari in più funzioni diverse, allora è il caso di fare **refactoring**: il gruppo di istruzioni ripetute va trasformato adeguatamente un una funzione e al posto del gruppo di istruzioni dovete mettere una chiamata alla funzione.
- Esempio 1: funzione swap negli algoritmi di ordinamento. Viene chiamata talmente tante volte, che sarebbe davvero poco opportuno non incapsulare le operazioni di swap dentro una singola funzione.
- Esempio 2: getNode negli alberi generici: in molte occasioni ci serve recuperare il puntatore a un nodo data la sua etichetta: implementiamo una funzione dedicata a questo scopo, la testiamo adeguatamente e poi chiamiamo quella tutte le volte che ci serve

Codice privo di commenti, nomi poco significativi

- Il codice va commentato: voi stessi, riprendendo del vostro codice sviluppato qualche mese prima, non sarete in grado di capire cosa fa se avete dato alle costanti, ai tipi, alle funzioni e alle variabili nomi poco significativi e se non avete adeguatamente commentato il vostro lavoro

Codice privo di commenti, nomi poco significativi

```
Pollo didi::di(const Tra t, const
Capolavoro& dai){
  if (a(dai))
    return inconsupertrafra;

  if ((dai->quelo).lampada == t)
      return (dai->quelo).comodino;

  if ((dai->quelo).lampada > t)
      return di(t, dai->uno);
  else
      return di(t, dai->due);
}
```

```
Value dict::search(const Key k, const
Dictionary& d){
  // albero vuoto: impossibile trovare
  if (isEmpty(d))
    return emptyValue;

  if ((d->elem).key == k)
      return (d->elem).value;

  if ((d->elem).key > k)
      // l'elemento cercato ha chiave <
della chiave cercata, scendo a sx con una
chiamata ricorsiva su d->leftChild
      return search(k, d->leftChild);
  else
      // l'elemento cercato ha chiave >=
della chiave cercata, scendo a dx con una
chiamata ricorsiva su d->rightChild
      return search(k, d->rightChild);
}
```

Codice contorto

- E' difficile fare un esempio generico, comunque codice contorto, ridondante, in cui è difficile seguire la logica dell'algoritmo, è specchio di una scarsa comprensione dell'algoritmo stesso e degli strumenti linguistici per implementarlo.
- Anche l'indentazione è importante per rendere il codice leggibile e chiaro.

Codice contorto

...Però se riuscite a implementare un qualunque laboratorio correttamente e ci consegnate un codice artistico come questo... vi diamo un punto in più!

C'é anche una gara di offuscamento di codice: http://www.ioccc.org/

Codice in cui vengono inutilmente esposti dettagli implementativi

Disseminiamo il file header di dichiarazioni di costanti quali

```
const Elem emptyElem = "$#$#$";
```

```
const List emptyList = NULL;
```

perché nell'implementazione delle operazioni, non vogliamo usare NULL o "$#$#$", che sono “dettagli implementativi”, ma emptyLabel ed emptyTree, che sono due costanti del tipo di dato Label e Tree rispettivamente, e si collocano quindi ad un maggiore livello di astrazione.

Sarebbe bello non vedere NULL e valori specifici nel vostro codice, ma solo i nomi delle costanti dichiarate nell'header.

C'é una ragione per questo suggerimento: se cambiamo l'implementazione e ad esempio i dati contenuti in una lista non sono più stringhe ma interi, possiamo cambiare la riga dell'header

const Elem emptyElem = -1;

(oltre ovviamente al typedef di Elem, che da string deve diventare int) e tutto il resto del codice, se su emptyElem ho fatto solo confronti di tipo == e !=, continuerà a funzionare senza cambiare nulla!!!! Mica male....

Non saremo sempre così fortunati, ma se usiamo tipi di base quali `int` e `string` e operazioni di confronto molto elementari quali == e !=, probabilmente riusciamo a cambiare l'implementazione apportando modifiche minimali circoscritte all'header file.

Funzioni booleane maltrattate

Il codice

```
if (a==b) /* o qualunque altra espressione booleana */
   return true;
else
   return false;
```

è sbagliato, anche se “fa quello che deve fare”.

Lascia supporre qualche incertezza nella comprensione di cos'è un'espressione e cosa fa lo statement “return”. Inoltre, è inutilmente inefficiente.

Un frammento di codice come questo va riscritto in

return (a==b); `/* o qualunque altra espressione booleana */`

Funzioni che dovrebbero restituire qualcosa e non lo restituiscono...

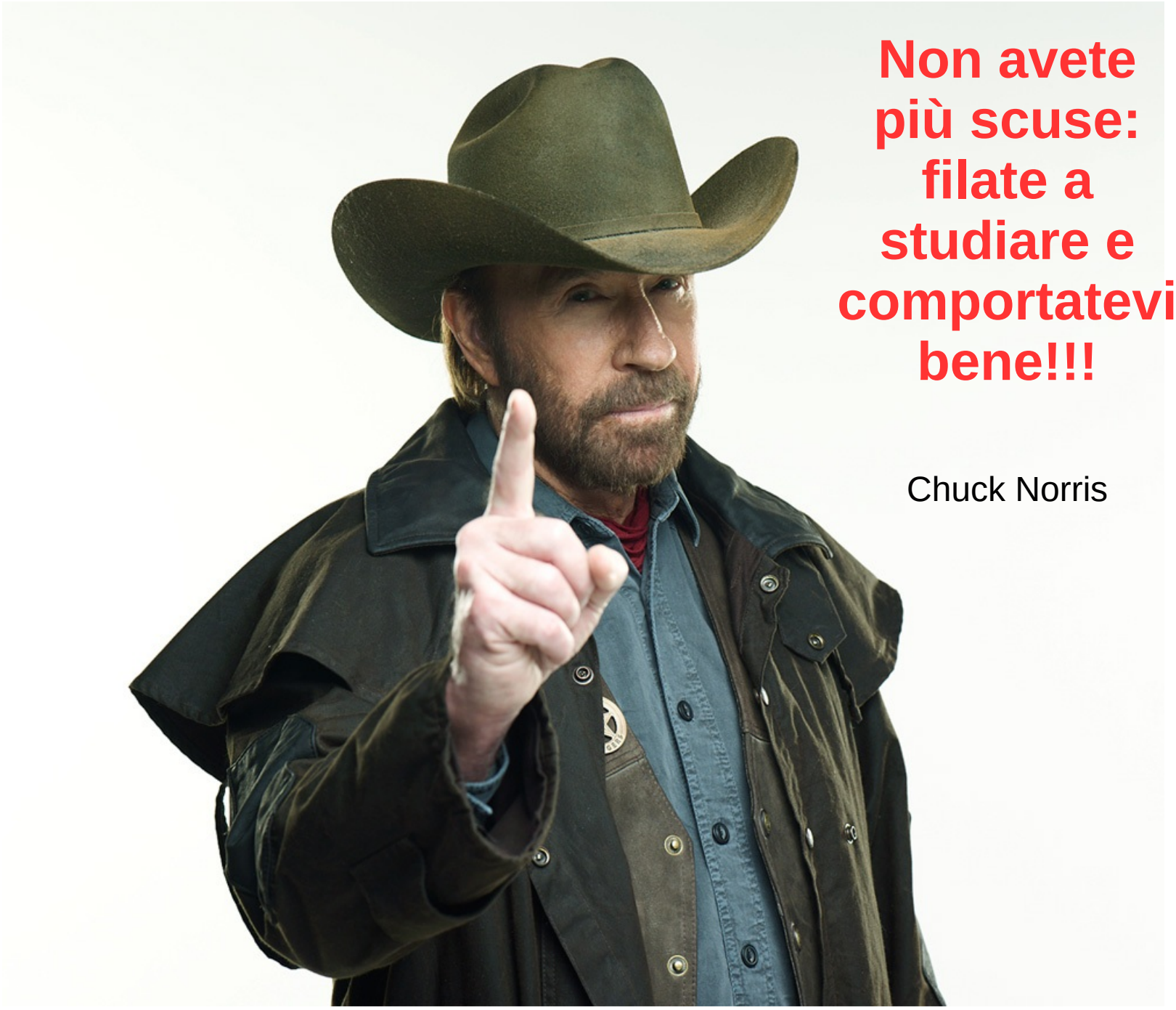
```
/* isThereAnyStringStartingWithCh restituisce l'etichetta del primo
nodo dell'albero (primo rispetto alla visita che si fa) la cui
etichetta inizia con il carattere ch */

Label tree::isThereAnyStringStartingWithCh(const char ch, const Tree&
t)

{

// se l'albero t e' vuoto, restituisco emptyLabel

    if (isEmpty(t))

        return emptyLabel;


// se l'etichetta di t inizia con ch, la restituisco

    if ((t->label).size() > 0 && (t->label)[0] == ch)

        return t->label;
```

Funzioni che dovrebbero restituire qualcosa e non lo restituiscono...

```
// Chiamata ricorsiva di isThereAnyStringStartingWithCh su ciascuno dei figli di t, finché una delle chiamate non restituisce un valore diverso da emptyLabel oppure finché non ci sono più figli
    Tree auxT = t->firstChild;
    while (auxT != emptyTree) {
        Label l = isThereAnyStringStartingWithCh(ch, auxT);
        if (l == emptyLabel)         // non ho trovato cercando in questo sottoalbero, devo proseguire la scansione dei fratelli
           auxT = auxT->nextSibling; // auxT punta al fratello successivo, per proseguire la scansione
        else                         // ho trovato: restituisco l'etichetta restituita da isThereAnyStringStartingWithCh(ch, auxT)
           return l;   // se questo return nel ramo else non c'é, non funziona niente!!!!
    }
return emptyLabel; // se esco dal while vuole dire che al termine di una ricerca esaustiva nell'albero t non ho trovato nessun nodo la cui etichetta inizia per ch; devo restituire emptyLabel; se non metto questo return, non funziona niente!!!
}
```

Per concludere...

- Sarebbe bello che imparaste (prendendovi il tempo necessario... ma non un tempo infinito!) a mettete in pratica questi principi di buona programmazione: https://www.artima.com/weblogs/viewpost.jsp?thread=331531
- Il momento giusto per iniziare....?


Non avete più scuse: filate a studiare e comportatevi bene!!!
Chuck Norris