1 Tecniche di programmazione

In questo capitolo tratteremo delle tecniche usate per la scrittura del software, delle diverse caratteristiche specifiche e di alcuni trucchi della programmazione in Assembly X86.
Nella parte finale dedicheremo l'attenzione alla scrittura di software ben documentato. Molte delle idee delineate in questa seconda parte sono applicabili allo sviluppo del software il generale, non solo alla programmazione Assembly.

Alcuni ignobili trucchi ("tricky programming")
L'estrema libertà resa possibile dall'Assembly rende possibili trucchi di tutti i tipi. Nel seguito ne illustriamo qualcuno, con l'avvertenza che, essendo questi trucchi molto oscuri a comprendersi andranno spiegati molto bene, nei commenti del codice e nell'altra documentazione del software.

Test se un registro è uguale a zero
Supponiamo di dover verificare se un registro è uguale a zero. Confrontiamo la soluzione "standard", più leggibile, con un trucco più efficiente:

Standard	L. m.	Trucco	L. m.
CMP SI, 0	83FE00	OR SI, SI	0BF6
JE EraZero	7400	JZ EraZero	7400

Con l'istruzione OR SI, SI il contenuto di SI non cambia, i flag sì, per cui successivamente, quando si fa la jump essa è identica nei due casi. La OR non chiede l'operando, che occupa un Byte a zero nella soluzione standard e quindi è un Byte in meno. Dato il "risparmio" esiguo la versione standard è migliore, a meno che non si debba scrivere un programma estremamente compatto.

Da numeri con segno a senza segno
Se un numero con segno è positivo, può essere memorizzato in una "variabile" senza segno, se è negativo non può, perché come numero senza segno sarebbe solo un numero "grande" (1 nel bit più significativo)
Nel programma seguente si usa un trucco analogo al precedente:

```
    ..
    ; AL contiene un numero con segno da "convertire" a numero senza segno:
    OR AX, AX    ; (questa per essere sicuro che i flag siano relativi
                 ;  al numero che c'è in AX, in un programma "normale"
                 ;  questa istruzione potrebbe non esserci, se sappiamo da
                 ;  dove vengono i flag)
    JS  Enegativo; se il flag di segno è ON, AL contiene un numero negativo
    ; il numero è positivo e lo copio nella "variabile" senza segno di
    ; destinazione:
    MOV[NumSenzaSegno], AX
    ..
    ; qui il programma prosegue ..

Enegativo:
    ; se il numero è negativo o faccio il casting (promozione) ad una
    ; "variabile" di 32 bit (vedi istruzione CWD) o segnalo errore
```

Copiare un bit in un Byte
Mettere uno zero nel bit dove si vuole copiare, poi fare una OR (facendo attenzione ai bit che non interessa ricopiare). Per esempio, se si vuole copiare il bit 2^3 di BL in AL:

```
    AND  AL, 11110111b   ; faccio un "buco" dove voglio ricopiare
    AND BL, 00001000b    ; lascio in BL solo il bit che mi
                         ; interessa copiare (distruggo BL!)
    OR AL, BL        ; copio il bit in AL, gli altri bit rimangono uguali a prima
```

Se si è a corto di registri e non si vuole distruggere il valore di BL, si può usare questa soluzione "di forza bruta":

```
    AND AL, 11110111b    ; faccio un "buco" dove voglio ricopiare
    TEST BL, 00001000b   ; guardo quanto vale il bit che mi interessa
                         ; (TEST non modifica BL!)
    JZ CeraZero      ; se in BL c'era zero, lascio AL com'è,
                     ; perché ho già messo 0 con la AND
    OR AL, 00001000b ; se in BL c'era uno, lo metto al suo posto
CeraZero:
    ..
```

Quest'ultima soluzione genera un codice un po' più lungo, ma non costringe a memorizzare BL, inoltre è decisamente più veloce sulle CPU più nuove, che possono eseguire più istruzioni in un unico ciclo di clock, dato che non blocca l'esecuzione in parallelo delle istruzioni, come accade nelle versione precedente .

Copiare il valore di DS in ES

```
PUSH DS
POP ES
```

non impegna registri ed è più sbrigativo di:

```
MOV AX, DS
MOV ES, AX
```

La seconda soluzione è più autoesplicativa, la prima non usa registri e può essere utile in situazioni come una ISR. Inoltre la prima è una soluzione più lenta, dato che fa due accessi alla memoria. Dunque in casi generali si dovrebbe preferire la seconda soluzione.

Lettura dell'indirizzo del programma già caricato in memoria:

```
    CALL FAR PTR ProprioQuiSotto
ProprioQuiSotto:
    POP AX   ; in AX IP dell'etichetta ProprioQuiSotto
    POP DX   ; in DX CS dell'etichetta ProprioQuiSotto
```

LOOP con il numero massimo di iterazioni

```
    MOV CX, 0
PerdoTempo:
    NOP
LOOP PerdoTempo
```

Contatori a modulo, con n potenza di 2

Un contatore a modulo n, in Elettronica come in Informatica, è un contatore che fa n conteggi. Parte da zero, si incrementa e ricomincia a contare da zero quando ha raggiunto il valore n – 1. In un programma normalmente si realizza con l'operazione di divisione intera (detta anche "modulo", appunto).
In Assembly si potrebbe usare una DIV. Se però n è una potenza di due si può usare un trucco che permette di realizzare il contatore con una sola AND, molto più veloce.
Per farlo basta mascherare in AND i bit più significativi, quelli che devono essere sempre a zero nel conteggio.
Supponiamo per esempio di voler contare da 0 a 7, poi ricominciare da zero (contatore modulo 8). Ogni volta che incrementiamo il contatore facciamo anche qualcosa del genere:

```
    INC [Conteggio]
    AND [Conteggio], 00000111b    ; quando Conteggio è 8 diventa 0,
                                  ; per cui si passa da 7 a 0
```

ATTENZIONE: non provare ad applicare questo trucco in linguaggi ad alto livello con numeri floating point, la rappresentazione è tutta diversa e si farebbero solo danni!. Attenzione anche al fatto che questo trucco non funziona neppure con i numeri interi con segno, se sono negativi.

Salti veloci

Il salto più veloce è quello che non si fa.
Per avere la massima velocità bisognerebbe organizzare il programma in modo che il caso più probabile delle jump sia quello che non fa il salto, cioè quello che rende falsa la condizione. Questo perché proseguire è più veloce che saltare.

1.1 Sviluppo del software

1.1.1 Documentazione dei programmi

Nomi

I nomi delle "variabili" e delle etichette saranno lunghi quanto serve ed esplicativi. Tutti gli assembler moderni non hanno "praticamente" limite al numero di caratteri dei simboli dell'utente.
Il nome di un simbolo deve sempre comunicare esattamente la ragione per cui quel simbolo è stato introdotto, per cui vale la pena di spendere qualche istante in più per cercare un nome significativo, piuttosto che ricorrere al repertorio dei cartoon (il programmatore che usa nomi di "fantasia" come Pippo, Pluto, Minnie in programmi "veri" probabilmente non farà quel mestiere per molto tempo!).
Ricordando che le funzioni sono nel programma "ordini", che "fanno" qualcosa, i nomi delle procedure dovrebbero contenere "verbi". I nomi delle "variabili" invece fanno riferimento ad "oggetti" e quindi di solito sono sostantivi.
Esempio:

```
..
LetturaA/D DB ?
NumeroDeiBlank DW ?
..
```

```
LeggiA/D PROC
..
ContaIblank PROC
..
```

Parametrizzazione

Un programma ben "parametrizzato" non ha solo parametri ben pensati nelle sue procedure, ma ha anche costanti simboliche che lo rendono flessibile e rapidamente adattabile a nuove esigenze.
Bisognerebbe sviluppare un'avversione istintiva per i numeri fissi.
Il buon programmatore dovrebbe abituarsi a scorrere regolarmente il programma che sta scrivendo e sostituire i numeri fissi con simboli autoesplicativi.
Vediamo un esempio non parametrizzato:

```
    ..
    ; cerco un blank nella stringa, se non lo trovo
    ; sommo 32 al codice ASCII che ho trovato:
    MOV CX, 32
32giri:
    CMP [BX], 32
    JE NonSommare
    ADD [BX], 32
NonSommare:
    LOOP 32giri
    ..
```

e confrontiamo con la versione parametrizzata:

```
MaxElementiStringa EQU 32
ASCIIcerco EQU 32
NumeroDaSommare EQU 32
    ..
    ; cerco un ASCIIcerco nella stringa, se non lo trovo
    ; sommo NumeroDaSommare al codice ASCII che ho trovato:
    MOV CX, MaxElementiStringa
GiraPerTuttaLaStringa:
    CMP [BX], ASCIIcerco
    JE NonSommare
    ADD [BX], NumeroDaSommare
NonSommare:
    LOOP GiraPerTuttaLaStringa
    ..
```

Confrontando le versioni si vede subito che la seconda è molto più comprensibile. Inoltre se si deve cambiare qualcosa, per esempio il carattere che si sta ricercando da blank diventa Esc (codice ASCII 27), nella seconda versione basta cambiare ASCIIcerco e mettere 27, mentre nella prima bisogna studiarsi di nuovo tutto il programma.

Scrivere per il riutilizzo

La parametrizzazione e la generalizzazione portano a scrivere programmi e procedure che si possono utilizzare anche in altri programmi.
Di solito ciò non comporta grosse perdite di velocità, mentre spesso richiede un certo sforzo "intellettuale" per dover analizzare più a fondo il problema.
Il lavoro su un programma non dovrebbe essere finito fino a quando non esistono costanti non parametrizzate o qualunque altra cosa che non sia codice che non possa essere facilmente modificabile.

Documentare il software

Source code has two equally-important functions: it must work, and it must clearly communicate how it works to a future programmer or the future version of yourself. (Ganssle)

Commenti

I commenti non devono spiegare cosa fanno le istruzioni, perché si può supporre che chi legge il sorgente conosca il linguaggio di programmazione in cui è scritto.
I commenti, ed in generale tutta la documentazione di un software, debbono essere pensati per spiegare *come* funziona la soluzione scelta e *perché* le cose vengono fatte *così*. Devono essere concisi ed espliciti, ma anche completi, devono cioè dire tutto quello che serve.

Esempio:

```
    MOV AX, 1; ~~Sposto uno in AX~~. ; commento al commento: è sbagliato
```

(Pensiero di chi legge: "Grazie! Lo sapevo già che sposti uno in AX!").
Questo è un commento inutile e dannoso perché indispettisce chi legge e non lo aiuta a capire a cosa serve AX).

```
MOV AX, 1 ; inizializzo il registro in cui accumulerò la produttoria degli elementi
```

(Pensiero di chi legge: "Bene. Ora so a cosa servirà AX da qui in avanti!")

oppure (e forse meglio):

```
MOV AX, 1 ; in AX la produttoria, inizializzazione
```

Nei commenti non usare tutto maiuscolo o tutto minuscolo, perché peggiora la leggibilità del codice. Molto raramente scrivere tutto in maiuscolo, solo per le cose molto importanti. Comportarsi come se scrivendo tutto in maiuscolo si stesse URLANDO.
Separare con linee bianche le parti di codice che fanno cose diverse.
Supponendo che chi legge abbia i rudimenti del linguaggio, non è necessario commentare tutte le linee di codice, ma piuttosto ogni piccolo blocco funzionale.
Commentare ogni istruzione che non sia perfettamente comprensibile, chi legge il vostro codice non ha tempo da perdere ad interpretare perché avete lavorato in quel modo! In particolare spiegare ogni "trucco" usato ed il perché .

Se si usano delle fincature, che rendono il sorgente più chiaro ed esteticamente più piacevole, evitare di obbligare chi modifica il codice a perdere tempo per rimettere in tutto in colonna:

```
;************************************************************
;* Se cambio la lunghezza di questo commento devo riempire    *
;* di blank se voglio allineare a destra le stelline          *
;************************************************************

;************************************************************
; In questo caso posso fare il commento largo come voglio
;************************************************************
```

Per maggiore comodità nelle modifiche dei programmi le linee di codice o di commento non dovrebbero essere più lunghe del numero di caratteri che sono visibili nella finestra dell'editor, quando essa è massimizzata.
Non scrivere comunque linee più lunghe di 75 - 80 caratteri, a meno che ciò non sia inevitabile. Questo perché le stampanti "a caratteri" si comportano in modo strano quando si spediscono linee più lunghe di 75 – 80 caratteri; alcune forzano un'andata a capo, altre addirittura non stampano i caratteri in più.

L'uso del tasto "Tab" può dare problemi se chi modifica il programma usa un editor diverso. Se si ha pazienza, non usare i Tab, comunque non esagerare con il loro uso.

In Assembly non serve a molto indentare il codice dentro i cicli, perché esso è intrinsecamente non strutturato, almeno "localmente".
Serve invece mettere in evidenza le etichette. Per capire come funziona un programma Assembly bisogna saltare alle etichette, che devono perciò essere molto visibili.
Per rendere visibili le etichette scrivere le istruzioni separate dal margine sinistro del sorgente, le etichette e le direttive sul margine sinistro.

Non usare commenti che prendono più di un'istruzione:

```
MOV [b], 2   ; Inizializzo il valore di b a 2 perché
XOR CX, CX   ; deve partire da un numero pari
```

Se prima di XOR CX, CX diventa necessario inserire una nuova istruzione si deve pasticciare parecchio con l'editor per rimettere tutto a posto.
Se invece il programma fosse stato scritto così:

```
MOV [b], 2   ; Inizializzo il valore di b a 2 perché
             ; deve partire da un numero pari
XOR CX, CX
```

L'aggiunta di un'istruzione in mezzo sarebbe stata un'operazione indolore.

Scrivere il codice in modo omogeneo
Definire uno "scheletro" di un modulo di programma, dal quale partire quando si comincia a scrivere un nuovo modulo.
In Inglese questi schemi di riferimento vengono detti "**template**".

Un template aiuta a:
- non dimenticarsi di ciò che bisogna fare
- sapere subito dove trovare le informazioni che servono, anche in programmi scritti da altri che però usano la stessa convenzione
- avere lo stesso aspetto grafico per tutti i moduli che compongono il programma.

Le cose che non sono di interesse per l'applicazione corrente si cancelleranno dal template.

Vediamo un esempio di template per programmi in TASM a MASM:

```
;************************************************************************
; Nome del programma: XXXX
; Nome del file: XXXX.ASM
; Versione: XXXX.XXXX.XXXX
; Data revisione: XXXX.XXXX.XXXX
;
; ---------------------------------------------
; Funzionalità del programma:
;  XXXX
; ---------------------------------------------
; Elenco dei file che costituiscono il programma:
; (Nome           Descrizione)
; XXXX.XXXX     XXXX
; -------------------------------------------------
; Storia delle revisioni (in ordine cronologico inverso):
;  Revisione: XXXX.XXXX.XXXX ------------- (es. 2.0.0)
;    Modificato da: XXXX
;    Data: XXXX.XXXX.XXXX
;    Descrizione delle modifiche:
;      XXXX
;************************************************************************
DATA SEGMENT

DATA ENDS
ASSUME CS:CODE, DS:DATA, SS:STACKlocale
CODE SEGMENT
InizioProgr:
    ; sistemazione DATA SEGMENT:
    MOV AX, SEG DATA
    MOV DS, AX
    ; sistemazione STACK SEGMENT:
    MOV AX, SEG STACKlocale
    MOV SS, AX
    ; sistemazione stack pointer:
    MOV SP, OFFSET InizioStack

    !!!! qui il resto del codice !!!!

    ; terminazione programma
    MOV AH, 4Ch
    INT 21h
CODE ENDS
STACKlocale SEGMENT
  FineStack DB XXXX DUP (?)
  InizioStack Label Word
STACKlocale ENDS
END InizioProgr
```

Template per le procedure:

```
XXXX PROC
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Scritta da: XXXX
; Data ultima modifica: XXXX.XXXX.XXXX
;   ------------------------------------------------------------------------
;   Descrizione:
;     XXXX
;   (descrizione della funzione svolta dalla procedura e di come la si usa)
;   ------------------------------------------------------------------------
;   Parametri passati attraverso:
```

```
;      Ingressi: registri | memoria | stack CANCELLARE QUELLI CHE NON !!!!
;      Uscite  : registri | memoria | stack CANCELLARE QUELLI CHE NON !!!!
;
;      Ingressi: ----------------------------
;    XXXX (significato dei parametri in ingresso)
;
;      Uscite: -----------------------------
;    XXXX (significato dei parametri in uscita)
;
;      Registri modificati: -----------------
;        XXXX (registri che la procedura lascia modificati)
;
;      Effetti collaterali sulla memoria: ---
;    XXXX (gli effetti che lascia su strutture dati in memoria)
;
;    ------------------------------------------------------------------------
;    Note sull'uso:
;       (particolarità nell'uso, casi in cui non funziona, altri effetti
;        collaterali)
;      XXXX
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;

!!!! scrivere qui il codice della procedura !!!!

XXXX ENDP
```

Si possono pensare anche righe di commento speciali, destinate a documentare sempre allo stesso modo operazioni particolari.
Per esempio:

```
;    RRRR da qui uso il registro XX per XXXX
```

Questo commento può servire a dare senso al contenuto di un registro, che non si può mai capire dal suo nome. Se ci si dimentica a cosa serve DX si può cercare all'indietro la stringa RRRR e si trova facilmente questo commento.
Alcuni editor "per programmatori" hanno la possibilità di definire template che possono essere adattati ed inseriti nel codice con la pressione di un singolo tasto.

Darsi uno standard per il processo di sviluppo

E' indispensabile darsi uno standard per l'organizzazione, specie quando i programmatori che lavorano ad un progetto sono molti.
In un ambiente con molti programmatori i loro computer sono collegati in rete locale, per cui si dovrà organizzare una struttura comune dei directory, stabilire convenzioni sui nomi dei file e su come comunicare agli altri che una nuova versione del nostro lavoro è stata "pubblicata" in questi directory comuni.

Scrittura del programma

Durante la stesura del programma ci sono sempre momenti in cui, per fretta o per pigrizia, non si può o non si vuole cercare una soluzione completa ad un problema oppure si deve scrivere una parte di codice che si sa bene che funzionerà solo in casi speciali.
In quei momenti però si è focalizzati nella soluzione di un altro problema, per cui non si vuole interrompere quel processo.
In questi casi è necessario lasciar traccia nel sorgente dei problemi incontrati, in modo da ritrovare rapidamente i punti ove c'è ancora da lavorare.
Un modo può essere inventare un "simbolo" particolare, per esempio quattro punti esclamativi !!!!, e scriverlo nei punti ove ci sono errori da correggere o parti di codice da aggiungere o migliorare. Un altro simbolo, per esempio quattro punti interrogativi, potrebbe essere usato nei punti in cui la soluzione è da provare più approfonditamente, perché ci sono dei dubbi.
Risolti i problemi si tolgono le !!!! e ???? e non si considera finito il programma fino a che ne esistono ancora delle altre.
Esempio:

```
    ..
    XOR AX, AX
Ciclo:
    INC AX

    .. qui mancano tre o quattro pagine di programma ..

    INC AX
    ' ???? non ho capito perché qui AX è sempre pari!
```

```
    ' !!!! per ora ho risolto cosi': !!!!
    DEC AX
    ; !!!! con questa DEC il programma funziona, ma bisogna
    ; !!!! capire perché e correggere
    LOOP Ciclo

    ; altro esempio:

    ..

    JO Errore

    ..
Errore: ; !!!! ancora da fare: avvertire l'utente dell'errore
        ; prima di uscire
    JMP EsciDalProgramma
```

To do list
E' una buona idea scrivere nel sorgente, o tenere in un file separato, una lista degli interventi da fare sul codice, eventualmente corredata da una stringa univoca che permetta di ritrovare subito sul codice il punto in cui intervenire.

Notorious bugs
Quando si rilascia un programma esso sarà stato esaurientemente collaudato, per cui non avrà errori evidenti.
Subito dopo il rilascio gli utenti cominceranno a trovare nuovi errori e li comunicheranno.
Per alcuni errori minori si potrà decidere di non emettere una nuova versione del programma, pubblicando una lista degli errori conosciuti, che verranno eliminati nella versione successiva.
L'utente, consultando quella lista, potrà evitare di far cadere il programma in quegli errori.

Versioni

Durante la "vita" di un programma esso sarà sottoposto a revisioni, aggiunte, modifiche.
Sarà necessario sviluppare una convenzione per la numerazione delle **versioni**.

Numerazione delle versioni

Di solito la numerazione delle versioni è costituita da tre numeri:

MajorRelease.Version.Build (es. 1.2.483)

- MajorRelease cambia quando si fanno ristrutturazioni fondamentali del programma, si aggiungono molte nuove funzioni, o si cambiano alle fondamenta quelle esistenti.
- Version (o "minor version") cambia quando ci sono modifiche significative rispetto alla versione precedente o quando sono stati corretti molti errori. Le versioni "bug fix" correggono solo gli errori, senza modificare le funzionalità
- Build cambia ogni volta che si mette insieme il programma e lo si compila, serve per tracciare le modifiche minori, al limite ad identificare il giorno in cui è stato introdotto un bug nel programma. Spesso i programmi non comunicano all'utente il numero di build, perché se si introducono nuove funzionalità interessanti per l'utente si modifica anche il numero di versione.

Se la MajorRelease è zero, ciò significa che il programma non è ancora considerato "finito" ed è una versione di sviluppo.
Nei numeri di versioni del S.O. Linux è stata adottata questa convenzione: se il numero di minor version è pari esso è la versione stabile, "di produzione", da usarsi da parte degli utenti normali. Se esso invece è dispari, la versione del kernel è sperimentale, e viene usata dai "kernel hacker" per lo sviluppo delle funzionalità del S.O. .
Una specifica versione di un programma distribuita agli utenti (released, "rilasciata") viene anche detta "release".

Tracciare la "storia" dell'applicazione

Le varie versioni del programma potrebbero sviluppare incompatibilità, per cui è possibile che si debba tornare al codice di una vecchia versione per correggere un errore che si è presentato tardi.
Per questo deve esistere un documento ben aggiornato che illustra tutte le versioni prodotte e/o rilasciate al cliente e bisogna salvare il sorgente di ciascuna versione.

Tracciare la versione nel programma

Per ogni evenienza è meglio scrivere il numero della versione anche nel codice, in modo che, male che vada, guardando l'EXE si possa capire di quale versione si tratta.
Esempio:

```
Versione$ EQU "0.01"
    VersioneProgramma DB "Inutile: programma che non fa niente. Versione ", Versione$
```

VersioneProgramma viene scritta nell'area dati con il numero di versione. Con un debugger o un editor esadecimale si potrà sempre .leggere.
Per modificare la versione basta cambiare la stringa in EQU (la versione cambierà spesso, qui è d'obbligo parametrizzare!).
Può essere una buona idea far scrivere la stringa VersioneProgramma quando il programma parte, così che l'utente o il tecnico che deve verificare un malfunzionamento può sapere che versione del software sta usando.

Uso di strumenti per il controllo delle versioni
Esistono programmi sofisticati che aiutano a tenere sotto controllo le varie versioni dei programmi.
Queste applicazioni sono pensate per software di grandi dimensioni per cui tengono conto della quantità di memoria di massa occupata dai salvataggi delle diverse versioni. Quando questi programmi salvano le nuove versioni usano algoritmi di compressione e memorizzano solo le differenze rispetto alla versione precedente.

1.2 Debugging

Si può ben dire che l'abilità nel debugging è una delle caratteristiche più importanti di un buon programmatore.
Il debugging è anche una delle attività più difficili del programmare. Richiede un certo "istinto", che in realtà non esiste perché si tratta solo di esperienza. L'esperienza si sviluppa provando, e sbagliando, per cui il buon cacciatore di errori ha alle spalle diversi mesi-uomo di rabbia e frustrazioni e ed una cocciuta volontà di riuscire nel suo compito.

1.2.1 Non fare errori

Disciplina
Bisogna sempre rispettare le regole per lo sviluppo e la documentazione del software che l'azienda per cui si lavora ha deciso di applicare. Le Aziende non fanno queste regole a cuor leggero, perché la documentazione e la rintracciabilità delle versioni per loro rappresentano un costo, per cui il fatto solo che quelle regole esistano significa che servono!

Istinto del rischio
Con l'esperienza e gli errori commessi si acquisisce un "sesto senso" che aiuta a fiutare le soluzioni "rischiose", quelle in cui il programma sarà nel maggior pericolo di non funzionare.
Se il tempo a disposizione è poco, evitare ogni rischio.
Spesso però prendere strade un po' rischiose può dare vantaggi nel lungo periodo.
E' il caso di imbarcarsi in una situazione rischiosa solo se c'è più tempo a disposizione ed il gioco vale la candela. In questo caso bisogna pianificare delle vie d'uscita, o preparare una soluzione "semplice" da utilizzare nel caso che la via rischiosa porti ad un vicolo cieco.

1.2.2 Togliere gli errori

"C'è sempre un altro maledetto bug!". Nel campo del software non c'è frase più vera. La caccia agli errori è perciò una attività necessaria per chi sviluppa software, anche se spesso malvista.
Per le persone competitive, che ricercano delle sfide, trovare gli errori nel software è spesso un'attività gratificante, in particolare quando altri non ci sono riusciti.
E' difficile dire come si fa, ognuno, con molta fatica e la dovuta testardaggine, sviluppa la sua tecnica. Ciò significa che le tecniche individuali per il debugging possono essere anche molto diverse.

Comunque si può provare a dare queste indicazioni:
Ascoltare Giulio Cesare: "Divide et Impera"!
- Separare mentalmente il programma in tanti piccoli pezzi, nei quali si possano individuare abbastanza chiaramente ingressi ed uscite.
- Affrontare uno alla volta questi pezzi.
- Cominciare dall'avversario più pericoloso. Esaminare per prima, in grande dettaglio, quella parte di programma nella quale si ritiene che l'errore sia più probabile.
- Mettere alla prova questa parte, identificando se è responsabile del difetto.

Per dare la responsabilità ad una parte di codice è molto importante usare i breakpoint. Per decidere qual è la parte sospetta serve l'esperienza acquisita (errori simili fatti in precedenza) ed un'analisi attenta dei "sintomi", cioè del modo con il quale il malfunzionamento si presenta.
Sapendo cosa fa ogni pezzo del programma e vedendo cosa succede, spesso si può essere certi che l'errore accade in una parte ben specifica, ben delimitata, del programma.
In questo aiuta molto realizzare il programma in modo strutturato, ma in Assembly ciò non è possibile! Ci accontenteremo di dividere il programma in procedure, che potranno essere messe alla prova indipendentemente una dall'altra.

Imparare dove mettere i breakpoint
Mettere un breakpoint all'inizio della parte di codice sospetta. Mandare il programma a tutta velocità fino al breakpoint. Lì il programma si ferma. Ora si può esaminare attentamente cosa "entra" nel nostro blocco, cioè lo stato attuale dei re-

gistri e delle aree di memoria che interessano. Ci fermiamo a controllare attentamente tutti gli ingressi del blocco sospetto che abbiamo individuato.
Se ci sono già degli errori sugli ingressi, bisogna cercare il bug, in qualche punto che ha già eseguito. Per questo bisogna spostare il punto d'esecuzione all'indietro; spesso è necessario rilanciare il programma dall'inizio, dopo avere impostato un breakpoint anticipato rispetto al precedente.
Se invece tutto ciò che entra nella parte sospetta è corretto, passiamo ad eseguirla passo per passo, pensando a cosa dovrebbe succedere prima di eseguire ogni singolo passo e confrontando quello che si prevedeva con quello che si è ottenuto.
Ogni contraddizione potrebbe essere dovuta al nostro bug. Tutte le volte che ci si trova in contraddizione bisogna capirne la ragione. Mettendo alla prova tutti i risultati "strani" alla fine troveremo il bug e ne capiremo la causa. Sapendone la causa sarà facile eliminarlo per sempre.
Se la parte che sospettavamo di più risulta innocente, dovremo ipotizzare una nuova fonte del problema, mettere alla prova l'ipotesi, provando come già detto, e continuare fino a che il programma non funziona correttamente e con tutti gli input che dovrà accettare in condizioni operative, anche quelli più strani.
Una cosa da tenere in mente è che la causa del bug potrebbe essere anche molto lontana rispetto al momento in cui quel bug si manifesta. Perciò non si deve cercare l'errore solo nei dintorni del punto ove se ne rilevano i sintomi.

Watchpoint sulle condizioni, non sono mai usati abbastanza
Le contraddizioni, delle quali parlavamo prima, spesso si evidenziano con valori "strani" in locazioni di memoria o in registri. Per andare a caccia del punto in cui si producono questi valori strani sono estremamente utili i watchpoint. Un watchpoint è un'espressione che viene calcolata automaticamente dal debugger ad ogni punto in cui i suoi operandi cambiano. Se l'espressione diventa vera il programma viene interrotto.
Dunque mettendo un watchpoint che scatta quando una variabile assume un valore anomalo è possibile lasciar andare tranquillamente il programma, che si fermerà esattamente nel momento in cui accade il comportamento anomalo.
Non tutti i debugger possiedono la funzione di watchpoint.

Se non esiste il debugger
In alcuni casi si può essere costretti a verificare un programma anche se non si ha a disposizione un debugger.
Questo può succedere quando si utilizzano CPU non molto diffuse o quando si sviluppano "device driver", che debbono lavorare allo stesso "livello" del Sistema Operativo (kernel).
Se il debugger per la nostra applicazione non esiste od è troppo costoso, ci si deve arrangiare facendo in modo che il nostro programma "stampi" regolarmente risultati intermedi e l'indicazione del punto in cui sta eseguendo. La "stampa" può avvenire sul video o su qualsiasi altro dispositivo esterno che si possa controllare regolarmente, ed eventualmente salvare su file.
La lettura di questi risultati intermedi potrà dare indicazioni su cosa succede quando il programma non funziona.

Errori tipici
Un errore software può essere di tre tipi:
- Errore del programmatore
- Errore della documentazione delle funzioni utilizzate
- Errore del sistema di sviluppo o nel sistema operativo

Tutti almeno una volta hanno creduto di essere i migliori programmatori del mondo, ma non bisogna cadere nella presunzione.
L'ordine "di probabilità" con cui si presentano gli errori è proprio quello indicato qui sopra. E' inutile dare la colpa al sistema operativo se la nostra applicazione non funziona, il 99% delle volte è solo perché non abbiamo fatto un buon debugging (o un buon programma!).

Errori di sintassi
Gli errori di sintassi sono individuati dal compilatore, per cui di solito non ci sono problemi a toglierli.
Per fare più in fretta bisogna LEGGERE l'indicazione di errore data dal compilatore e RAGIONARE su cosa significa.

Loop infiniti
Sintomo: Il programma non finisce mai. Se lo si esegue da un debugger dovrebbe essere possibile interromperlo (per esempio con i tasti Ctrl – C o Ctrl - Bloc- Scorr), anche mentre è in una situazione di blocco apparente. Il punto dove si trova il programma al momento dell'interruzione è probabilmente in un ciclo che non finisce mai.
Di solito ciò accade per problemi nel controllo che deve far concludere il ciclo.
Se non si riesce a riprendere il controllo del programma attraverso il debugger operare una ricerca "binaria" del punto ove il programma si blocca. Mettere un breakpoint circa a metà del codice e lanciare il debugger a tutta velocità. Se si ferma al breakpoint il problema è dopo altrimenti è prima. Mettere un breakpoint a "metà" del codice dove è stato individuato il problema e ripetere, dividendo sempre per "due", fino ad individuare il punto dove c'è il loop infinito. Per capire come mai il loop non esce, tracciare il codice passo passo, analizzandolo con molta attenzione.

Stack non equilibrato
Il sintomo tipico è la morte del programma. Quando lo si esegue con il debugger si vede che esso si blocca al momento di una RET oppure, se si è fortunati, si vede che dalla RET non si torna all'istruzione che segue la chiamata, ma in un punto strano, non previsto. In questi casi si tratta quasi sempre di uno stack non svuotato, o troppo svuotato. Spesso si controlla che lo stack sia equilibrato solo nel "percorso normale" del programma, cioè quando si verificano tutte le con-

dizioni normali. Se vengono eseguiti dei casi particolari, che fanno scattare dei test che mandano direttamente alla RET, essa potrebbe eseguire "improvvisamente" senza fare le opportune POP.
Per esempio:

```
    PUSH AX
    ADD AX, BX
    JC ErroreCarry

    ..

    POP AX       ' questa POP NON viene eseguita se il programma prende la strada
                 ' dell'ErroreCarry. In questo caso lo stack è sbilanciato e
                 ' il programma si blocca irrimediabilmente quando giunge alla RET
                 ' Se la JC non viene "esercitata" durante la fase di test, questo
                 ' errore potrebbe essere rilasciato con la versione finale del programma

    ..

ErroreCarry:
    RET
```

Salti o chiamante near o far sbagliate
Questo è un caso particolare del precedente, quindi si presenta con gli stessi, drammatici, effetti.
Se si salta in una procedura FAR con una CALL NEAR o, viceversa, se si salta in una procedura NEAR con una CALL FAR, lo stack diviene non equilibrato e, presto o tardi, se ne subiranno le conseguenze.

Metterci delle pezze (patches[1])

Spesso nell'avvicinarsi della data di consegna di un software non c'è tempo per trovare un rimedio "giusto" ai problemi che emergono.
Allora ci si risolve ad aggiustarlo come si può, alla svelta, rimediando ai "sintomi" dell'errore perché non c'è tempo per cercarne le cause. In questi casi è necessario documentare ampiamente la correzione che si è fatta, indicando chiaramente che è "provvisoria". Ciò in modo che qualcuno possa andare alla ricerca delle cause vere del problema e curarlo alla radice, in un tempo successivo ed a mente fredda.

"un bug non è risolto fino a che non si capisce esattamente cosa sta succedendo e perché" (C. Pescio, in "Computer Programming")

Non ritardare il test dei programmi
E' meglio cominciare a provare le parti di un programma subito dopo averle scritte, senza attendere di avere tutta l'applicazione pronta.
In questo caso gli errori si presentano appena dopo aver scritto quella parte di codice, l'algoritmo utilizzato per risolvere il problema è ancora fresco nella nostra mente, possiamo modificarlo facilmente se ci rendiamo conto che non funziona.
Inoltre la prova precoce tiene gli errori costantemente sotto controllo e ci sarà meno probabilità che un errore in una parte del programma si ripercuota su altre parti.
Peraltro la prova precoce può significare un certo sforzo in più, perché bisogna organizzare il test di piccole parti di codice, creando variabili temporanee, piccole procedure di prova, ed altri strumenti per il test.

Tenere traccia degli errori commessi
Ogni programmatore dovrebbe avere un suo "libro degli orrori", in cui segnare i principali problemi incontrati nello sviluppo delle applicazioni e le soluzioni trovate. Dopotutto, con gli strumenti di taglia e incolla a disposizione di ogni computer, è un lampo copiare il codice "rotto" e la soluzione "aggiustata" in un file di testo, o meglio ancora in un database, con alcune chiavi per una ricerca più veloce. Questo naturalmente richiede metodo e pazienza, doti molto importanti per i programmatori, ma che non tutti possiedono in quantità sufficiente.
Una riguardata periodica al libro permetterebbe di non incappare più in errori già fatti. Purtroppo sono pochissimi quelli che tengono un tale registro anche se non sono costretti da regole aziendali (lo scrivente ammette di non essere nel bel numero). Sono invece in aumento le aziende che "costringono" i programmatori a tenere quel registro, anche per soddisfare i requisiti delle norme sulla Garanzia di Qualità.

Collaudi (test)

Durante i collaudi nelle prime fasi dello sviluppo applicare al programma i suoi ingressi tipici, quelli che ci si può aspettare in condizioni normali e con un utente accorto.
Fino a che il programma non funziona in condizioni tipiche, non è il caso di stressarlo troppo con casi particolari. Viceversa, quando il programma comincia a funzionare, bisogna passare alla "prova tortura", verificandolo con i peggiori ingressi che gli possono capitare e con i casi particolari più involuti.
Verificare anche il comportamento del programma in risposta a dati errati presentati dall'utente. Gli ingressi che di solito danno più problemi sono quelli agli estremi del loro campo di variabilità ed il valore zero.

1 il termine "patch" in Inglese significa "pezza", di stoffa.

Non considerare mai il programma finito se è stato provato con una sola sequenza di ingressi, anche se complicata. Provare invece con diverse sequenze perché l'ordine con il quale gli ingressi si presentano al programma potrebbe influire sul suo comportamento.
Se il programma è impostato con funzioni che hanno scopo e modalità di funzionamento chiare e ben definite la sua verifica è facilitata perché è facile stabilire i dati di ingresso alle funzioni in fase di test ed è facile immaginare quali sono il risultati da aspettarsi.
Per non dimenticarsi alcune prove importanti ci si può creare una lista delle cose più tipiche da controllare ("check list").

1.3 Manutenzione del software

Le applicazioni fortunate possono "vivere" parecchi anni, ma perché ciò sia possibile devono essere "restaurate" regolarmente.
Manutenere un programma è un compito complesso e delicato, che è molto agevolato se esso è scritto bene.
Mentre si scrive un programma bisogna prendere l'abitudine di "guardare avanti" pensando a cosa dovremo fare quando ogni parte del codice che scriviamo dovrà essere modificata o sostituita.
Seguono alcune raccomandazioni, che non sono dissimili da quelle appena fatte per la fase di sviluppo.

Ancora la documentazione!
Oltre a documentare il codice e gli algoritmi usati, documentare anche le modifiche effettuate.
Tenere traccia delle modifiche effettuate e salvare le vecchie versioni (questo può permettere di tornare indietro ad una versione sicura se alcune modifiche si sono rivelate troppo inaffidabili).

Modifiche
Collaudare ogni modifica, analizzare quali influenze può avere una modifica fatta in una parte di un programma sul funzionamento di altre parti non modificate.
Tenere traccia anche dei collaudi effettuati.

Strumenti di sviluppo
E' buona norma conservare la versione del software di sviluppo usata per produrre il programma (compilatore, debugger, ambiente di sviluppo). Infatti anche gli strumenti di sviluppo hanno nuove versioni e molto spesso le nuove non accettano il codice scritto per le vecchie.
Se non si conservano i vecchi strumenti di sviluppo può darsi il caso che, per togliere un piccolo errore in una vecchia versione, si sia costretti a pesanti modifiche, per l'adattamento alla nuova versione del compilatore, ed a nuovi collaudi.

Da ultimo si fa rilevare che, a differenza dell'hardware, con il software non è possibile applicare quella che J. Ganssle definisce "percussive maintenance" ovverosia "the fine art of whacking a device to get it working" (L'autore garantisce che la tecnica funziona con il telecomando della sua TV!).

Curiosità

L'esplosione di Ariane 5
Il razzo Ariane 5, dell'agenzia spaziale europea, era molto più potente dei precedenti ed era in grado di portare in orbita un carico pagante molto più pesante. Durante la sua prima missione, di prova e senza carico pagante, fu fatto esplodere in volo pochi istanti dopo la partenza perché aveva preso una rotta del tutto sbagliata.
Il problema che causò l'errore di rotta fu esclusivamente software.
La parte di programma che teneva traccia della posizione del missile non era stata modificata rispetto a quella di Ariane4, per cui i tecnici si fidavano di essa, dato che aveva volato con successo per molte decine di volte. Cionondimeno essi la provarono estesamente, insieme a tutto il nuovo software di Ariane5.
Il simulatore del razzo, che doveva fornire al software dati "realistici", fu pronto solo nella fase finale dello sviluppo e fu utilizzato solo per le parti del programma meno "mature". Dato che mancava il tempo, la parte di rilevazione della posizione fu rilasciata senza provarla al simulatore, dato che non presentava malfunzionamenti da anni.
Quando Ariane5 partì, i motori potenti del razzo lo fecero accelerare più velocemente di quanto Ariane4 avesse mai fatto e la variabile che teneva traccia della posizione del razzo andò in overflow .. BUM !! :-(.