Proposta di stile per i codici sorgenti

Paolo Caressa

Roma, maggio 2005

1 Introduzione

Queste brevissime note sono un contributo alla standardizzazione del codice sorgente fra i membri del nostro gruppo: non si tratta di mia farina ma di indicazioni prese dalla letteratura o dal folklore informatico. Parimenti non si tratta di regole migliori di altre, ma solo di regole a me familiari e che considero ragionevoli: ogni regola può andare bene, ed il principio fondamentale (metaregola) dovrebbe essere *qualsiasi stile va bene purché ci si attenga coerentemente ad esso.* La principale motivazione per l'adozione di un insieme di regole di questo tipo sta nel ridurre le scelte che un singolo programmatore deve fare, e quindi di standardizzare il codice ai fini della leggibilità e della sua mantenibilità.

Le regole e le convenzioni stilistiche che qui riporto sono grosso modo quelle nel quale è scritto il codice dei libri di Kernighan e Ritchie [7] e Hanson [2], e quelle consigliate nel volume di Kernighan e Pike [6] (qualcosa è anche ispirato dal libro di Horstmann [4]). Naturalmente qui faccio riferimento al `C`, ed alcune di queste regole sono già *built-in* in altri linguaggi: tuttavia si possono utilmente applicare anche a `C++`, `C#`, `Java`, `Matlab`, etc.

2 Struttura del programma: interfacce ed implementazioni

Il programma (con questo intendo semplicemente un pezzo di software, anche una `dll`) si presenta come un insieme di moduli che, alla bisogna,

comunicano fra di loro: un modulo esporta una interfaccia, che è tutto quello che un suo utilizzatore (*client*) deve conoscere di essa, e possiede almeno una implementazione, che soddisfa le specifiche dell'interfaccia.

In `C` l'interfaccia è contenuta in un file sorgente, l'*header*, con estensione `.h`: questo file deve *contenere esclusivamente* dichiarazioni di costanti, funzioni e/o variabili, ed eventualmente le macro definizioni che il modulo mette a disposizione dei moduli che lo includono, e la documentazione che specifica cosa (le implementazioni di) queste funzioni debbono fare.

L'unico modulo che può non avere interfaccia è il modulo che contiene la funzione che implementa il programma principale, la `main`. Un'altra eccezione è il modulo delle variabili globali, se ve ne sono (e le regole seguenti ne sconsigliano l'uso), che invece possiede solo l'interfaccia ma non l'implementazione (vedi ???).

2.1 Un esempio di interfaccia

Per chiarire gli aspetti stilistici essenziali che qui si propongono, possiamo cominciare con un semplice e classico esempio di modulo: uno stack. L'interfaccia espone un tipo di dati che può essere utilizzato in altri pezzi di codice: le operazioni sono quelle solite per una pila, cioè l'inserimento di un elemento in cima allo stack (push), l'eliminazione di un elemento dalla cima dello stack (pop) e l'ispezione della cima dello stack senza eliminarla (tos che sta per *top of stack*); inoltre prevede una funzione booleana per verificare se lo stack è vuoto, ed una funzione che torna il numero di elementi di uno stack:

Listing 1: L'interfaccia stack

```
/* stack.h — Questa interfaccia esporta un tipo stackT, che consente
   di memorizzare tipi di dato scalari: interi, double o *void. */

#if !defined(stack_VERSION)
#define stack_VERSION  1.0

/* TIPI */

typedef  struct stackT  *stackT;

/* COSTANTI */

/* stack inizialmente vuoto */
extern  const  stackT  stackEMPTY;

/* FUNZIONI */
```

```
/* Le funzioni stackPushXXX(s,x) premono su uno stack s un dato x e, se hanno
   successo, ritornano il nuovo stack, altrimenti stackEMPTY */
extern  stackT  stackPushInt( stackT s, int i );
extern  stackT  stackPushDouble( stackT s, double d );
extern  stackT  stackPushVoid( stackT s, void *v );

/* Le funzioni stackPopXXX(s,&x) tolgono dallo stack s un dato x e, se hanno
   successo, ritornano il nuovo stack e memorizzano il dato in x, altrimenti
   stackEMPTY e lasciano x inalterato */
extern  stackT  stackPopInt( stackT s, int *i );
extern  stackT  stackPopDouble( stackT s, double *d );
extern  stackT  stackPopVoid( stackT s, void **v );

/* Le funzioni stackTOS*(s) tornano il contenuto della testa dello stack, senza
   però eliminarla: se s==stackEMPTY, avviene un checked runtime error */
extern  int     stackTOSInt( stackT s );
extern  double  stackTOSDouble( stackT s );
extern  void    *stackTOSVoid( stackT s );

/* La funzione stackIsEmpty(s) torna 1 se lo stack s è vuoto, altrimenti torna 0 */
extern  int     stackIsEmpty( stackT s);

/* La funzione stackSize(s) torna il numero di elementi nello stack s */
extern  int     stackSize( stackT s);

/* MACRO */

/* Attenzione: non è garantito che funzionino per il tipo double!!! */
#define  stackPUSH(s,x)  (s = stackPushInt(s, (int)x))
#define  stackPOP(s,x)  (s = stackPopInt(s, (int*)x))
#define  stackTOS(s)  (stackTOSInt(s))

#endif
```

Come in ogni file, si inizia con un commento che riporta il nome del file e più o meno cosa contiene: i dettagli dei singoli oggetti commentati sono riportati prima della dichiarazione degli oggetti stessi. Si noti che un file di inclusione può essere incluso più volte, ecco perché usiamo una **#if** per verificare se una macro, esportata da questo file, è già stata definita: in questo caso, infatti, vuol dire che il file è stato già incluso, e quindi non va incluso nuovamente.

La prima cosa che viene fatta è poi definire questa macro che caratterizza il file: *come ogni nome esportato dal file inizia col nome del file stesso.* Che si tratta di macro, costanti, funzioni o altro, gli oggetti esposti dall'interfaccia hanno come nome un prefisso, che è il nome dell'interfaccia stessa. Questa caratteristica, apparentemente noiosa, ha almeno un duplice scopo:

(1) Definire uno spazio dei nomi (soluzione che ad esempio in C++ si potrebbe ottenere mettendo gli oggetti in un namespace) che non abbia conflitti con il resto del programma;

(2) Rendere trasparente il file dove un certo oggetto è definito: se inizia con stack allora sta per forza nel file `stack.h`.

Nel nostro caso definiamo una macro stackVERSION definita ad un numero double, che potrebbe ad esempio essere la versione, oppure la data, etc.

Per quanto non sia strettamente necessario, è più chiaro, ove possibile, raggruppare i dati dichiarati in base al loro tipo: ad esempio prima le costanti, i tipi, le variabili ed infine le funzioni. Nel caso dell'interfaccia 1 definiamo un tipo di dato stackT, che, come si vede, è opaco all'utente dell'interfaccia, nel senso che non si può accedere direttamente al suo contenuto: solo l'implementazione può farlo. Si noti che lo spazio dei nomi delle strutture e quello dei tipi è diverso in C, quindi possiamo dare lo stesso nome alla struttura ed al tipo che è definito per mezzo di essa.

Definiamo poi una costante per denotare lo stack vuoto: dalla definizione è chiaro che questa sarà NULL, ma è sempre meglio usare una costante propria del tipo.

Le funzioni e le variabili in una interfaccia debbono tutte essere dichiarate **extern**: notiamo che i commenti descrivono cosa ci si aspetta da queste funzioni, all'implementazione sta il come farlo.

Infine diamo alcune macro: molti estensori di regole stilistiche suggeriscono l'eliminazione o almeno l'uso solo in casi eccezionali delle macro, ed hanno in buona parte ragione, dato che una macro nasconde il codice che poi verrà effettivamente compilato. In questo caso, come spesso accade, usiamo alcune macro per stenografare le funzioni che abbiamo definito più sopra: proviamo a scrivere delle macro generiche, proprio perché l'interfaccia presume di definire stack per più tipi di dato: comunque questo tipo di soluzione, in C, non è molto pulita e tende ad essere soggetta ai capricci delle singole implementazioni: qualsiasi macro o gruppo di macro va commentata, ed in questo caso il commento è l'avvertimento appena detto.

È costume, da parte dei progettisti di interfacce, fornire anche delle macro che stenografino tutti gli oggetti definiti nell'interfaccia stessa, in modo che l'utente possa decidere se usare nomi più semplici di quelli imposti dalle regole: ad esempio si potrebbe aggiungere:

```
/* ALIAS */

/* Se l'utente ha definito una macro stackUSE allora può usare le seguenti
   forme abbreviate per gli oggetti definiti da questa interfaccia */
```

```
#if  defined(stackUSE)
#       define  stack  stackT
#       define  EMPTY  stackEMPTY
#       define  pushInt  stackPushInt
#       define  pushDouble  stackPushDouble
#       define  pushVoid  stackPushVoid
#       define  popInt  stackPopInt
#       define  popDouble  stackPopDouble
#       define  popVoid  stackPopVoid
#       define  TOSInt  stackTOSInt
#       define  TOSDouble  stackTOSDouble
#       define  TOSVoid  stackTOSVoid
#       define  isEmpty  stackIsEmpty
#       define  size  stackSize
#endif
```

2.2 Un esempio di implementazione

Per usare l'interfaccia è necessario averne almeno una implementazione: eccone una.

Listing 2: Una implementazione di stack

```
/* stack.c — Implementazione del modulo stack */

#include  <assert.h>    /* assert */
#include  <stdlib.h>    /* malloc, free, NULL */

#include  "stack.h"

/* IMPLEMENTAZIONE DEI TIPI */

struct stackT {
        union {
                int  i;
                double  d;
                void  *v;
        } data;
        stackT  next;
};

/* IMPLEMENTAZIONE DELLE COSTANTI */
const  stackT  stackEMPTY = NULL;

/* DICHIARAZIONE DI DATI PRIVATI */
```

```
enum { INT_TYPE, DOUBLE_TYPE, VOID_TYPE };

/* push preme sullo stack un dato: ritorna lo stack aggiornato se l'operazione
   va a buon fine, altrimenti ritorna NULL: il primo argomento denota il tipo di
   dato da premere sullo stack */
static stackT push( int t, int i, double d, void *v, stackT s );

/* pop estrae un dato dallo stack, e, a seconda del tipo, lo va a scrivere in
   uno dei tre parametri passati per riferimento; se lo stack è vuoto allora torna
   NULL, altrimenti torna lo stack aggiornato */
static stackT pop( int t, int *i, double *d, void **v, stackT s );

/* IMPLEMENTAZIONE DELLE FUNZIONI PUBBLICHE */

stackT stackPushInt( stackT s, int i )
{
        return push( INT_TYPE, i, 0.0, NULL, s);
}

stackT stackPushDouble( stackT s, double d )
{
        return push( DOUBLE_TYPE, 0, d, NULL, s);
}

stackT stackPushVoid( stackT s, void *v )
{
        return push( VOID_TYPE, 0, 0.0, v, s);
}

stackT stackPopInt( stackT s, int *i )
{
        double d;
        void *v;

        return pop( INT_TYPE, i, &d, &v, s );
}

stackT stackPopDouble( stackT s, double *d )
{
        int i;
        void *v;

        return pop( DOUBLE_TYPE, &i, d, &v, s );
}

stackT stackPopVoid( stackT s, void **v )
{
        int i;
        double d;
```

```
        return pop( VOID_TYPE, &i, &d, v, s );
}

int  stackTOSInt( stackT s )
{
        assert( s != NULL && "Stack vuoto!!!");
        return s->data.i;
}

double  stackTOSDouble( stackT s )
{
        assert( s != NULL && "Stack vuoto!!!");
        return s->data.d;
}

void  *stackTOSVoid( stackT s )
{
        assert( s != NULL && "Stack vuoto!!!");
        return s->data.v;
}

int  stackIsEmpty( stackT s)
{
        return s == NULL;
}

int  stackSize( stackT s )
{
        int  size = 0;

        while ( s != NULL ) {
                ++ size;
                s = s->next;
        }
        return size;
}

/* IMPLEMENTAZIONE DELLE FUNZIONI PRIVATE */

static  stackT  push( int t, int i, double d, void *v, stackT s )
{
        stackT  tos = malloc(sizeof(struct stackT));

        if ( tos != NULL ) {
                switch ( t ) {
                        case INT_TYPE:
                                tos->data.i = i;
                                break;
```

```
                        case DOUBLE_TYPE:
                                tos->data.d = d;
                                break;
                        case VOID_TYPE:
                                tos->data.v = v;
                                break;
                        default:
                                assert(!"Errore interno!!!");
                }
                tos->next = s;
                s = tos;
        }
        return tos;
}

static  stackT  pop( int t, int *i, double *d, void **v, stackT s )
{
        stackT  tos = s; /* top of stack */

        if ( tos != NULL ) {
                switch ( t ) {
                        case INT_TYPE:
                                *i = tos->data.i;
                                break;
                        case DOUBLE_TYPE:
                                *d = tos->data.d;
                                break;
                        case VOID_TYPE:
                                *v = tos->data.v;
                                break;
                        default:
                                assert(!"Errore interno!!!");
                }
                tos = s->next;
                free(s);
        }
        return tos;
}
```

Non è necessario in questo caso definire una macro per vedere se il file è stato già incluso: infatti un file di implementazione (come qualsiasi altro file `.c`) va compilato una sola volta, singolarmente o insieme ad altri, di questo se ne occupa il compilatore o l'ambiente di sviluppo.

In un file di implementazione debbono essere definiti gli oggetti dichiarati dall'implementazione, e possono essere definiti degli oggetti interni all'implementazione, che si vogliono occultare al resto del programma, in particolare

al cliente dell'interfaccia: di solito i dati privati all'implementazione vengono dichiarati all'inizio del file, e poi definiti in coda.

In questo caso l'implementazione è così semplice che non richiederebbe una struttura interna elaborata, comunque: per prima cosa implementiamo i tipi e le costanti esportate. Uno `stackT` punta ad una struttura `struct stackT`, che possiede una parte contenente il dato in cima allo stack, ed un puntatore al resto dello stack; la costante `stackEMPTY` è semplicemente il puntatore vuoto `NULL`.

Poi abbiamo una parte dichiarativa, che introduce i dati privati dell'implementazione, invisibili all'esterno: questi sono delle costanti (definite per mezzo di un tipo enumerato e non con delle macro: una alternativa era usare il costrutto `const int INT_TYPE=0; ...`) e due funzioni, documentate con un commento.

A questo punto nel file sono definite la funzioni esposte dall'interfaccia, poggiando sulle funzioni private all'implementazione, che sono definite nell'ultima sezione del file `stack.c`. Si noti l'uso delle asserzioni, per catturare gli errori a run-time: un'interfaccia dovrebbe limitarsi a segnalare gli errori che intercetta e non a risolverli (ad esempio tornando `NULL` laddove deve tornare un puntatore), ma anche gestire gli errori di run-time che sono dichiarati *checked run-time error* dall'interfaccia stessa, cioè che bloccano l'esecuzione ma in modo consapevole: in fase di produzione si può optare per eliminare questi controlli, e per farlo basta definire la macro `NDEBUG`: in questo modo la macro `assert(x)` si espande a `((void)0)`.

Questa implementazione dovrebbe essere facilmente intellegibile, ed illustrare le principali regole stilistiche che qui si propongono in fatto di spaziatura, indentazione, accoppiamento delle parentesi graffe, etc. Il principio generale di questo stile è che lo spazio verticale è più prezioso di quello orizzontale: ad esempio non ci sono graffe su singole righe, tranne che per delimitare il corpo di una funzione, che è un blocco di tipo particolare. Le keyword e gli operatori binari sono sempre delimitati da almeno uno spazio, e `return` non ha l'argomento fra parentesi (secondo la scuola di pensiero che la vuole come un enunciato e non come una funzione).

Seguendo la consuetudine del `C`, i dati vengono definiti all'inizio di un blocco, preferibilmente all'inizio di una funzione, e sono separati dal codice eseguibile con una riga vuota (anche se appaiono in un blocco interno alla funzione). Ovviamente per `C++`, `Java`, etc. si può definire una variabile non appena se ne ha bisogno, ma in generale sembra raccomandabile avere una lista delle variabili usate, ed imporsi di dichiararle all'inizio incita a scrivere funzioni più corte.

2.3 Un esempio di programma

Per concludere mostriamo un esempio di programma che utilizza (per un toy-test) l'interfaccia appena definita ed implementata.

Listing 3: prova-stack.c

```
/* prova-stack.c - programma di prova dello stack */

#include <stdio.h>

#include "stack.h"

int main( int arg_num, char **args )
{
        const int MAX = 10;
        int i;
        stackT istack = stackEMPTY;     /* uno stack di interi */
        stackT dstack = stackEMPTY;     /* uno stack di double */
        stackT cstack = stackEMPTY;     /* uno stack di caratteri */

        for ( i = 0; i < MAX; ++ i ) {
                stackPUSH(istack, i);
        }
        printf("Ora dovrebbe stampare i numeri da %i a 0:\n", MAX-1);
        while ( !stackIsEmpty(istack) ) {
                stackPOP(istack, &i);
                printf("%i ", i);
        }
        putchar('\n');

        stackPUSH(cstack, "gamma");
        stackPUSH(cstack, "delta");
        stackPUSH(cstack, "beta");
        stackPUSH(cstack, "alpha");
        puts("Ora dovrebbe stampare le prime 4 lettere dell'alfabeto greco:");
        while ( stackSize(cstack) > 0 ) {
                void *c;
                stackPOP(cstack, &c);
                puts(c);
        }

        dstack = stackPushDouble(dstack, 1.0);
        puts("Successione ricorsiva f(n)=f(n-1)+n:");
        for ( i = 1; i < MAX; ++ i ) {
                dstack = stackPushDouble(dstack, i + stackTOSDouble(dstack));
        }
        while ( !stackIsEmpty(dstack) ) {
                double d;
```

```
                dstack = stackPopDouble(dstack, &d);
                printf("%f ", d);
        }
        putchar('\n');

        puts("Ora interrompe l'esecuzione perché lo stack è vuoto...");
        i = stackTOS(istack);
        return 0;
}
```

Nel caso di linguaggi orientati agli oggetti, le singole classi vanno un po' considerate come dei moduli, ed è bene porre la dichiarazione di una classe in un singolo file, magari chiamato come la classe stessa, e l'implementazione in un altro file, con diversa estensione: per quel che riguarda l'ordine degli elementi nella classe, è spesso suggerito il seguente:

(1) costruttori;

(2) metodi istanza;

(3) metodi statici;

(4) variabili istanza;

(5) variabili statiche;

(6) classi interne.

Ciascun metodo (escluso `main`) va commentato: nel caso di `Java` i commenti dovrebbero essere possibilmente nel formato `javadoc`.

3 Regole stilistiche e convenzioni lessicali

Le convenzioni lessicali di carattere generale dovrebbero essere chiare dagli esempi precedenti: una graffa aperta è l'ultimo carattere di una riga (a meno di un commento) ed una graffa chiusa il primo, le graffe corrispondenti si incolonnano solo nel caso in cui delimitino il corpo di una funzione. Le righe non dovrebbero superare le 80 colonne, un limite di leggibilità, e possono essere proseguite nelle righe seguenti con un doppio rientro, possibilmente iniziando con un operatore, come in

```
var = ..............................................................
    \
                + ..............................;
```

```
if ( ( var == 0  && ..................................................
    \
                ||  a-1 == b ) {
        var = a + b;
        ...
}
```

In generale è sempre meglio avvolgere un enunciato in graffe, anche se questo non è strettamente necessario: ad esempio la sequenza

```
if ( x == 0 )
        if ( y < 0 )
                x = -y;
        else
                x = y;
```

non è molto chiara, meglio scriverla come

```
if ( x == 0 ) {
        if ( y < 0 ) {
                x = -y;
        } else {
                x = y;
        }
}
```

che in fondo richiede solo una riga di spazio in più. Inoltre questa convenzione elimina il tipico bug di inconsistenza fra indentazione e semantica, come il seguente

```
if ( x == 0 )
        if ( y < 0 )
                x = -y;
else
        x = y;
```

(*dangling else problem*). Chi legge questo codice potrebbe pensare che le due if sono ridondanti e riscriverlo come

```
if ( x == 0  &&  y < 0 )
        x = -y;
else
        x = y;
```

introducendo un bug.

3.1 Scelta dei nomi

I nomi possono denotare essenzialmente due tipi di oggetti: oggetti locali ed oggetti globali. Per gli oggetti locali è consigliabile usare nomi brevi (la

lunghezza del nome dovrebbe essere inversamente proporzionale alla frequenza d'uso), anche monoletterali nel caso di contatori, etc. Ad esempio `n_points` è preferibile a `number_of_points`: la tradizione del `C` aborre la mescolanza di maiuscole e minuscole, `comeInQuestoNome`, favorendo l'uso della sottolineatura `come_in_quest_altro_caso`: tuttavia entrambe le convenzioni possono essere utilmente impiegate.

Specificatamente, osserviamo le seguenti regole:

(1) i nomi di variabili locali o parametri di funzione (che sono delle particolari variabili locali) sono tutti in minuscole `con_eventuali_sottolineature`.

(2) i nomi di costanti o macro locali sono scritti in `MAIUSCOLO_CON_SOTTOLINEATURA`.

(3) i nomi di funzioni o di variabili esterne sono scritti con un prefisso di minuscole, che coincide col nome del file di inclusione nel quale sono dichiarati (senza estensione), e proseguono con un nome scritto in maiuscole e minuscole: ad esempio `stackPop`.

(4) i nomi di funzioni e variabili private in un modulo (ad esempio le funzioni statiche del `C`) possono essere trattati come i nomi delle variabili locali, o come nomi di funzioni esterne, se si vuole evitare di confonderle con nomi della libreria: ad esempio `push` potrebbe essere anche chiamata `stackPush`.

Di solito è inutile inserire nel nome una caratteristica che riveli la natura dell'oggetto: ad esempio se un nome denota un tipo di dati o una funzione lo si vede dal contesto. Anche le seguenti regole di carattere generale (cfr. [6]) sono utili da tenere a mente:

- Essere consistenti, cioè dare ad oggetti fra loro legati dei nomi che ne evidenzino il legame e ne sottolineino le differenze, e che non convoglino informazioni inutili: un campo di una struttura `stack` non deve, ad esempio, chiamarsi `stack_size`.
- Usare nomi attivi per le funzioni, cioè che esprimano l'azione che la funzione compie, usando la sintassi verbo-oggetto, ad esempio `getDate()`; inoltre per le booleane, che denotino in modo non ambiguo il risultato, come `is_char` da preferire a `check_char`.
- Scegliere nomi che possono sostituire un commento, a meno che non sia chiaro il ruolo dell'oggetto da nominare: ad esempio `index_of_loop` va scartata in luogo di `i`, mentre `paymentNumber` può risparmiare un commento a `pn`.

- Scegliere in generale nomi che siano facili da leggere ad alta voce: ad esempio non eliminare le `vcl`...

Naturalmente la regola che governa ogni insieme di regole è che, alla bisogna, le si può violare: ad esempio alcuni programmi, come `Murex`, richiedono di adeguarsi alle loro regole stilistiche se si vogliono programmare le loro estensioni: in particolare `Murex` usa una notazione che ricorda quella ungherese della Microsoft per i dati, imponendo un prefisso al nome della variabile a denotarne il nome, come `pcLabel` che denota un `p`untatore a `c`har (questo non è considerato un bello stile, cfr. e.g. [12], ma in questo caso "il client non ha sempre ragione" e si deve adattare alle convenzioni del programma).

Un ultimo avviso: lo standard `C` riserva (anche per future estensioni) certi nomi: tanto per dirne una `strano` e `memoria` sono da considerarsi riservati: inoltre se si scrive `C` e si vuole essere compatibili col `C++`, bisogna evitare nomi che siano parole chiave di quest'ultimo. La stessa avvertenza vale per altri linguaggi, cfr. [1], [5].

3.2 Espressioni

Le espressioni nei linguaggi derivati dal `C`, o comunque la cui sintassi si ispira al `C`, sono molto ricche ed offrono numerosi modi equivalenti di effettuare uno stesso calcolo: ad esempio in `C` qualsiasi istruzione è in realtà la valutazione di un'espressione. Questo implica che si possono scrivere espressioni molto compatte, che altrimenti richiederebbero diverse istruzioni, come

```
x = ( x == 0 ? (( y > 0 ) y : -y ) : ( ( y == 0 ) ? -x : y ) );
```

che è una forma molto compatta del seguente brano di codice:

```
if ( x == 0 ) {
        if ( y > 0 )
                x = y;
        else
                x = -y;
} else {
        if ( y == 0 )
                x = -x;
        else
                x = y;
        }
}
```

Il compromesso è spesso la soluzione migliore, ad esempio imporsi di limitare l'uso dell'operatore condizionale `:?` ad un solo livello di nidificazione:

```
if ( x == 0 ) {
```

```
        x = ( y > 0 ) ? y : -y;
} else {
        x = ( y == 0 ) ? -x : y;
}
```

L'abbondanza di parentesi anche quando non sono strettamente necessarie è invece sempre da considerarsi come salutare, un po' perché evita a chi legge il dilemma di ricordare la precedenza degli operatori, un po' per chiarezza: ad esempio l'espressione

```
if ( x&0355 == bit_sequence ) ...
```

viene valutata dal compilatore come

```
if ( x & ( 0355 == bit_sequence )  ) ...
```

mentre probabilmente il programmatore intendeva

```
if ( ( x & 0355) == bit_sequence ) ...
```

Le espressioni complesse risultano sempre difficili da decifrare, sia per gli umani che per i compilatori, che in genere ottimizzano meglio espressioni sintetiche: alcuni suggeriscono addirittura di scrivere una sequenza

```
x *= y / ( x - a );      /* x = xy/(x-a) */
```

come

```
temp = x - a;
x /= temp;
x *= y;
```

adducendo a giustificazione che queste tre istruzioni corrispondono quasi alla lettera a tre istruzioni macchina: questa posizione è estrema, ma, al solito, accettarne lo spirito più che la lettera porta giovamento.

Inoltre è buona norma evitare gli effetti laterali, cioè gli operatori di assegnazione, incremento e decremento di variabili all'interno di espressioni: questa è quasi una bestemmia per i programmatori C, che si compiacciono di espressioni idiomatiche come la seguente:

```
while ( *p++ = *q++ ) ;
```

che copia una stringa `q` su una stringa `p`: questa espressione contiene comunque un errore stilistico notevole, e cioè il `;` a seguire l'espressione. Ciò è male per almeno due motivi: primo, dimenticare questo `;` ha effetti disastrosi, come in

```
while ( *p++ = *q++ )
free(q);
```

Secondo, l'istruzione vuota, cioè `;`, ha la dignità di istruzione, e quindi va evidenziata su una sola riga come in

```
while ( *p++ = *q++ )
        ; /* empty! */
```

Questo ciclo si può scrivere in modo prolisso ma decisamente più intellegibile come

```
do {
        *p = *q;
        ++ p;
        ++ q;
} while ( *p != '\0' );
```

Verosimilmente questa codifica genererà un codice non meno efficiente dell'altra: in generale una assegnazione o un incremento andrebbero sempre scritti in una singola istruzione.

Un altro buon motivo per evitare effetti laterali è che una delle funzioni coinvolte nell'espressione potrebbe essere in realtà una macro, o contenere delle variabili statiche. Ad esempio

```
#define  MAX(a,b) ( ( a > b ) ? a : b )
...
x = MAX(++i, ++j);
```

Qui, l'istruzione che il compilatore compila è la seguente:

```
x = ( ( ++i > ++j ) ? ++i : ++j );
```

col risultato che una delle due variabili sarà incrementata due volte. Altro esempio: data la funzione

```
int  f( int i )
{
        static  int  inner = 0;
        ++ inner;       /* numero di volte che la funzione è chiamata */
        return i + inner;
}
```

l'espressione `f(x) == f(x)` *sarà sempre falsa!!!*

Ultimo esempio sugli accidenti degli effetti laterali, fra i quali vanno rubricate le operazioni di input/output:

```
scanf("%d, %d", &yr, &profit[yr]);
```

Dato che `scanf` valuta i suoi argomenti prima delle operazioni di input/slash output, l'elemento del vettore `profit` che sarà modificato sarà quello puntato da `yr` prima di leggerne il valore, che verosimilmente non è quanto intendeva il programmatore: di nuovo la soluzione sta nello scindere l'espressione in

```
scanf("%d,", &yr);
scanf("%d", &profit[yr]);
```

Nelle espressioni logiche, è sempre bene usare le parentesi fra i termini collegati dai connettivi logici, a meno che non si tratti di un connettivo solo o sempre dello stesso: ad esempio

```
if ( !a && b || !c ) ...
```

andrebbe scritta come

```
if ( ( !a && b ) || !c ) ...
```

A proposito delle negazioni, poi, andrebbero evitate il più possibile: ad esempio alla condizione

```
if ( !( a || !( b && !c ) ) ) ...
```

è bene sostituire la condizione

```
if ( !a && ( !b || c) ) ...
```

ad essa logicamente equivalente ma con meno negazioni.

Infine bisogna sempre verificare se un dato è uguale o meno al dato nullo invece che usare la notazione ellittica propria del C:

```
if ( p = malloc(SIZE) ) {
        ...
} else {
        error();
}
```

va riscritto come

```
p = malloc(SIZE)
if ( p == NULL ) {
        error();
} else {
        ...
}
```

Questo è, nel caso dei numeri in virgola mobile, anche poco sicuro: non scriviamo

```
double val;
if ( !val ) ...
```

bensì

```
double val;
if ( val == 0.0 ) ...
```

anche se dovremmo scrivere

```
double val;
if ( fabs(val) < EPSILON ) ...
```

o qualcosa del genere. Lo stesso discorso vale per gli interi, i puntatori, etc: in generale l'operatore `!` dovrebbe essere applicato solo alle variabili booleane.

3.3 Macro, costanti e funzioni

Le macro debbono essere usate con parsimonia: la tradizione che ne vuole i nomi in maiuscole è sorta proprio per evidenziarne la pericolosità d'uso... Per definire una costante è sempre meglio usare una dichiarazione come

```
const char USAGE[] = "Usage: foo filename\n";
```

o sfruttare il tipo enumerato, come in

```
enum { ANS_OK = 1000, ANS_NO = 2000, ANS_UNKNOWN = 3000 };
```

da preferire a

```
#define ANS_OK 1000
#define ANS_NO 2000
#define ANS_UNKNOWN 3000
```

Il motivo è ovvio: una macro è solo un oggetto sintattico, mentre una costante è sottoposta a tutti i tipi di checking delle variabili (sul tipo, sui valori, etc.).

A proposito di costanti, la principare regola è di evitare costanti magiche, cioè numeri che apparentemente non hanno un significato particolare ma che stanno in mezzo al codice: ad esempio in

```
x = 360 * ry - 12 * rm;
```

può essere evidente che 360 sono i giorni dell'anno lavorativo e 12 i mesi del suddetto, ma potrebbe anche non esserlo: non costa nulla definire due costanti YEAR_DAYS e MONTH_DAYS definite come 360 e 12, anche perché potrebbe accadere di voler cambiare il numero convenzionale di giorni dell'anno lavorativo, e quindi di dover cercare tutte le occorrenze di 360 nel codice, mentre in questo modo basterà cambiare il valore di una costante. Ovviamente alcuni numeri molto comuni, come -1, 0, 1 e 2, sono da considerarsi costanti non magiche, e quindi possono apparire liberamente nel codice.

Vanno considerate costanti magiche anche le "taglie" dei tipi di dato: ad esempio si scriva

```
heap = calloc(HEAP_SIZE, sizeof(int));
```

invece di

```
heap = calloc(HEAP_SIZE, 4);
```

Altro esempio: per calcolare la lunghezza di un vettore si potrebbe usare

```
#define ASIZE(v) ( sizeof(v) / sizeof(v[0]) )
```

e quindi usare questa espressione ad esempio in un ciclo:

```
for ( i = 0; i < ASIZE(v); ++ i ) {
        ...
}
```

In generale è sempre bene lasciare al compilatore il compito di calcolare non solo le dimensioni degli oggetti, ma anche espressioni costanti, che possono essere lasciate per esteso in favore della chiarezza: ad esempio alla sequenza

```
const double FT2METER = 0.3048;
const double METER2FT = 3.28084;
```

va preferita

```
const double FT2METER = 0.3048;
const double METER2FT = 1.0 / FT2METER;
```

Le macro-funzioni pure andrebbero evitate: il vantaggio in efficienza rispetto alle funzioni propriamente dette è quasi sempre trascurabile, mentre i possibili bug sono difficili da stanare: un classico è

```
#define SQUARE(x) x * x
```

Con questa definizione, il brano di codice

```
x += SQUARE(y-a);
```

diviene

```
x += y-a * y-a;
```

Anche la correzione

```
#define SQUARE(x) (x) * (x)
```

dà luogo ad un errore nell'espansione di

```
z = 1 / SQUARE(y);
```

che diviene

```
z = 1 / (y) * (y);
```

La versione corretta è:

```
#define SQUARE(x) ( (x) * (x) )
```

e la regola recita: racchiudere fra parentesi qualsiasi cosa appaia in una macrodefinizione a partire dalla definizione stessa. Spesso le macro sono usate congiuntamente agli effetti laterali, il che le rende di difficile interpretazione, come

```
#define FOO(x) (!(x==0) || x=1, x=2)
```

3.4 Enunciati

Sulla forma delle istruzioni imperative abbiamo insistito abbastanza, specie per quel che riguarda l'uso delle graffe. Un ottimo consiglio che spesso viene dato è di non abusare del ciclo for, ma di utilizzarlo solo se viene naturale farlo: ad esempio

```
for ( i=0, *p=*q; *q != '\0' && i < MAX; ++ i, *p++=*q++  )
        ;
```

non è naturale come

```
*p = *q
for ( i = 0; *q != '\0'  &&  i < MAX; ++ i )
        ++ p;
        ++ q;
        *p = *q;
}
```

A proposito di questo esempio dobbiamo menzionare la *vexata quæstio* dell'uso dei salti incondizionati **break**, **continue** e **return** (il **goto** non lo usiamo per principio). Una scuola di pensiero li aborre (cfr. [4]) e propone di sostituirli sempre con test su variabili booleane ausiliarie, o semplicemente imponendo condizioni di fine ciclo: è quanto abbiamo fatto nel caso precedente anziché scrivere

```
*p = *q
for ( i = 0; i < MAX; ++ i )
        if ( *q == '\0' )  break;
        ++ p;
        ++ q;
        *p = *q;
}
```

che in effetti è un pessimo stile: tuttavia possono sussistere situazioni in cui un codice senza salti incondizionati è meno leggibile di uno che li sostituisce con condizioni booleane. In generale proporrei la liceità dei salti incondizionati se servono a saltare al livello di *scope* più alto della funzione, e se la funzione in questione è breve (diciamo non più di dieci righe di codice, commenti esclusi): ad esempio la seguente funzione di ricerca lineare

```
int  find( double a[], double x, int size_a )
{
        register  int  i;

        for ( i = 0; i < size_a; ++ i )
                if ( a[i] == x )  return i;
        return -1;
}
```

è facilmente intellegibile: volendo abolire i **return** per amore di simmetria (una funzione ha solo un *entry-point*, quindi perché dovrebbe avere più *leaving-point*?) possiamo usare

```
int  find( double a[], double x, int size_a )
{
        register  int  i;

        for ( i = 0; i < size_a; ++ i )
                if ( a[i] == x )  break;
        return ( i == size_a) ? -1 : i;
}
```

che però introduce un condizionale alla fine della funzione e ne peggiora la leggibilità e la semplicità. Più leggibile ma non più della versione con il **return**, è

```
int  find( double a[], double x, int size_a )
{
        register  int  i;
        register  int  found = 0;

        for ( i = 0; !found  &&  i < size_a; ++ i ) {
                if ( a[i] == x )  found = 1;
        }
        return ( found ) ? i : -1;
}
```

che fra l'altro si potrebbe riscrivere risparmando una negazione, come

```
int  find( double a[], double x, int size_a )
{
        register  int  i;
        register  int  not_found = 1;

        for ( i = 0; not_found  &&  i < size_a; ++ i ) {
                if ( a[i] == x )  not_found = 0;
        }
        return ( not_found ) ? -1 : i;
}
```

In questo caso la versione più semplice (e quindi quella da preferire) è quella che usa il **return**: la funzione è così breve che non si può parlare di codice spaghetti in questo caso...

In generale l'avviso che molti compilatori di regole stilistiche sogliono dare è di scrivere un codice il più possibile semplice da leggere e privo di ambiguità, anche a scapito della compattezza e delle espressioni idiomatiche del linguaggio. Un codice semplice da leggere per un umano sarà anche facile da compilare per la macchina, in linea di massima.

Molti sconsigliano l'uso dell'enunciato **switch** per il noto "errore di progetto" dovuto al fatto di dover chiudere un blocco **case** con un **break**, che potrebbe essere facile omettere per sbaglio: comunque in certe situazioni lo **switch** è la scelta più naturale. Kernighan e Pike in [7] propongono il seguente esempio di *pessimo* stile:

```
switch ( c ) {
case '-': sign = -1;
case '+': c = getchar();
case '.': break;
default:  if ( !isdigit(c) )  return 0;
}
```

che mostra un uso cosciente l'effetto *fall-throughs* del blocco case, mentre la seguente forma equivalente è consigliata in questo caso:

```
switch ( c ) {
case '-':
        sign = -1;
        /* no break: fall through */
case '+':
        c = getchar();
        break;
case '.':
        break;
default:
        if ( !isdigit(c) )  return 0;
        break;
}
```

Altro consiglio spesso impartito: utilizzare per scelte multiple una struttura di tipo IF-ELIF...-ELSE, che certi linguaggi hanno. Ad esempio, sempre citando da [7], la sequenza

```
if ( argc == 3 )
        if ( (fin = fopen(argv[1], "r")) != NULL )
                if ( (fout = fopen(argv[2], "w")) != NULL ) {
                        while ( (c = fgetc(fin)) != EOF )
                                fputc(c, fout);
                        fclose(fout);
                        fclose(fin);
                } else
                        printf("Can't open output file %s", argv[2]);
        else
                printf("Can't open input file %s", argv[1]);
else
        puts("Usage:  cp infile outfile");
```

si presta ad essere scritta come

```
IF ( argc != 3 )
```

```
        puts("Usage:  cp infile outfile");
ELIF ( (fin = fopen(argv[1], "r")) == NULL )
        printf("Can't open input file %s", argv[1]);
ELIF ( (fout = fopen(argv[2], "w")) != NULL )
        printf("Can't open output file %s", argv[2]);
ELSE
        while ( (c = fgetc(fin)) != EOF )
                fputc(c, fout);
        fclose(fout);
        fclose(fin);
ENDIF
```

In C si può scrivere ad esempio

```
if ( argc != 3 )
        puts("Usage:  cp infile outfile");
else if ( (fin = fopen(argv[1], "r")) == NULL )
        printf("Can't open input file %s", argv[1]);
else if ( (fout = fopen(argv[2], "w")) != NULL )
        printf("Can't open output file %s", argv[2]);
else {
        while ( (c = fgetc(fin)) != EOF )
                fputc(c, fout);
        fclose(fout);
        fclose(fin);
}
```

Alternativamente si potrebbero definire le seguenti macro:

```
#define  IF(cond)   if (cond) {
#define  ELIF(cond)   } else if (cond) {
#define  ELSE   } else {
#define  ENDIF }
```

3.5 Commenti

Commentare il codice è notoriamente più difficile che scriverlo: i programmatori molto esperti consigliano di ridurre al minimo i commenti, e di rendere il codice facile da capire, ad esempio scegliendo nomi appropriati per le variabili e le funzioni.

In generale qualsiasi oggetto globale e qualsiasi funzione o metodo andrebbero commentati, come pure gli oggetti statici nelle funzioni e nelle classi, se non altro per spiegare perché sono statici. Se una funzione usa un algoritmo non ovvio si può dare uina descrizione o rinviare a qualche fonte che spieghi l'algoritmo, etc.

Un distillato di regole può essere:

(1) Non commentare l'ovvio: ad esempio
```
++ counter; /*incrementa il contatore */
```

(2) Commentare sempre macro, funzioni ed oggetti globali.

(3) Non cercare di spiegare un codice scritto male ma riscrivere il codice: si dice spesso che se un codice richiede molti commenti è meglio riscriverlo daccapo.

(4) Non contraddire il codice: ogni volta che si cambia il codice va cambiato (o alla peggio eliminato) anche il commento, e viceversa.

(5) Usare poche parole ma scrivere frasi in italiano (o qualsiasi altra lingua) corretto, con punteggiatura, evitando ellitticità: ad esempio
```
p = f(p)— 10; /*setta param. a val = count — 10 */
```
è sicuramente poco intellegibile.

4 Altro

Un documento serio sullo stile e la standardizzazione delle regole di programmazione dovrebbe anche occuparsi di stabilire regole condivise per:

- La progettazione dei moduli o delle classi.
- Il debugging del codice.
- Le politiche di testing e profiling.
- Gli strumenti di tuning ed ottimizzazione del codice.

Per tutti questi argomenti il libro di Kernighan e Pike [7] fornisce un distillato di massime; per quel che riguarda la progettazione di moduli e classi si veda anche il libro di Hanson [2]; per tecniche elementari di ottimizzazione e tuning del codice si possono vedere Garg [3], Lee [10] ed anche il libro di La Mothe [8].

Un esempio reale di linee guida per un (enorme) ambiente di sviluppo è quello della NASA [11]; un altro esempio più semplice ma ben ragionato è quello di Jones [5]; infine, alcune massime divertenti ma non inappropriate sono contenute nel decalogo di Spencer [13].

Bibliografia

[1] Brown S., *C Reserved Identifiers*:
http://www.oakroadsystems.com/tech/c-predef.htm

[2] Hanson D., *C: Interfaces and Implementations*, Addison Wesley, 1997.

[3] Garg S., *How To Optimize C/C++ Source- Performance Programming*,
http://bdn.borland.com/article/0,1410,28278,00.html

[4] Horstmann C., *Concetti di informatica e fondamenti di Java2*, Apogeo, 2002.

[5] Jones D.W., *A Manual of C Style*:
http://www.cs.uiowa.edu/~jones/syssoft/style.html

[6] Kernighan B., Pike R., *The Practice of Programming*, Addison Wesley, 1999.

[7] Kernighan B., Ritchie D., *The C Programming Language*, Addison Wesley, 1988[2].

[8] La Mothe A., *Tricks of the Windows Game Programming Gurus*, Sams, 2002.

[9] Larson J., *Standards and Style for Coding in ANSI C*:
http://www.jetcafe.org/ jim/c-style.html

[10] Lee M.E., *Optimization of Computer Programs in C*:
http://vision.eng.shu.ac.uk/bala/c/c/optimisation/1/optimization.html

[11] NASA Software Engineering Laboratory, *C Style Guide*, 1994.

[12] Newcomer J.M., Rector B.E., *Win32 Programming*, Addison Wesley, 1997.

[13] Spencer H, *The Ten Commandments for C Programmers (Annotated Edition)*:

http://www.lysator.liu.se/c/ten-commandments.html