Stile di programmazione

Laboratorio di Informatica

```
v,i,j,k,l,s,a[99];
main()
{
for(scanf("%d",&s);*a-s;v=a[j*=v]-
a[i],k=i<s,j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-
(!l<<!j),"#Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-
i+j&&v+i-
j))&&!(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&+
+a[--i])
                ;
}
```

Baruch Nissenbaum, Best Small Program, The International Obfuscated C Code Contest, 1990

```
if ( (country == SING) || (country == BRNI) ||
(country == POL) || (country == ITALY) )
{
/*
* Se il paese è Singapore, Brunei o Polonia
* allora il tempo corrente è il tempo di risposta
* piuttosto che il tempo di conversazione.
* /
```

➔ Cosa è successo all'Italia?

➔ Cosa hanno in comune questi quattro paesi?

Esempio di codice ben scritto, ma non comprensibile

Perché uno stile di programmazione?

Un programma è un elemento di conoscenza

- → per la macchina
  - *è comprensibile se può essere compilato*
- → per il programmatore
  - *è comprensibile se è chiaro e semplice, cioé:*
    - Ha una logica immediata
    - Usa espressioni vicine al linguaggio naturale
    - Usa forme convenzionali
    - Usa nomi significativi
    - Usa una formattazione pulita, che ne riflette la struttura
    - Ha commenti significativi, che aiutano la comprensione

Nomi

- Cosa c'è in un nome?
  - *Il nome di una funzione o di una variabile porta informazione sul suo scopo*
- Caratteristiche di un buon nome
  - *Informativo (chiarisce lo scopo)*
  - *Conciso*
  - *Mnemonico*
  - *Pronunciabile*
- Più ampio è l'ambito (scope) di un nome, maggiore è l'informazione portata dal nome

Descrittività dei nomi

- Variabili e funzioni ad ampio ambito
  - *utilizzate in sezioni disparate del programma*
  - *richiedono nomi lunghi e descrittivi*
  - *utile accompagnarle da un commento*
  - *es.* `int n_pending = 0; // numero di elementi in coda`
- Variabili e funzioni ad ambito limitato
  - *L'ambito può fornire già informazioni sul loro ruolo*
  - *meglio usare nomi brevi*
    - i,j,k per indici
    - s,t per stringhe; c per caratteri
    - n,m per interi
    - p,q per puntatori
    - x,y,u,v,z per numeri frazionari

Esempi

```
for (theElementIndex = 0 ;
    theElementIndex < number0fElements;
    theElementIndex++)
        elementArray[theElementIndex] =
                    theElementIndex;
```

```
for (i = 0 ; i < nelems; i++)
    elem[i] = i ;
```

Consistenza dei nomi

- Variabili o funzioni correlate devono avere nomi che evidenziano la correlazione
- Variabili o funzioni che possono essere confusi devono avere nomi che evidenziano le differenze

Esempi

```
int num_items_in_Q;
int frontQueue;
int queue_capacity;
int noOfUsersInQueue(){...}
```

```
static int queue_items_num;
int queue_front;
int queue_capacity;
int get_queue_items_num(){...}
```

Nomi attivi per le funzioni e le procedure

- Funzioni e procedure compiono azioni.
  → usare forme attive per denominarle
  - *es. now = get_time();*
  - *es. putchar('\n')M*
- Le funzioni che ritornano un valore Booleano esprimono proposizioni.
  → usare la forma "is" + aggettivo
  - *es. isoctal(c)*

Correttezza

- Un nome porta un significato
  - *ciò che il nome richiama nella mente di chi lo legge*
- Una funzione o procedura ha un significato
  - *il corpo della funzione*
- Il significato del nome e il significato della funzione devono coincidere

Esempi

```
int isoctal(char c)
{
   return (c >= '0') && (c <= '9');
}
```

```
int isoctal(char c)
{
   return (c >= '0') && (c <= '7');
}
```

Esercizio

Commentare il seguente brano di codice

```
#define TRUE 0
#define FALSE 1
if ((ch = getchar()) == EOF)
   not-eof = FALSE;
```

Espressioni e istruzioni

- Le espressioni devono essere scritte in modo che il loro significato sia il più possibile trasparente
- Più espressioni sono equivalenti: scegliere quella più chiara
  - *Non sempre la più chiara è la più breve*
  - *Usare spazi tra operatori per suggerire raggruppamenti*
  - *In generale, formattare l'espressione per aumentarne la leggibilità*

Indentazione

- L'indentazione mostra la struttura di un programma
  - *In alcuni linguaggi, l'indentazione definisce la struttura di un programma*
    - In C la struttura è definita con le parentesi graffe {...}
- L'indentazione offre una visione gerarchica di un programma
  - *La gerarchia aiuta a organizzare la complessità*
  - *La gerarchia è insita nel programma*

Esempi

```
for(n++;n<100;field[n++]='\0');
*i='\0'; return('\n');
```

```
for(n=n+1; n<100; n++){
   field[n] = '\0';
}
*i = '\0';
return '\n';
```

Forme “naturali”

- Scrivere le espressioni come se fossero pronunciate ad alta voce
  - *Limitare l’uso delle forme negate*
  - *Evitare espressioni che siano troppo lunghe*
    - nel caso, spezzare le espressioni in sotto-espressioni oppure fattorizzare in funzioni

- `if(!(a==0) || !(b==0))...`
- `if(!((a==0) && (b==0)))...`
- `if((a!=0) || (b!=0))...`

Uso delle parentesi

- Le parentesi raggruppano
  - *possono chiarire il significato di un'espressione anche quando non sono necessarie*
  - *Come l'indentazione, il raggruppamento organizza la complessità*
- Quando si usano operatori di diversa natura, le regole di precedenza non sono immediate.
  - *Es: a!=0&&b+1==0 → (a!=0) && (b==-1)*

Esempi

```
leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

```
leap_year = ((y % 4 == 0) && (y % 100 != 0))
          || (y % 400 == 0);
```

Espressioni complesse

```
x += (xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

➔ Le espressioni complesse devono essere spezzate in espressioni più semplici da comprendere

```
if (2*k < n-m)
   xp = c[k+1];
else
   xp = d[k--];
x += *xp;
```

Chiarezza delle espressioni

```
child = (!LC&&!RC)?0:(!LC?RC:LC)
```

➔ Il codice deve esprimere chiaramente l'intenzione del programmatore, non la sua bravura

```
if ((LC==0) && (RC==0))
   child = 0;
else if (LC==0)
   child = RC;
else
   child = LC;
```

Attenzione agli effetti collaterali

- Un effetto collaterale è una modifica dello stato di un programma in esecuzione, dovuto indirettamente dall'esecuzione di un'istruzione
  - *es. ordine di esecuzione non definito*
  - *short-circuit evaluation*

Ordine di esecuzione indeterminato

```
str[i++] = str[i++] = ' ';
```

L’ordine di valutazione è deciso dal compilatore, e può determinare comportamenti inattesi.

```
str[i++] = ' ';
str[i++] = ' ';
```

Ordine di esecuzione determinato

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

L’ordine è determinato, ma l’istruzione è controintuitiva.

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

Esercizio

Migliorare la leggibilità dei seguenti snippet

```
if (!(c='y' || c='Y')) return;
```

```
flag = flag ? 0 : 1;
```

```
length = (length < BUFSIZE) ? length : BUFSIZE;
```

Consistenza e idiomi

- Consistenza nel codice = usare la stessa forma per snippet di codice che hanno un significato simile
  - *identazione e uso delle parentesi*
  - *struttura dei cicli*
  - *struttura delle decisioni*
  - *…*
- Convenzioni di stile
  - *es. Linux Kernel Coding Style*

Uso delle graffe

- Usare le graffe per racchiudere statement, anche uno solo
  - *In futuro, più statement possono essere aggiunti*
  - *Si evita il problema del “dangling else”*

Dangling else

```
if (month==FEB) {
   if (year%4 == 0)
      if (day > 29)
         legal = FALSE;
   else
      if (day > 28)
         legal = FALSE;
}
```

```
if (month==FEB) {
   if (year%4 == 0) {
      if (day > 29)
         legal = FALSE;
   } else {
      if (day > 28)
         legal = FALSE;
   }
```

… ancora meglio

```
if (month==FEB) {
   if (year%4 == 0)
      if (day > 29)
         legal = FALSE;
   else
      if (day > 28)
         legal = FALSE;
}
```



```
if (month==FEB) {
   if (year%4 == 0) {
      if (day > 29)
         legal = FALSE;
   } else {
      if (day > 28)
         legal = FALSE;
   }
```

```
if (month == FEB) {
   int nday;

   nday = 28;
   if (year%4 == 0)
      nday = 29;

   if (day > nday)
      legal = FALSE;
}
```

Cicli idiomatici

```
i=0;
while (i <= n-1)
   array[i++] = 1.0;
```

```
for (i=0; i<n; )
   array[i++] = 1.0;
```

```
for (i=n; --i >= 0; )
   array[i] = 1.0;
```

```
for (i=0; i<n; i++)
   array[i] = 1.0;
```

Decisioni multiple

```
if (condition_1) {
   ...
} else if (condition_2) {
   ...
} else if...
   ...
} else {
   // caso di default
}
```

Evitare codice “diagonale”

```
if (argc==3)
   if ((fin = fopen(argv[l], "r")) != NULL)
      if ((fout = fopen(argv[2], "w")) != NULL)
         while ((c = getc(fin)) != EOF) {
            putc(c, fout);
            close(fin );
            fclose(fout);
         }
      else
         printf("Can't open output file %s\n", argv[Z]) ;
   else
      printf("Can't open input file %s\n", argv[l]);
else
   printf ("Usage: cp inputfile outputfile\n");
```

… meglio

```
if (argc != 3) {
   printf ("Usage: cp inputfile outputfile\n");
} else if ((fin=fopen(argv[l], "r")) == NULL) {
   printf("Can't open input file %s\n", argv[l]);
} else if ((fout = fopen(argv[2], "w")) == NULL) {
   printf("Can't open output file %s\n", argv[2]) ;
   fclose(fin);
} else {
   while ((c = getc(fin)) != EOF)
   putc(c, fout);
   fclose(fin);
   fclose(fout);
}
```

switch

- Ogni case ha un break
- Se un case non ha un break, va segnalato opportunamente
- Anche il default ha un break

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

Numeri magici

Un numero magico è

- una costante
- una dimensione di array
- una posizione in un array
- un fattore di conversione
- qualunque letterale

che appare nelle istruzioni di un programma

I numeri magici riducono la leggibilità e minano la robustezza di un programma.

Denominare i numeri magici

- Ogni numero diverso da 0 e 1 dovrebbe essere fattorizzato in una costante simbolica
- Un numero non fornisce informazioni a chi legge il programma
  - *Da dove deriva? Perché è importante?*
- Una costante simbolica fornisce informazioni aggiuntive attraverso il nome
  - *La modifica del valore di una costante si propaga in tutto il programma → più facile la manutenzione*

Esempio

```
fac = lim / 20;   /* set scale factor */
if (fac < 1)
   fac = 1;

/* generate histogram */
for (i = 0 , col = 0 ; i < 27; i++, j++) {
   col += 3;
   k = 21 - (let[i] / fac);
   star = (let[i] == 0) ? ' ' : '*';
   for (j = k; j < 22; j++)
      draw(j, col, star) ;
}
draw(23, 2, ' ') ;
for (i = 'A'; i <= 'Z'; i++)
   printf("%c ",i) ;
```

Fattorizziamo le costanti

```
enum {
  MINROW   = 1,              /* top edge */
  MINCOL   = 1,              /* left edge */
  MAXROW   = 1,              /* bottom edge */
  MAXCOL   = 80,             /* right edge */
  LABELROW = 1,              /* position of labels */
  NLET     = 26,             /* size of alphabet */
  HEIGHT   = MAXROW-4,       /* height of bars */
  WIDTH    = (MAXCOL-1)/NLET   /* width of bars */
};
```

Demistifichiamo il codice

```
fac = (lim + HEIGHT-1) / HEIGHT;   /* set scale factor */
if (fac < 1)
   fac = 1;

/* generate histogram */
for (i = 0; i < NLET; i++) {
   if (let[i] == 0)
      continue;
   for (j = HEIGHT - let[i]/fac; j < HEIGHT; j++)
      draw(j+1+LABELROW, (i+1)*WIDTH, '*') ;
}
draw(MAXROW-1, MINCOL+1, ' ') ;
for (i = 'A'; i <= 'Z'; i++)
   printf("%c ",i);
```

Caratteri costanti

L’espressione

```
(c >= 65 && c <= 90)
```

è equivalente a

```
(c >= 'A' && c <= 'Z')
```

ma l’ultima è certamente più leggibile.

La costante nulla

La costante 0 ha molti tipi. Le istruzioni

```
ptr = 0;
name[i] = 0;
x = 0;
```

sono equivalenti a:

```
ptr = NULL;
name[i] = '\0';
x = 0.0;
```

ma queste ultime sono più leggibili.

L’operatore sizeof

- L’operatore sizeof restituisce la dimensione in byte di una variabile o di un tipo
  - ```
    char buf[1024];
    fgets(buf, sizeof(buf), stdin);
    ```
  - *Evita di riscrivere le dimensioni di un oggetto, quando sono note al compilatore*
- Per conoscere il numero di elementi di un array si può scrivere

```
sizeof(array) / sizeof(array[0]);
```
```

Commenti

- I commenti sono intesi per aiutare il lettore di un programma
  - *Non aiutano se dicono le stesse cose del codice*
  - *Non aiutano se sono in contraddizione con il codice*
  - *Non aiutano se distraggono il lettore con inutili svolazzi tipografici*

Commenti perfettamente inutili

```
/*
 * default
 */
default:
   break;

/* return SUCCESS */
   return SUCCESS;

zerocount++; /* Increment zero entry counter */

/* Initialize "total" to "number_received" */
node- >total = node->number_received;
```

Commenti ridondanti

Un uso appropriato dei nomi limita la necessità dei commenti

```
while ((c = getchar()!=EOF && isspace(c))
                    /* skip whitespace */
if (c == EOF)        /* end of file */
   type = endoffile;
else if (c == '(')   /* left paren */
   type = leftparen;
else if (c == ')')   /* right paren */
   type = rightparen;
else if (c == ';')   /* semicolon*/
   type = semicolon;
else if (is_op(c))   /* operator */
   type = operator;
...
```

Commentare funzioni

- I commenti alle funzioni sono indispensabili
  - *Documentazione del codice*
  - *Indicano cosa fa la funzione*
  - *Indicano il significato dei parametri*
  - *Aggiunge informazioni non desumibili dalla firma della funzione*

Commentare o riscrivere?

- I commenti sono utili a chiarire snippet di codice confusi o non usuali
- ma se il commento stesso diventa troppo lungo e intricato, meglio riscrivere il codice!

Non contraddire il codice

- I programmi tendono a evolvere nel tempo
- Se i commenti non evolvono insieme, possono disallinearsi con il codice

Chiarire, non confondere

```
int strcmp(char *s1, char *s2)
/* string comparison routine returns -1 if s1 is */
/* above s2 in an ascending order list, 0 if equal */
/* 1 if s1 below s2 */
{
   while(*s1==*s2){
      ...
```

```
int strcmp(char *s1, char *s2)
/* strcmp: return <0 if s1<s2, >0 if s1>s2, 0 if equal */
/* ANSI C, section 4.11.4.2 */
{
   while(*s1==*s2){
      if(*sl=='\O') return(0);
         ...
```

K&R coding style

Indentazione

- Ampia tabulazione
  - *4 o 8 spazi*
  - *Scoraggia l'annidamento*
- Si indentano
  - *statement nel corpo di una funzione*
  - *statement nei blocchi*
  - *statement nei case*
- NON si indentano
  - *statement in una switch*

```
int compare(int x, int y)
{
        if (x < y) {
                return -1;
        } else if (x > y) {
                return 1;
        } else {
                return 0;
        }
}
```

Indentazione

- Ampia tabulazione
  - *4 o 8 spazi*
  - *Scoraggia l'annidamento*
- Si indentano
  - *statement nel corpo di una funzione*
  - *statement nei blocchi*
  - *statement nei case*
- NON si indentano
  - *statement in una switch*

```
int foo(int bar)
{
        switch (bar) {
        case 0:
                ++bar;
                break;
        case 1:
                --bar;
        default: {
                bar += bar;
                break;
        }
        }
}
```

Parentesi graffe (“*brace*”)

- A capo nella definizione di una funzione
- Sulla stessa linea negli altri casi

```
int digits[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
};

int compare(int x, int y)
{
        if (x < y) {
                return -1;
        } else if (x > y) {
                return 1;
        } else {
                return 0;
        }
}
```

Controllo del flusso di esecuzione

```
void bar()
{
        do {
        } while (true);
}
```

```
void foo2()
{
        if (1) {
                return;
        }
        if (1) {
                return;
        } else if (0) {
                return;
        } else {
                return;
        }
}
```

Controllo del flusso di esecuzione

```
void bar()
{
        do {
        } while (true);
}
```

```
void foo(int state)
{
        if (1)
                return;
        if (1)
                return;
        else if (0)
                return;
        else
                return;
}
```

Statement multi-linea

```
void f(int arg1, int arg2,
                int arg3, int arg4,
                int arg5, int arg6)
{
}
```

```
enum Example {
                CANCELLED,
                RUNNING,
                WAITING,
                FINISHED
};
```

Statement multi-linea

```
void foo()
{
        bar(100, 200, 300, 400,
            600, 700, 800, 900);
}
```

```
int foo()
{
        int sum = 100 + 200
                        + 500 +
600 + 700;
        int product = 1 * 2 * 3
                          * 7 *
8;
        return product / sum;
}
```

Statement multi-linea

```
int compare(int argument,
            int argument2)
{
              return arg > arg2 ?
                                    100000 :
200000;
}
```

```
static char* string =
                          "text text text";

void foo()
{
              for (int i = 0; i < 10;
          i++) {
              }
              const char* char_string;
              char_string =
                          "text text text";
}
```

Nomi

- MY_CONSTANT
- my_variable o myvar
- my_function o myfun

- I nomi devono essere preferibilmente in inglese
  - *In ogni caso, non mescolare italiano e inglese,*
  - *ma soprattutto, non scrivere in Inglish, Italglish, Engliano, Italish*

Commenti

```
/* i commenti su linee dedicate documentano
 * blocchi di codice
 */

int x = 0; // brevi commenti in linea
int y = 0; /* commenti in linea
              piu' lunghi */

// puts("codice temporaneamente eliminato");
```

The Ten Commandments for C Programmers

Henri Spencer

1. Thou shalt run lint frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine.
2. Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.
3. Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.
4. If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.
5. Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest ``foo'' someone someday shall type ``supercalifragilisticexpialidocious''.
6. If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest ``it cannot happen to me'', the gods shall surely punish thee for thy arrogance.
7. Thou shalt study thy libraries and strive not to re-invent them without cause, that thy code may be short and readable and thy days pleasant and productive.
8. Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.
9. Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.
10. Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that ``All the world's a VAX'', and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.