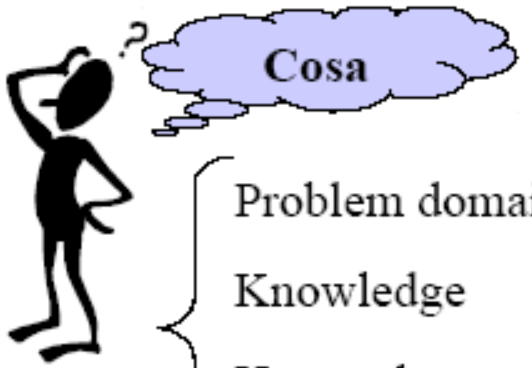
Stile di Programmazione

“A computer will do what you tell it to do, but that may be much different from what you had in mind”

Joseph Weizenbaum

Time

Dal "cosa" al "come" …



- Problem domain
- Knowledge
- Human language
- Goals, needs, desires

«La scienza del software studia il passaggio dal "cosa" al "come" »

E.A. Feigenbaum

Communications of the ACM, Maggio 1996

Motivazioni

- Un programma dovrebbe racchiudere una soluzione pensata dal programmatore quindi rappresenta una forma di conoscenza
- Se scritto bene, è **comprensibile** per la macchina ed è compilato e quindi eseguito
- I programmi complessi, più generalmente, sistemi software, sono soggetti a manutenzione ed evoluzione, spesso da programmatori diversi e differenti da quelli che lo hanno sviluppato originariamente
- Se scritti male, **non** possono essere **manutenuti** ed **estesi**

Motivazioni

- Per essere scritti bene, I sistemi software devono essere comprensibili ad altri programmatori, cioè devono
  1. Usare nomi ed identificatori significativi
  2. Usare forme convenzionali a tutti i programmatori
  3. Usare formattazione consistente e semplice
  4. Avere una logica immediata e organizzazione chiara e condivisibile
  5. Essere documentati e commentati in modo da facilitarne la comprensione

Nomi ed identificatori significativi

- I nomi delle cose devono rappresentarle: quando pronunciamo *mamma* tutti pensiamo ad una persona di sesso femminile in rapporto esclusivo alla prole, e con una intensa accentuazione affettiva
- Nomi e identificatori di variabili, funzioni, procedure e strutture dati devono essere esplicativi del loro uso e del loro scopo.
- Deve essere coinciso e mnemonico
- Maggiore è il numero di funzioni, procedure e blocchi di istruzioni che usano un nome, più informativo deve essere il nome perchè sia comprensibile in differenti ambiti

Nomi ed identificatori significativi

Ad esempio:

- Una variabile contatore è usata nell'ambito di una struttura iterative, quindi non è necessario che sia particolarmente informative

  int i=0, int k=0

- variabili usate come parametri di funzioni e procedure dovrebbero essere **auto-esplicativi**

  int calcolo(float media, int massimo)

- identificatori di funzioni o di dichiarazioni di tipo (*typef*) devono essere **particolarmente informative**

  typedef persona,
  typedef account,
  int calcolaBMI( )

Nomi ed identificatori significativi

- Il C è case sensitive, quindi

  int variabileuno è differente da int Variabileuno

- ma è buona norma non utilizzare variabili con lo stesso nome che differiscono solo per alcune lettere minuscole o maiuscole

  int variabileuno, Variabileuno, variabileUno;

Nomi ed identificatori significativi

Quindi per scegliere il nome di

•variabile o funzioni con pochi ambiti d'uso possiamo usare **nomi brevi**

- i,j,k per indici
- s,t per stringhe
- c per caratteri
- n,m per interi
- x,y per numeri frazionari
- p,q per puntatori

Nomi ed identificatori significativi

Quindi per scegliere il nome di

- variabile o funzioni con molti ambiti d'uso possiamo usare **nomi lunghi ed esplicativi, e commentate**

```
int max_bmi_maggiorenni = 0; // valore massimo del BMI
                             per i maggiorenni
```

Nomi ed identificatori significativi

Ad esempio:

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

Nomi ed identificatori significativi

Ad esempio:

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

```
for(i = 0; i < n_elem; i++) {
      elem[i] = i ;
}
```

Nomi ed identificatori significativi

Variabili e funzioni correlati devono avere nomi simili.

```
int calcola_massimo()
int calcola_minino()
int prendi_minimo()  NO

float max_bmi = 0;
float min_bmi = 0;
float valore_basso_bmi  NO
int num_studenti = 0;
int eta_stud = 0;  NO
int eta_studenti = 0;
```

stud e **studenti** sono due nomi diversi, che si riferiscono allo stesso concetto

Nomi ed identificatori significativi: Numeri magici

- Un numero (senza fornire ulteriori dettagli) non fornisce informazioni a chi legge il programma
- Da dove deriva? Come è stato calcolato?
- Un numero magico è una costante, una dimensione di array, una posizione in un array, un fattore di conversione, qualunque letterale.
- Tutti i numeri presenti in un programma (con esclusione degli 0 e 1) dovrebbe essere sostituiti da **costanti simboliche**
- La modifica del valore di una costante si propaga in tutto il programma, senza la necessità di modificare tutte le istruzioni in cui quel numero è usato.

Numeri magici

- Cosa non fare:

```
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) ;
```

Numeri magici

- Cosa fare:

```
enum{

MINROW = 1, /* top edge*/
MINCOL = 1, /* leftedge*/
MAXROW = 1, /* bottom edge*/
MAXCOL = 80, /* right edge*/
LABELROW = 1, /* position of labels*/
NLET = 26, /* sizeof alphabet*/
HEIGHT = MAXROW-4, /* heightof bars*/
WIDTH = (MAXCOL-1)/NLET /* widthof bars*/

};
```

Si procede definendo le costanti simboliche nel programma e sostituendole al codice

Numeri magici

- Cosa fare:

```
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);
```

Forme convenzionali

Vi sono regole comuni ai programmatori

- le costanti sono riportate in maiuscolo associando il loro valore

```
#define STUDENTI 150
```

- usare l'underscore o maiuscolo per rendere leggibili nomi troppo lunghi

```
int maxbmimaggiorenni → NO
int max_bmi_maggiorenni → SI
int maxBMIMaggiorenni → SI
```

Forme convenzionali

Vi sono regole comuni ai programmatori

- i nomi delle funzioni devono descrivere le azioni principali da eseguire

  printf() → stampa
  calcolaBMI() → calcola il BMI
  calculateAverage() → calcola la media

- I nomi di funzioni che restituiscono un valore booleano devono aver prefisso is

  isDigit(char)
  isAlNum(char)
  isAdult(age)

Forme convenzionali

Vi sono regole comuni ai programmatori

- i nomi delle funzioni devono sintetizzare ciò che realmente fanno e cosa ci aspettiamo come risultato, altrimenti possono essere usate erroneamente:

```
int isAdult(int age) {
//cosa ci aspettiamo dentro questa funzione?
}
```

Forme convenzionali

Vi sono regole comuni ai programmatori

•i nomi delle funzioni devono sintetizzare ciò che realmente fanno e cosa ci aspettiamo come risultato, altrimenti possono essere usate erroneamente:

```
int isAdult(int age) {
//cosa ci aspettiamo dentro questa funzione?
}
```

```
int isAdult(int age) {          NO
if(age>25) return 1;
else return 0;
}
```

Forme convenzionali

Vi sono regole comuni ai programmatori

•i nomi delle funzioni devono sintetizzare ciò che realmente fanno e cosa ci aspettiamo come risultato, altrimenti possono essere usate erroneamente:

```
int isAdult(int age) {
//cosa ci aspettiamo dentro questa funzione?
}
```

```
int isAdult(int age) {        SI
if(age>18) return 1;
else return 0;
}
```

Forme convenzionali

Esempio

```
#define TRUE 0
#define FALSE 1

if((ch= getchar()) == EOF)
        non_e_finito= FALSE;
```

getchar():è una funzione che legge un carattere da file
ch: variabile che memorizza un carattere
EOF: indicatore di fine file

Il nome `non_e_finito` è incoerente con cosa rappresenta.

Forme convenzionali

Esempio

```
#define TRUE 0
#define FALSE 1

if((ch= getchar()) == EOF)
      e_finito= TRUE;
```

`e_finito` è coerente con cosa rappresenta e rende leggibile e comprensibile il programma.

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- limitare l'uso di forme negate

```
if(!(a==0) || !(b==0))     NO
```

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- limitare l'uso di forme negate

```
if((a!=0) || (b!=0)) SI
```

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- evitare espressioni lunghe usando spazi, fattorizzazione di termini o raggruppando termini con parentesi

```
a!=0&&b+1==0  NO
```

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- evitare espressioni lunghe usando fattorizzazione di termini o raggruppando termini con parentesi

```
(a!=0) && (b==-1)  SI
```

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- evitare espressioni lunghe usando fattorizzazione di termini o raggruppando termini con parentesi
- Esempio: espressione che calcoli se l'anno è bisestile

```
leap_year = y % 4== 0&& y % 100!= 0|| y % 400== 0;     NO
```

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- evitare espressioni lunghe usando fattorizzazione di termini o raggruppando termini con parentesi
- Esempio: espressione che calcoli se l'anno è bisestile

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

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- più espressioni possono essere equivalenti e non sempre la più chiara corrisponde alla più breve

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

NO, poco leggibile perchè troppo compatto

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- più espressioni possono essere equivalenti e non sempre la più chiara corrisponde alla più breve

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

SI, meno compatto ma più comprensibile

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- sebbene messi a disposizione dai linguaggi di programmazione, gli operatori di abbreviazione (come l'operatore ternario di selezione) sono consigliati per espressioni semplici.

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

NO, sebbene sia una sola istruzione, non è comprensibile a chi la intepreterà in un successivo momento.

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- sebbene messi a disposizione dai linguaggi di programmazione, gli operatori di abbreviazione (come l'operatore ternario di selezione) sono consigliati per espressioni semplici.

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

SI

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- sebbene messi a disposizione dai linguaggi di programmazione, gli operatori di abbreviazione (come l'operatore ternario di selezione) sono consigliati per espressioni semplici.

```
printf('Indica posizione del vettore da modificare e nuovo valore:');
scanf("%d %d", &yr, &profit[yr]);
```

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- sebbene messi a disposizione dai linguaggi di programmazione, gli operatori di abbreviazione (come l'operatore ternario di selezione) sono consigliati per espressioni semplici.

```
printf('Indica posizione del vettore da modificare e nuovo valore:');

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

Legge due variabili di tipo intero. Ci aspetteremmo che il valore letto dalla variabile yr venga utilizzato per modificare la posizione del vettore

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- sebbene messi a disposizione dai linguaggi di programmazione, gli operatori di abbreviazione (come l'operatore ternario di selezione) sono consigliati per espressioni semplici.

```
printf('Indica posizione del vettore da modificare e nuovo valore:');

scanf("%d %d", &yr, &profit[yr]);        NO
```

Questo non accade, perché il linguaggio C valuta tutta l'intera istruzione scanf nello stesso momento. Dunque il valore yr nell'indice del vettore profit non viene modificato

Logica immediata e organizzazione chiara

Le espressioni matematiche devono essere scritte in modo che la loro logica sia trasparente:

- sebbene messi a disposizione dai linguaggi di programmazione, gli operatori di abbreviazione (come l'operatore ternario di selezione) sono consigliati per espressioni semplici.

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

Separando due diverse istruzioni anziché una, il codice diventa comprensibile e funzionante.

Logica immediata e organizzazione chiara

L’indentazione può migliorare la scrittura e comprensibilità delle espressioni matematiche e più generalmente migliorano la formattazione di tutto il codice.

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

NO

Logica immediata e organizzazione chiara

L’indentazione può migliorare la scrittura e comprensibilità delle espressioni matematiche e più generalmente migliorano la formattazione di tutto il codice.

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

SI
```

Logica immediata e organizzazione chiara

Alcuni esempi

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

Logica immediata e organizzazione chiara

Alcuni esempi

```
if (c!='y‘ && c!=‘Y’)
   return;
```

Logica immediata e organizzazione chiara

Alcuni esempi

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

Logica immediata e organizzazione chiara

Alcuni esempi

```
if (length > BUFSIZE)
   length = BUFSIZE;
```

- Quando si utilizzano operatori logici (&&, ||) **è utile aggiungere degli spazi** aggiuntivi per migliorare la leggibilità.
- Spesso è utile anche con **gli operatori relazionali (>, <, >=, etc.)**

Formattazione consistente

Parentesi graffe usate per delimitare blocchi di istruzioni:

•Devono essere usate sempre allo stesso modo, cioè messe sullo stesso rigo della istruzione

```
for(n=n+1; n<100; n++){
```

oppure al rigo successivo

```
for(n=n+1; n<100; n++)
{
```

e quella scelta deve essere applicata sempre.

Formattazione consistente

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

L'espressione **if(day > 28)** viene valutata come alternativa a **if(day > 28)**, generando un comportamento inaspettato!

Formattazione consistente

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

L'uso delle { } in più rende il codice leggibile e soprattutto evita errori inattesi!

Formattazione consistente

Blocchi di istruzioni che hanno una funzionalità simile, ad esempio, strutture di cicli e strutture di decisione, devono essere scritti nella medesima forma usando indentazione e parentesi.

A tal proposito sviluppatori del linguaggio hanno fissato alcune regole di formattazione:
https://www.kernel.org/doc/Documentation/process/coding-style.rst

Formattazione consistente

Per le strutture a cicli è fuorviante usare:

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

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

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

Formattazione consistente

Per le strutture a cicli è fuorviante usare:

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

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

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

rispetto a cui si deve usare la seguente forma

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

Formattazione consistente

Per le strutture a decisione è fuorviante usare una eccessiva indentazione, che renderebbe il codice in forma «diagonale» e rende difficoltoso capire l'esito di ciascuna decisione.

```
if(argc==3)
   if((fin = fopen(argv[1], "r")) != NULL)
      if((fout = fopen(argv[2], "w")) != NULL)
         while((c = getc(fin)) != EOF) {
            putc(c, fout);
            fclose(fin );
            fclose(fout);
         }
      else
         printf("Can't open output file %s\n", argv[Z]) ;
else
      printf("Can't open input file %s\n", argv[1]);
else
   printf("Usage: cp inputfile outputfile\n");
```

Formattazione consistente

Per le strutture a decisione si usa una organizzazione a sequenza, dove le singole decisioni sono riportate una dopo l'altra «verticalmente»

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

Formattazione consistente

… che renderebbe leggibile e comprensibile il codice precedente

```
if(argc != 3)
{
    printf ("Usage: cp inputfile outputfile\n");
} else     if((fin=fopen(argv[1], "r")) == NULL)
{
    printf("Can't open input file %s\n", argv[1]);
} 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);
}
```

Formattazione consistente

Stessa cosa dicasi per la struttura switch-case la cui organizzazione deve essere così scritta.

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

- Ogni case deve avere il suo ‘break’
- Anche il default deve avere il suo ‘break’
- Quando il ‘break’ non è presente, bisogna esplicitamente indicarlo nei commenti!

Essere documentati e commentati

A proposito dei commenti:

- Non aiutano se dicono le stesse cose del codice

  es.:
  ```
  count++; //incremento il valore del contatore
  default: // questo è il ramo di default
  i=0; // inizializzazione della variabile i
  ```

- Non aiutano se sono in contraddizione con il codice, devono essere aggiornati se vi sono modifiche al codice e devono evolvere con il codice
- Non aiutano se distraggono il lettore con inutili particolarità tipografiche
- Devono essere sintetici
- Devono essere separati da almeno uno spazio o tabulazione dalla istruzione che descrivono

Essere documentati e commentati

A proposito dei commenti:

- Ogni programma deve cominciare con un commento che deve contenere:

- Nome, cognome di tutti i componenti del gruppo (o del responsabile o dell'azienda)
- Data di inizio della stesura del programma
- Nome del file, per garantire chiarezza nella stampa
- Testo del problema, scritto in maniera più chiara e descrittiva possibile
- Assunzioni aggiuntive o limitazioni del programma
- Eventuali note sugli algoritmi usati.

Essere documentati e commentati

I commenti alle funzioni sono indispensabili:

- Indicano cosa fa la funzione
- Indicano il significato dei parametri
- Aggiunge informazioni non desumibili dalla firma della funzione
- Chiariscono passaggi fondamentali nell'implementazione

Essere documentati e commentati

Non chiarisce cosa la funzione realmente fa

```
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){
...
```

SI

Suggerimenti finali

- **Inizializzare** sempre le variabili prima di utilizzarle.
- I vari compilatori C potrebbero inizializzare il contenuto delle variabili con valori indicativi, **non sempre** con i valori che il programmatore si aspetta.
- Un errore dovuto ad una variabile non inizializzata si verifica **durante l'esecuzione** del codice, il che non aiuta il programmatore nella soluzione del problema.
- Inoltre, per rendere il codice portabile su varie piattaforme hardware, o semplicemente per poter rigenerare il codice con una versione differente di compilatore è importante inizializzare sempre le variabili prima di utilizzarle.