Fondamenti di Informatica – 082746
Scuola di Ingegneria Industriale e dell'Informazione

Cristiana Bolchini

Linee guida - Scelte stilistiche

in relazione alla programmazione in linguaggio C - ANSI 89

Potreste pensare che la motivazione per seguire delle linee guida per la scrittura del codice sia qualcosa che serve per far contenta/o la/il docente, ma chi beneficia maggiormente di questo siete voi. Impegnarsi per scrivere del codice ordinato, ben strutturato consentirà di collaudarlo più facilmente, di introdurre meno bug o di trovarli più facilmente. Di fatto comunque, scrivere del codice "bene" non richiede più fatica di scriverlo male, per cui: perché no?

Le indicazioni qui riportate presentano alcuni aspetti di interesse cui ci atterremo in questo corso. Come per tutte le cose complicate, non esiste un "ottimo" assoluto, e neanche una lista esaustiva che copra ogni situazione. Ci sono però scelte migliori ed altre peggiori, e lo scopo di questa guida è di farvi propendere per le prime. Nella stragrande maggioranza dei casi sono linee guida adottate comunemente perchè vincenti, in alcuni casi si tratta di preferenze mie.

Può succedere che in base a precedenti esperienze abbiate adottato delle soluzioni diverse, proprio perchè non esiste un ottimo assoluto. Ovunque lavorerete dovrete adottare le linee guida stabilite in quell'ambito: in questo corso, dovreste fare lo sforzo di seguire lo stile di scrittura del codice qua riportato. Questo consentirà una maggior uniformità, anche tra di voi, semplificando la comprensione e il confronto del codice durante le vostre discussioni. Se avete opinioni fortemente diverse supportate da buone motivazioni (tra cui non rientra "ho sempre fatto così"), venite a parlarne.

Infine, alcune scelte di utilizzare o meno dei costrutti sono motivate dall'approccio didattico utilizzato, finalizzato ad illustrare e far apprendere i metodi di soluzione dei problemi ritenuti più adatti ed efficaci, discussi durante il corso. Il riferimento è allo standard ANSI 89.

Indentazione e spazi

Nella scrittura del codice (sia su carta, sia con il calcolatore) deve essere facile comprendere cosa si voglia fare. Di fatto, il linguaggio C non pone vincoli sulla spaziatura e sull'andare a capo, per cui si può scrivere un intero programma su una singola riga. Il compilatore non avrebbe problemi e se non ci sono errori di sintassi, crea un eseguibile. Lo stesso eseguibile creato a partire da un programma scritto andando a capo a ogni istruzione e indentando le istruzioni. La prima soluzione sarebbe per noi abbastanza difficile da comprendere, la seconda - si spera - un po' meno.

- **indentazione**: utilizzate l'indentazione e la spaziatura. Indentare il codice aggiunge leggibilità alla struttura del programma ed al flusso di controllo.
  - impostate la tabulazione a 2 spazi, è sufficiente per evidenziare l'indentazione, senza generare righe vuote e lunghe;
  - incrementate il livello di indentazione ogni volta che aprite una parentesi graffa `{`, e decrementatelo ogni volta che la chiudete `}`;
  - andare a capo dopo ogni `{`;
  - scrivete una istruzione per riga.

```
/* codice poco leggibile  */
while(x < y){
if(y %% BASE == 0){
x = modify(y); y++;
} else x++;
}
```

```
/* codice leggibile  */
while(x < y){
  if(y %% BASE == 0){
    x = modify(y);
    y++;
  } else
    x++;
}
```

- **righe lunghe**: quando una riga va oltre i 100 caratteri, andate a capo ed allineate la seconda riga alla prima.

```
result = longFunctionName(argument, N * expression * variable) + variable -
longerFunctionName() + otherFunction(variable);
```

```
result =  longFunctionName(argument, N * expression * variable) +
          variable - longerFunctionName() + otherFunction(variable);
```

- **righe vuote**: introducete delle linee vuote per separare parti di codice e sottoprogrammi.
- **spazi**: mettete spazi bianchi tra gli operatori e i loro operandi. Utilizzate le parentesi per indicare la precedenza qualora non fosse immediata.

```
r = (-b+subp(a))/2*a;
```

```
r = (-b + subp(a)) / (2 * a);
```

Nomi

Scegliete nomi significativi per variabili e simboli di define: devono in qualche modo suggerire a chi legge il significato dell'informazione/dato rappresentato.

- **define**: utilizzate le direttive define per evidenziare aspetti caratterizzanti del programma come ad esempio la dimensione di un array, evitando di spargere numeri magici nel codice. Questo aumenta la leggibilità e la manutebilità del codice.

```
...
while(val != 55){
  ...
}
```

```
#define STOP 55
...
while(val != STOP){
  ...
}
```

- **sostantivi per le variabili**: per le variabili la domanda è "cosa è?" Utilizzate un sostantivo (ad esempio `prezzo`, `ora`, ...) ed eventualmente aggiungete dei modificatori per specializzare meglio (ad esempio `oraApertura`, `oraChiusura`). Non mettete il nome del tipo nella variabile (`intVal`) ed evitate nomi di una singola lettera quali `a` (se non per i contatori dei cicli, gli indici degli array, le coordinate `x` e `y`). Non chiamate una variabile `l` in quanto si confonde facilmente con il numero uno.
- **verbi per i sottoprogrammi**: per i sottoprogrammi la domanda è "cosa fa?" Per questo motivo i sottoprogrammi sono ben rappresentati da verbi (ad esempio `rimuoviPari`, `eliminaPunteggiatura`). I sottoprogrammi utilizzati prevalentemente per il valore restituito (ossia le funzioni) dovrebbero avere un identificativo che indica la proprietà che viene restituita (ad esempio `ePrimo`, `trovaMinimo`).
- **maiuscole/minuscole**: utilizzate identificatori maiuscoli solo per i simboli delle direttive define (`STOP`), minuscole per variabili e sottoprogrammi, eventualmente utilizzando la cosiddetta "notazione a cammello" (`val`, `valMax`), minuscole per il nome dei tipi definiti dall'utente terminanti con `_t`.
- **nuovi tipi**: introducete nuovi tipi ogni volta che avete un dato strutturato, non fatelo per rinominare tipi standard.

```
struct strutturaPunti {
  int x, y;
  float value;
}
...

struct strutturaPunti p;
```

```
typedef struct strutturaPunti {
  int x, y;
  float value;
} point_t;
...
point_t p;
```

```
typedef char string_t[DIM+1];
```

Tipo e visibilità delle variabili

Si adotta lo standard ANSI C 89 per quanto riguarda i tipi base supportati e il non mischiare dichiarazioni e istruzioni, limitandoci a dichiarare variabili solo all'inizio del (sotto)programma.

- **Tipo delle variabili**: dichiarare le variabili del tipo opportuno in base al dato che devono memorizzare (ad esempio, utilizzate una variabile di tipo `int` per una variabile che contiene solo valori interi, non un `float` perchè poi viene utilizzata per calcolare altro, ma il suo valore resta sempre intero).
- **Non usate variabili globali**: a meno che non sia indicato dal testo, non fate uso di variabili globali. Nel momento in cui un sottoprogramma deve avere accesso ad informazioni, queste dovrebbero essere scambiate tramite i parametri e i valori restituiti.
- **Dichiarate variabili solo all'inizio del (sotto)programma:** sebbene lo standard C89 consenta di dichiarare variabili all'inizio di ogni blocco delimitato da {}, limitiamo le dichiarazioni in un unico punto.
- **Non dichiarate variabili con visibilità limitata ad un ciclo**: anche nel momento in cui la variabile serve solo per le iterazioni di un ciclo, non dichiaratela all'interno di un ciclo. La precedente indicazione esclude già questa possibilità del C99, ribadita per maggior chiarezza.

```
for(int i = 0; i < N; i++)
```

Spesso si tratta di un automatismo che porta a scrivere:

```
for(int i = 0; i < N; i++)
    /* ciclo 1 */
...
```

```
for(int i = 0; i < N; i++)
/* ciclo 2 */
```

Costrutti

- **inizializzazione delle variabili**: inizializzate le variabili solo prima di usarle, non in fase di dichiarazione. Quando dichiariamo una variabile non c'è motivo per cui debba avere un determinato valore, se non per la logica del programma, cosa che si evidenzia quando la si usa.

```
int sum = 0;
/* perche' sum e' 0 a priori? */
...
while(val != STOP){
    sum += val;
    scanf("%d", &val);
}
```

```
int sum;
...
sum = 0;
while(val != STOP){
  sum += val;
  scanf("%d", &val);
}
```

In questa ottica, non inizializzare la variabile subito dopo le dichiarazioni solo per aderire all'indicazione, re-inizializzando la variabile al termine di un ciclo (invece che all'inizio):

```
int flag;
flag = 0;
...
...
for(i = 0; i < N; i++){
  ...
  if(cond)
    flag = 1;
  ...
  flag = 0;
}
```

```
int flag;
...
...
for(i = 0; i < N; i++){
  flag = 0;
  ...
  if(cond)
    flag = 1;
  ...
}
```

Utilizzare la parte `else` del costrutto `if` per specificare il valore cui inizializzare la variabile, invece di usare un valore predefinito e cambiarlo se non va bene; la semantica è migliore e si evitano istruzioni inutili:

```
var = 1;
if(cond)
  var = 0;
...
```

```
if(cond)
  var = 0;
else
  var = 1;
...
```

Fa eccezione la dichiarazione di teste di lista, in quanto all'inizio sono sempre vuote, per cui l'inizializzazione è sempre necessaria e sempre a `NULL`.

```
list_t * testa = NULL;
```

- **array di caratteri gestiti come stringhe**: utilizzate una direttiva `define` per specificare il numero di caratteri "utili" di una stringa, e dimensionare l'array di un elemento in più per evidenziare lo spazio necessario per il terminatore.

```
#define L 30
...
char voc[L+1];
```

- **`while` vs `do-while`**: utilizzate il tipo di ciclo opportuno (a condizione iniziale rispetto a quello a condizione finale): sebbene sia possibile ottenere l'effetto desiderato con entrambi i costrutti, uno è il costrutto corretto, l'altro è una forzatura.

```
/* costrutto inappropriato */
scanf("%d", &val);
while(val < MIN || val > MAX)
   scanf("%d", &val);
```

```
/* costrutto appropriato */
do
  scanf("%d", &val);
while(val < MIN || val > MAX);
/* costrutto appropriato */
scanf("%d", &val);
while(val < MIN || val > MAX){
  printf("valore nell'intervallo %d
%d\n", MIN, MAX);
  scanf("%d", &val);
}
```

```
/* costrutto inappropriato */
do {
  val = getval();
  if(val != STOP){
    sottop(val);
    n++;
  }
}while(val != STOP);
/* stessa valutazione già fatta */
```

```
/* costrutto appropriato */
val = getval();
while(val != STOP){
/* corpo del ciclo non e' ripetuto */
  sottop(val);
  val = getval();
}
```

- **uso del costrutto `for`**: mettete nelle parti del costrutto `for` tutte e sole istruzioni relative alle iterazioni.

```
for(i = 0; i < NUM; scanf("%d", &val[i++]));
```

- **non utilizzate `break` e `continue` per modificare il flusso di controllo di un ciclo, introducete variabili sentinella**: aumenta la leggibilità della condizione che controlla il ciclo, evidenziando che il ciclo potrebbe interrompersi prima.

```
while(i < NUM){
  /* elaborazione */
  ...
  if(cond)
    break;
}
```

```
stop = 0;
while(i < NUM && !stop){
  /* elaborazione */
  ...
  if(cond)
    stop = 1;
}
```

- **evitate chiamate a sottoprogrammi non necessarie**: se il valore restituito da una chiamata viene riutilizzato più volte, evitare di chiamare più volte il sottoprogramma.

```
/* tutte le volte si calcola la
lunghezza */
for(i = 0; i < strlen(s); i++){
  ...
}
```

```
/* calcolo una volta sola */
dim = strlen(s);
for(i = 0; i < dim; i++){
   ...
}
```

Nel caso in cui si scandisca comunque la stringa andando ad analizzarne i caratteri, è inutile scandirla (anche se lo fa il sottoprogramma) per calcolarne la dimensione per poi scandirla per l'analisi, quindi evitare la chiamata al sottoprogramma per la lunghezza della stringa.

```
/* calcolo e successiva scansione */
dim = strlen(s);
for(i = 0; i < dim; i++){
  if(s[i] ... )
}
```

```
/* analisi senza un calcolo */
for(i = 0; s[i] != '\0'; i++){
  if(s[i] ...)
    ...
}
```

- **evitate di sviluppare sottoprogrammi costituiti da una sola istruzione**: chiamare un sottoprogramma ha un costo in termini di prestazioni, quindi non definite sottoprogrammi che non offrono benefici in termini di mascherare complessità (una sola istruzione è comprensibile) e introducono penalità (domandatevi perchè non esiste il sottoprogramma di libreria `elevaAlQuadrato` invece di dover scrivere `val*val`).

Linee guida

- **controllo delle anomalie**: l'obiettivo di un algoritmo è effettuare delle elaborazioni: quando si gestiscono anche i casi d'errore/anomalie nei dati (cosa assolutamente necessaria), questi comportamenti essendo eccezioni, vengono illustrate nella parte `else`.

```
if(condizione_errore)
  /* gestione eccezioni */
else
  /* algoritmo principale */
if(!condizione_errore)
  /* algoritmo principale */
else
  /* gestione eccezioni */
```

- **allocazione memoria**: verificare sempre che l'allocazione sia andata a buon fine

```
vett = malloc(sizeof(int)*num);
if(vett != NULL){
  /* uso della memoria */
  ...
} else {
  printf("Errore allocazione %d interi\n", num);
  /* eventuale gestione */
}
```

- **accesso a file**: verificare sempre che l'accesso ad un file sia andato a buon fine (anche in scrittura, potreste non avere i permessi di accesso in scrittura al file system)

```
fp = fopen(filename, "w");
if(fp != NULL){
  /* accesso ai dati */
  ...
} else {
  printf("Errore accesso al file %s\n", filename);
  /* eventuale gestione */
}
```

- **passaggio array a sottoprogramma**: quando si passa ad un sottoprogramma un array, passare anche il numero di elementi significativi presenti nell'array (potrebbe essere diverso dalla dimensione effettiva dell'array).

```
void sottop(int vett[], int dim){
  int i;
  ...
  for(i = 0; i < dim; i++)
  ...
}
```

Fa eccezione quando si passa una stringa, in quanto è possibile determinare il contenuto significativo dell'array, delimitato dal terminatore. Nel caso di array multidimensionali, passare **tutte** le dimensioni.

- **passaggio di tutti e soli parametri strettamente necessari**: passare ad un sottoprogramma tutti e soli i parametri ritenuti necessari per poter svolgere l'elaborazione. Se un sottoprogramma ha bisogno di un sapere un dato, non si può presumere che esista una direttiva `define` che specifichi tale informazione (e che simbolo ha poi?) ... deve essere un parametro che il chiamante passa al sottoprogramma. Similmente, è inutile farsi passare dei parametri che non servono (per esempio, il numero di caratteri significativi presenti in una stringa).
- **notazione * vs []**: se si utilizza la notazione * per il passaggio di un array, utilizzate la stessa notazione anche nel corpo del sottoprogramma (non utilizzate poi la notazione `[]`), mantenendo consistenza.

```
void sottop(char * s){
  int i;
  ...
  for(i = 0; s[i] != '\0'; i++)
    ...
}
```

```
void sottop(char * s){
  int i;
  ...
  for(i = 0; *(s+i) != '\0'; i++)
    ...
}
void sottop(char s[])
  int i;
  ...
  for(i = 0; s[i] != '\0'; i++)
    ...
}
```

È più comune utilizzate la notazione `[]` quando abbiamo array, * quando abbiamo riferimenti a strutture più articolate.

- **variabili strutturate**: passare/restituire ai sottoprogrammi le variabili strutturate mediante passaggio per indirizzo.
- **codice ridondante**: se lo stesso codice è ripetuto più volte, trovate un modo di rimuoverlo.

```
if(x < y){
  subp(x);
  x++;
} else {
  subp(y);
  x++;
}
```

```
if(x < y)
  subp(x);
else
  subp(y);
x++;
```

- **sottoprogrammi che calcolano più di un risultato**: non definire delle strutture dati che fungono da aggregatori di dati indipendenti per restituire al chiamante più di un dato quando il sottoprogramma deve trasmettere al chiamante più risultati dell'elaborazione: usare il passaggio parametri per indirizzo.

```
typedef struct s {
    int min, max;
    float avg;
} ris_t;
....
ris_t sottop(int v[], int dim){
    ris_r risultati;
    ...
    return risultati;
}
```

```
void sottop(int v[], int dim, int *
min, int * max, float * avg){
    ...
    *min = ...;
    *max = ...;
    *avg = ...;
}
```