Linee Guida per Progetto miniLaska

Alvise Spanò

1 Introduzione

In questo documento forniamo agli studenti alcune linee guida pratiche per sviluppare il progetto. Il rispetto delle tecniche descritte in questo documento sarà tra i *criteri di giudizio* del progetto in sede d'esame. Sebbene non sia necessario seguire tutto questo alla lettera, invitiamo gli studenti ad adottare almeno i principi più importanti nell'implementazione del progetto.

Ad esempio, un progetto che implementa le funzioni di manipolazione della scacchiera seguendo l'approccio ADT (vedi apposita sezione più in basso) verrà valutato più positivamente rispetto ad un progetto che accede alle strutture dati della scacchiera in maniera diretta.

O ancora: un codice che consiste in poche funzioni molto lunghe e complesse verrà valutato meno di un codice ben suddiviso in funzioni corte, chiare e riusate più volte.

2 Approccio ADT

Essendo `miniLaska` un programma chiuso e non una libreria, potrebbe sembrare inutile adottare l'approccio ADT (*Abstract Data Type*) per le strutture dati, poiché non c'è nessuno a cui "nascondere" l'implementazione. Ricapitoliamo i vantaggi dell'approccio ADT:

- l'implementazione del tipo `struct` è "privata", ovvero *nascosta* al chiamante;
- il chiamante è costretto a usare la "libreria" a disposizione, ovvero le funzioni esposte nel file header che manipolano il tipo di dato astratto.

Consigliamo di definire le vostre `struct` per i tipi di dato che manipolate, *non* rappresentate la scacchiera o altre strutture dati importanti con semplici array di `int` o altri tipi base. I vantaggi nell'usare tipi strutturati definiti dal programmatore sono molti: manutenibilità, scalabilità e generalità del codice. Se durante lo sviluppo del programma sentite l'esigenza di cambiare qualche dettaglio di una struttura dati importante, aver seguito l'approccio ADT renderà il vostro codice più resiliente alle modifiche: potrete apportare dei cambiamenti alle vostre `struct` senza dover modificare tutto il programma, limitando gli interventi alle funzioni che compongono la vostra piccola "libreria".

Nel caso di un programma stand-alone come `miniLaska` non è necessario nascondere davvero l'implementazione della `struct`, potete anche definirla direttamente nei file header; la cosa importante è la "libreria" di funzioni: chiamate solo quelle dal resto del programma, facendo finta di non conoscere il contenuto della `struct` ed evitando di accedere ai suoi campi.

3 Valori vs Puntatori

Passare argomenti di tipo `struct` alle funzioni solleva un dilemma: passare per puntatore o passare per valore? Vediamo pro e contro dei 2 approcci.

3.1 Per Valore

Passare per valore significa *copiare*, quindi è meno performante di passare un puntatore. Tuttavia, se la copia interessa un paio di campi, la perdita di performance è trascurabile ed emergono invece i vantaggi: non serve fare la `free()` della `struct` in sé, ma solo degli eventuali dati dinamici interni. Ad esempio:

```c
struct my_array {
    double* pt;
    size_t len;
};

struct my_array create_my_array(size_t len) {
  struct my_array r;
  r.pt = (double*) malloc(len * sizeof(double));
  r.len = len;
  return r;
}

void free_my_array(struct my_array a) {
    free(a.pt);
}

/* questa funzione MODIFICA l'array dinamico dentro la struct my_array
   siamo costretti a ritornarne una nuova per valore */
struct my_array append(struct my_array a, double x) {
  struct my_array new = create_my_array(len + 1);
  memcpy(new.pt, a.pt, a.len * sizeof(double));
  new.pt[new.len - 1] = x;
  free_my_array(a);
  return new;
}

int main() {
  struct my_array a = create_my_array(10);

  a = append(a);  /* devo riassegnare a perché ritorna una nuova struct */

  /* altro codice che usa la variabile a */

  free_my_array(a); /* alla fine chiamo la free */
  return 0;
}
```

Passare per valore le `struct` semplifica la vita in alcuni casi, tuttavia non elimina del tutto la necessità di liberare la memoria e solleva un nuovo problema: ogni qual volta una funzione modifica la `struct my_array` è necessario ritornarne una nuova, sempre per valore. Il problema non è di per sé la copia, poiché essa sarebbe necessaria anche se passassimo `struct my_array*` per puntatore: il problema è che non si più modificare la `struct` passata in input perché è una *copia* e bisogna ritornarne una *nuova* in output, assicurandosi di *riassegnarla* nello scope del chiamante.

Questo approccio porta il codice in direzione dello stile *puro*, ovvero lo stile comunemente ritenuto più sicuro, in cui le funzioni prendono un input le nostre `struct` e producono un output una *nuova* `struct` senza modificare quella in input-> Tuttavia un programma ampiamente *imperativo* si sposa male con lo stile *puro* e il codice chiamante è costretto a fare continui riassegnamenti, il che vanifica gli effetti benefici dello stile puro e introduce anzi maggiori potenziali errori.

3.2 Per Puntatore

Passare per puntatore ha il vantaggio di non fare copie e di togliere al programmatore l'onere di riassegnare continuamente le nuove `struct` prodotte dalle funzioni che manipolano la nostra struttura dati. Portare il codice verso uno stile *impuro* si sposa meglio con un programma *imperativo*: quando l'esigenza è quella di avere strutture dati *modificabili*, tanto vale evitare i compromessi e non tentare di mescolare due stili ortogonali ed incompatibili tra loro. Se devono essere pointer, che pointer siano:

```c
struct my_array {
    double* pt;
    size_t len;
};

struct my_array* create_my_array(size_t len) {
  struct my_array* r = (struct my_array*) malloc(sizeof(struct my_array));
  r->pt = (double*) malloc(len * sizeof(double));
  r->len = len;
  return r;
}

void free_my_array(struct my_array* a) {
    free(a.pt);
    free(a);
}

/* questa funzione MODIFICA l'array dinamico dentro la struct my_array */
void append(struct my_array* a, double x) {
  struct my_array* new = create_my_array(a->len + 1);
  memcpy(new->pt, a->pt, a->len * sizeof(double));
  new->pt[new->len - 1] = x;
  free_my_array(a);
}

int main() {
  struct my_array* a = create_my_array(10);

  append(a);  /* non serve riassegnare nulla: semplicemente modifichiamo l'input */

  free_my_array(a); /* alla fine la free bisogna chiamarla comunque */
  return 0;
}
```

3.3 Approccio Misto

Evitate gli approcci misti, perché sono inconsistenti e poco chiari:

```c
struct my_array {
    double* pt;
    size_t len;
};

struct my_array create_my_array(size_t len) { /*...*/ }

void free_my_array(struct my_array a) { /*...*/ }

void append(struct my_array* a, double x) {
  struct my_array new = create_my_array(a->len + 1);
  memcpy(new.pt, a->pt, a->len * sizeof(double));
  new.pt[new->len - 1] = x;
  free_my_array(*a);  /* devo dereferenziare per chiamare la free_my_array() */
}

int main() {
  struct my_array a = create_my_array(10);  /* tengo la struct come VALORE nello scope del chiamante */
```

```
  append(&a);  /* mi tocca estrarre l'indirizzo quando chiamo funzioni che modificano */

  free_my_array(a);
  return 0;
}
```

In questo modo siamo costretti ad usare l'operatore di reference (`&`) in alcuni casi ed quello di dereference (`*`) in altri, rendendo poco chiaro il codice.

4 Altri Consigli

Seguono ora dei consigli più generali su come impostare il programma.

4.1 Quando Passare le `struct` per Valore

Ci sono dei casi in cui passare le `struct` per valore anziché per puntatore è utile: quando esse rappresentano tipi di dato *piccoli* e *immutabili*. Per esempio, se abbiamo bisogno di un tipo che rappresenta coppie di coordinate possiamo definirlo così:

```
typedef struct { unsigned int x, y; } point_t;

point_t create_point(unsigned int x, unsigned int y) {
  point_t p;
  p.x = x;
  p.y = y;
  return p;
}

point_t move_point(point_t p, unsigned int dx, unsigned int dy) {
  return create_point(p.x + dx, p.x + dy);
}
```

Il tipo `point_t` è piccolo ed immutabile: quando "muoviamo" un punto in realtà non modifichiamo i campi `x` ed `y` ma creiamo un nuovo punto. Questo approccio, detto *puro*, è adatto a questo genere di strutture dati piccole che rappresentano dati di natura matematica o geometrica.

4.2 Riusare il Codice

Uno dei più importanti principi della programmazione è il *riuso di codice*. Scrivere blocchi lunghi e molto simili tra loro è una pratica *error-prone* da evitare: consigliamo di definire *funzioni* per tutto ciò che è *ripetibile*. Quando si scrive codice che consiste in blocchi lunghi e simili, si confrontino tali blocchi con attenzione e si individuino quali sono le espressioni ripetute e quali quelle diverse tra loro. Dopodiché si definiscano funzioni che eseguono quel codice quasi sempre uguale prendendo come argomento gli elementi che sono diversi.

Per esempio, si consideri il seguente scenario: abbiamo una array bidimensionale di `bool` e dobbiamo mettere a `false` le celle indicate da un punto e dalle 8 *adiacenti* ad esso (sopra, sotto, destra, sinistra e a X in diagonale). Teniamo validi il tipo `point_t` e le funzioni definite nella sezione precedente.

```
typedef struct {
  bool* cells;
  size_t w, h;
} table_t;

typedef struct {
  size_t x, y;
} point_t;

void clear_around(table_t* t, point_t p) {
  if (p.x >= 0 && p.x < t->w && p.y >= 0 && p.y < t->h) { /* questa guardia è quasi uguale a quelle sotto
```

```
      t->cells[p.y * t->w + p.x] = false;   /* questo assegnamento è quasi uguale ai quelli sotto */
  }
  if (p.x + 1 >= 0 && p.x + 1 < t->w && p.y >= 0 && p.y < t->h) {
      t->cells[p.y * t->w + p.x + 1] = false;
  }
  if (p.x - 1 >= 0 && p.x - 1 < t->w && p.y >= 0 && p.y < t->h) {
      t->cells[p.y * t->w + p.x - 1] = false;
  }
  if (p.x >= 0 && p.x < t->w && p.y + 1 >= 0 && p.y + 1 < t->h) {
      t->cells[(p.y + 1) * t->w + p.x] = false;
  }
  if (p.x >= 0 && p.x < t->w && p.y - 1 >= 0 && p.y - 1 < t->h) {
      t->cells[(p.y - 1) * t->w + p.x] = false;
  }
    if (p.x + 1 >= 0 && p.x + 1 < t->w && p.y + 1 >= 0 && p.y + 1 < t->h) {
      t->cells[(p.y + 1) * t->w + p.x + 1] = false;
  }
  if (p.x - 1 >= 0 && p.x - 1 < t->w && p.y + 1 >= 0 && p.y + 1 < t->h) {
      t->cells[(p.y + 1) * t->w + p.x - 1] = false;
  }
  if (p.x + 1 >= 0 && p.x + 1 < t->w && p.y - 1 >= 0 && p.y - 1 < t->h) {
      t->cells[(p.y - 1) * t->w + p.x + 1] = false;
  }
  if (p.x - 1 >= 0 && p.x - 1 < t->w && p.y - 1 >= 0 && p.y - 1 < t->h) {
      t->cells[(p.y - 1) * t->w + p.x - 1] = false;
  }
}
```

Possiamo trasformarlo in una forma più elegante con meno ripetizioni:

```
bool is_within(table_t* t, point_t p) {
  return p.x >= 0 && p.x < t->w && p.y >= 0 && p.y < t->h;
}

void clear_around(table_t* t, point_t p) {
  if (is_within(t, p)) {
      t->cells[p.y * t->w + p.x] = false;
  }
  if (is_within(t, move_point(p, 1, 0))) {
      t->cells[p.y * t->w + p.x + 1] = false;
  }
  if (is_within(t, move_point(p, -1, 0))) {
      t->cells[p.y * t->w + p.x - 1] = false;
  }
  if (is_within(t, move_point(p, 0, 1))) {
      t->cells[(p.y + 1) * t->w + p.x + 1] = false;
  }
  if (is_within(t, move_point(p, 0, -1))) {
      t->cells[(p.y - 1) * t->w + p.x] = false;
  }
  if (is_within(t, move_point(p, 1, 1))) {
      t->cells[(p.y + 1) * t->w + p.x + 1] = false;
  }
  if (is_within(t, move_point(p, -1, 1))) {
      t->cells[(p.y + 1) * t->w + p.x - 1] = false;
  }
  if (is_within(t, move_point(p, 1, -1))) {
```

```
      t->cells[(p.y - 1) * t->w + p.x + 1] = false;
  }
  if (is_within(t, move_point(p, -1, -1))) {
      t->cells[(p.y - 1) * t->w + p.x - 1] = false;
  }
}
```

Ma anche quegli assegnamenti e quei *subscript* sono ripetizioni che possiamo rimpiazzare con chiamate a funzione:

```
bool is_within(table_t* t, point_t p) {
  return p.x >= 0 && p.x < t->w && p.y >= 0 && p.y < t->h;
}

void clear_at(table_t* t, point_t p) {
  t->cells[p.y * t->w + p.x] = false;
}

void clear_around(table_t* t, point_t p) {
  if (is_within(t, p)) {
      clear_at(t, p);
  }
  if (is_within(t, move_point(p, 1, 0))) {
      clear_at(t, move_point(p, 1, 0));  /* la move_point() viene chiamata 2 volte per ogni if: anche ques
  }
  if (is_within(t, move_point(p, -1, 0))) {
      clear_at(t, move_point(p, -1, 0));
  }
  if (is_within(t, move_point(p, 0, 1))) {
      clear_at(t, move_point(p, 0, 1));
  }
  if (is_within(t, move_point(p, 0, -1))) {
      clear_at(t, move_point(p, 0, -1));
  }
  if (is_within(t, move_point(p, 1, 1))) {
      clear_at(t, move_point(p, 1, 1));
  }
  if (is_within(t, move_point(p, -1, 1))) {
      clear_at(t, move_point(p, -1, 1));
  }
  if (is_within(t, move_point(p, 1, -1))) {
      clear_at(t, move_point(p, 1, 1));
  }
  if (is_within(t, move_point(p, -1, -1))) {
      clear_at(t, move_point(p, -1, -1));
  }
}
```

La nostra riscrittura ha tolto delle ripetizioni ma ne ha prodotte di nuove: la `move_point()` viene invocata 2 volte per ogni `if` con gli stessi argomenti. Eliminiamole:

```
bool is_within(table_t* t, point_t p) {
  return p.x >= 0 && p.x < t->w && p.y >= 0 && p.y < t->h;
}

void clear_at(table_t* t, point_t p) {
  t->cells[p.y * t->w + p.x] = false;
}
```

```
void safe_clear_at(table_t* t, point_t p) {
  if (is_within(t, p))
    clear_at(t, p, b);
}

void clear_around(table_t* t, point_t p) {
  safe_clear_at(t, p);
  safe_clear_at(t, move_point(p, 1, 0));
  safe_clear_at(t, move_point(p, -1, 0));
  safe_clear_at(t, move_point(p, 0, 1));
  safe_clear_at(t, move_point(p, 0, -1));
  safe_clear_at(t, move_point(p, 1, 1));
  safe_clear_at(t, move_point(p, -1, 1));
  safe_clear_at(t, move_point(p, 1, -1));
  safe_clear_at(t, move_point(p, -1, -1));
}
```

Un'ultima ripetizione è rimasta: le 9 chiamate esplicite per tutte le 9 coordinate. Possiamo eliminare persino queste:

```
bool is_within(table_t* t, point_t p) {
  return p.x >= 0 && p.x < t->w && p.y >= 0 && p.y < t->h;
}

void clear_at(table_t* t, point_t p) {
  t->cells[p.y * t->w + p.x] = false;
}

void safe_clear_at(table_t* t, point_t p) {
  if (is_within(t, p))
    clear_at(t, p, b);
}

void clear_around(table_t* t, point_t p) {
  size_t dx, dy;
  for (dy = -1; dy < 1; ++dy)
    for (dx = -1; dx < 1; ++dx)
      safe_clear_at(t, move_point(p, dx, dy));
}
```

Confrontate ora il primo sorgente con quest'ultimo: il *refactoring* di un programma è un lavoro di riscrittura e raffinamento del codice che gradualmente ne migliora la qualità, togliendo ripetizioni e riducendo i possibili errori.

4.3 Funzioni Brevi

Anche quando siamo in presenza di codice non ripetuto, possiamo ugualmente *spezzare* un blocco lungo in funzioni *brevi*, dal nome chiaro e con argomenti che si commentano da soli: anche se le chiamate una volta sola, va bene lo stesso. E' più facile leggere un blocco che chiama alcune funzioni dal nome esplicativo piuttosto che un blocco lungo.

4.4 Evitare Copie Locali di Campi

E non è tutto - prendete questo esempio:

```
int main() {
  struct my_array a = create_my_array(10);

  double* oldpt = a.pt;  /* tenere una copia locale del pointer è SBAGLIATO */

  a = append(a);  /* dopo l'assegnamento a.pt non è più quello di prima */
```

```
  printf("%f\n", oldpt[0]);  /* la copia del vecchio pointer non punta a memoria valida */

  free_my_array(a);
  return 0;
}
```

Conservare in una variabile locale una copia di un campo della `struct` (ad esempio il pointer pt) è una pratica *error-prone* perché quel campo è soggetto a cambiamento ed il puntatore non punta più alla vecchia memoria.

4.5 Dove Liberare la Memoria

Un altro grande dilemma è *quando* liberare la memoria: la regola generale più sicura è farlo *simmetricamente* all'operazione di allocazione. Pensate all'allocazione di memoria come ad una operazione di *preambolo* e alla liberazione come ad una operazione di *epilogo*. Un programma (o un sotto-programma) può dunque essere suddiviso in 3 *momenti*:

```
PREAMBOLO: allocare la memoria

CORPO: usare la memoria

EPILOGO: liberare la memoria
```

Non sempre questo è possibile, però si cerchi di progettare il codice in modo da rispettare questo pattern. Quando si è nel dubbio si provi a spostare *in fuori* l'allocazione, cioè spostarla nello scope del chiamante; o, più in generale, si cerchi di spostare il preambolo nello stesso scope dove siete sicuri di poter mettere l'epilogo.