ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

PROGRAMMAZIONE OBJECT-ORIENTED

COSIMO LANEVE
cosimo.laneve@unibo.it

CORSO 00819 – PROGRAMMAZIONE

ARGOMENTI [SAVITCH, CAPITOLO 10]

1. la programmazione object-oriented e le classi
2. campi e metodi
3. membri private e public
4. costruttori
5. esempi/esercizi su classi con liste, pile, code

CHE COSA È UNA CLASSE

una *classe* è un tipo di dato che specifica come i suoi elementi, detti *oggetti*, possono essere creati e usati

per definire una classe occorre

- definire i ***campi*** degli oggetti (tipo e identificatore)
- definire le funzioni della classe, dette ***metodi***

una classe **estende il concetto di struttura**

- le classi racchiudono ***in un unico costrutto sintattico*** la definizione dei ***valori*** e quella delle ***operazioni*** su quei valori (nelle strutture la connessione con le operazioni non c'è)

UTILITÀ DELLE CLASSI

- in C++, **come nella maggior parte dei linguaggi di programmazione moderni**, le classi sono il mattone elementare per costruire grossi programmi
- e sono anche molto utili per piccoli programmi
- il concetto fu introdotto originariamente in `Simula67`

UN PRIMO ESEMPIO DI CLASSE

definire una classe **`rectangle`**

- che ha **due campi** che memorizzano la **`base`** e l'**`altezza`**
- ed ha i **due metodi**
  - **`area`** ritorna l'area del rettangolo relativo, ad esempio ritorna **6** se i due campi memorizzano **2** e **3**
  - **`perimeter`** ritorna il perimetro del rettangolo relativo, ad esempio ritorna **10** se i due campi memorizzano **2** e **3**

`public` significa che tutti i metodi e i campi sono visibili all'esterno

```
class rectangle{
    public:
        int base ;                          // primo campo
        int altezza ;                       // secondo campo
        int area(){                         // operazione area
              return(abs(base*altezza)) ;
        } ;
        int perimeter(){                    // operazione perimetro
        return(2*abs(base)+2*abs(altezza)) ;
        } ;
 } ;
```

CLASSE RECTANGLE: USO

con questa definizione di **rectangle**, nel **main** possiamo scrivere

```
    . . .
rectangle x , y;
x.base = 2 ; y.base = 3 ;
x.altezza = -3 ; y.altezza = 4 ;
cout << x.area() << y.perimeter() ;
    . . .
```

si accede ai metodi allo stesso modo di come si accede ai campi!

1. dichiaro gli oggetti usando il nome della classe come tipo
2. l'accesso ai campi è come per le strutture
3. l'accesso ai metodi è simile a quello dei campi

problema: in questa versione di **rectangle**, l'implementazione è nota all'esterno della classe

- **non è necessario** che l'utente della classe sappia che un rettangolo ha due campi di tipo **int** che si chiamano **base** e **altezza**
- **se si decide di cambiare** l'implementazione (ad es. il nome dei campi o il loro tipo) sarà necessario cambiare tutto il codice

ESERCIZI

1. definire una classe cerchio (ispirandosi alla classe rettangolo) con i metodi per area e circonferenza
   - dopo aver definito la classe, scrivere un programma che crea un cerchio e ne stampa l'area
2. nel definire la classe cerchio avete dovuto scegliere se memorizzare in un campo il raggio o il diametro
   - modificare la classe cerchio precedentemente definita cambiando questa scelta (usare il diametro al posto del raggio o viceversa)
   - il programma precedente funziona anche con la nuova classe cerchio? O dovete cambiare anche altre parti del programma?

ENCAPSULATION (INCAPSULAMENTO)

- nell'esercizio precedente un cambio alla struttura interna del cerchio **richiede** un cambio del programma che usa il cerchio
- **questo non va bene**
  - rende il programma **difficile da modificare**
  - non posso far sviluppare la classe e il codice che lo usa a persone diverse
  - …proprio le cose che la programmazione orientata agli oggetti deve consentire
- la ragione è che non abbiamo separato **parte pubblica** e **parte privata**
- l'**incapsulamento** consente di nascondere i dettagli implementativi di una classe

RECTANGLE RIVISTO E L'INCAPSULAMENTO

public vs protected

se omesso, tutto ciò che non è public è protetto

```
class better_rectangle{
        protected:
              int base ;                  // i due campi non sono più public
              int altezza ;               // e contengono interi unsigned
        public:
              int area(){ return(base*altezza) ;
              } ;
               int perimeter(){ return(2*base + 2*altezza) ;
              } ;
              void make(int m, int n){ base = abs(m); altezza = abs(n) ;
              } ;                         // è possibile rimuovere le invocazioni
} ;                                        // ad abs in area e perimeter!
```

con questa definizione di `rectangle`, nel `main` ***NON*** possiamo scrivere

```
. . .
better_rectangle x ;
x.base = 2 ;          // error: 'int rectangle::base' is protected
```

perché i campi **non sono più visibili all'esterno della classe**

per questo motivo è stato introdotto il metodo `make`

LE DICHIARAZIONI DELLE CLASSI

```
class Class_Name {
      protected:
           Member_Specification_1
           Member_Specification_2
           . . .
      public:
           Member_Specification_n+1
           Member_Specification_n+2
           Member_Specification_n+3
           . . .
};
```

lo specifier "protected" può essere omesso

osservazioni:

vedi metodo make in better_rectangle

- anche i metodi possono essere "`protected`"
- i campi "`protected`" richiedono la presenza di metodi che li inizializzano e li aggiornano

INCAPSULAMENTO – RIEPILOGO

- un oggetto contiene una parte **pubblica** e una parte **privata**
  - la parte pubblica è visibile da chi usa l'oggetto
  - la parte privata no
  - si dice che la parte privata è **incapsulata**

- se un programmatore modifica la parte privata lasciando inalterata quella pubblica **gli altri programmatori non se ne accorgono** perchè la ignorano

LE DICHIARAZIONI DELLE CLASSI/CONT.

in C++ è anche possibile definire i metodi all'esterno della classe (notazione "`::`")

```
class another_rectangle{
    protected:
        int base ;
        int altezza ;
    public:
        int area() ;
        int perimeter() ;
        void make(int, int) ;
} ;
int another_rectangle::area(){ return(base*altezza) ; } ;
int another_rectangle::perimeter(){return(2*base + 2*altezza); } ;
void another_rectangle::make(int x, int y){
                    base= abs(x); altezza= abs(y);
} ;
```

problemi: in questo modo si perdono i vantaggi del concetto di classe, cioè avere in un **unico costrutto/parte del codice** sia i campi che le operazioni sui campi

COSA DEFINIRE PUBLIC E COSA PROTECTED

- dall'esterno è possibile usare solo campi e metodi public
- è possibile modificare campi e metodi protected **senza dover modificare** i programmi che usano la classe
- **desiderata**: avere **meno campi e metodi public possibile** (information hiding)
- di solito sono protected tutte le informazioni legate all'implementazione
  - tutti i campi
  - tutti i metodi ausiliari
- quando questo accade si parla di **abstract data type**

I COSTRUTTORI

quando si crea un oggetto di una classe occorre inizializzarne i campi

- conviene creare un metodo **`public`** che li inizializza (così non ci si dimentica di qualche campo)
- tale metodo di inizializzazione **è necessario** sui campi **`protected`**
- per facilitare l'inizializzazione, **`C++`** da la possibilità di definire un metodo il cui identificatore è lo stesso di quello della classe: il **costruttore**

```
class best_rectangle{
    protected: int base ;
               int altezza ;
    public:
       int area(){ return(base*altezza) ; } ;
       int perimeter(){ return(2*base + 2*altezza) ; } ;
       best_rectangle(int m = 0, int n = 0){
              base = abs(m); altezza = abs(n); } ;
} ;
```

DETTAGLI SUI COSTRUTTORI

- i costruttori **non hanno tipo di ritorno** (è implicitamente la classe corrispondente)
- nel chiamante c'è **una delle due dichiarazioni**:

```
best_rectangle x(3,-4) ;
best_rectangle x = best_rectangle(3,-4) ;
```

- i costruttori non si invocano come gli altri metodi: **non è possibile scrivere**

```
best_rectangle x ;
x.best_rectangle(3,-4) ;
// error: no matching function for call
// to 'best_rectangle::best_rectangle()'
```

- è possibile anche definire **i valori di default nel costruttore** (se nella dichiarazione dell'oggetto il costruttore non viene invocato, allora i campi sono inizializzati con i valori di default):

```
best_rectangle(int m=0, int n=0){ base= abs(m); altezza= abs(n); };
```

ESERCIZI

1. definire una classe **`frazione`** con un opportuno costruttore ed i metodi **`stampa`**, **`moltiplica`** e **`inverso`**
   - il metodo **`moltiplica`** prende come parametri due frazioni e mette il risultato nella frazione su cui è invocato
   - il metodo **`inverso`** inverte la frazione su cui è invocato

PUNTATORI A OGGETTI E ACCESSO AGLI ELMENTI `PUBLIC`

è possibile definire **puntatori a oggetti**, quindi per accedere agli elementi (campi e metodi **`public`**) si usano gli stessi meccanismi delle strutture

```
int main(){
        best_rectangle a = best_rectangle(3,5) ;
        best_rectangle *b, *c;
        best_rectangle *d = new best_rectangle(2,3) ;
        b = new best_rectangle() ;
        c = &a ;
        cout << a.area() ;
        cout << b->area();
        delete b;  delete c;
        return 0;
}
```

ATTENZIONE: un oggetto non può mai essere NULL

oggetto ⇜⇝ struttura

puntatore ⇜⇝ NULL

CLASSI/IL TIPO DI DATO ASTRATTO PILA

```
class stack {

    struct pila { int val ; pila *next ; } ;
    typedef pila *p_pila ;

    protected:
        p_pila elem ;
    public:
        stack(){ elem = NULL ; } ;
        bool is_empty(){
            return(elem == NULL) ;
        } ;
        void push(int e){
            p_pila x  = new(pila) ;
            x->val = e ; x->next = elem ;
            elem = x ;
        } ;
        void pop(){
            p_pila x ;
            if (elem != NULL){
                x = elem ;
                elem = elem->next ;
                delete(x) ;
            } ;
        } ;
        int top(){
            if (elem != NULL) return(elem->val) ;
        } ;
} ;
```

CLASSI/IL TIPO DI DATO ASTRATTO PILA

```
class stack {

     typedef stack *p_stack ;

     protected:
          int val ;
          p_stack next ;
     public:
          stack(){ val = 0 ; next = NULL ;
          } ;
          bool is_empty(){ return(next == NULL) ;
          } ;
          void push(int e){
               p_stack tmp  = new stack() ;
               tmp->val = e ;
               tmp->next = next ;
               next = tmp ;
          } ;
          void pop(){
               if (next != NULL){
                    p_stack tmp = next ;
                    next = next->next ;
                    delete(tmp) ;
               } ;
          } ;
          int top(){
               if (next != NULL) return(next->val) ;
          } ;
} ;
```

una definizione alternativa!

- usare puntatori a **`stack!`**
- non è possibile usare
  `stack next`
  perchè le strutture non possono essere ricorsive!

CLASSI/IL TIPO DI DATO ASTRATTO PILA CON GLI ARRAY

```
const int length = 1000 ;
class stackwitharray {
      protected:
            int pila[length] ;
            int num_el ;
      public:
            stackwitharray(){ num_el = 0 ; } ;
            bool is_empty(){ return(num_el == 0) ;
            } ;
            void push(int e){
                  if (num_el == length) cout << "OUT OF MEMORY \n" ;
                  else {   pila[num_el] = e ;
                           num_el = num_el + 1; } ;
            } ;
            void pop(){
                  if (num_el != 0) num_el = num_el - 1 ;
            } ;
            int top(){
                  if (num_el != 0) return(pila[num_el - 1]) ;
            } ;
} ;
```

- l'incapsulamento consente di modificare l'implementazione di un tipo di dato (classe) senza dover modificare i programmi che lo usano
- per una classe pila posso passare da implementazione con puntatori a implementazione con array

CLASSI/`THIS`

quando si scrive il codice di un metodo, la variabile predefinita `this` contiene un puntatore all'oggetto corrente

- ogni campo può essere referenziato con uno dei due modi equivalenti

```
elem = 10 ;   oppure   this->elem = 10 ;
```

`this` è utile quando

- si deve passare l'**oggetto corrente** come argomento di una funzione/metodo
- quando **si deve tornare come valore** un puntatore all'oggetto corrente

`this` è una **costante**: non si può assegnare

CLASSI/ABSTRACT DATA TYPE CONTEST

1. implementare il **tipo di dato coda** con le classi non facendo alcuna assunzione sulla dimensione delle code
2. implementare il **tipo di dato insieme** di interi con le classi