

Programmazione orientata agli oggetti (OOP) in C++

Emanuele Ing. Benatti

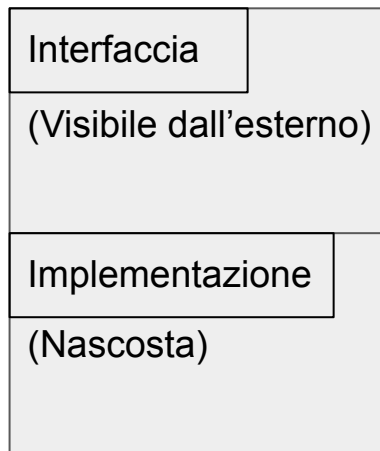
La programmazione modulare

```
17 string input;  
18 int ilength, iN;  
19 double dblTemp;  
20 bool again = true;  
21  
22 while (again) {  
23     iN = -1;  
24     again = false;  
25     getline(cin, input);  
26     system("cls");  
27     stringstream(sinput) >> dblTemp;  
28     ilength = sinput.length();  
29     if (ilength < 4) {  
30         again = true;  
31         continue;  
32     } else if (sinput[ilength - 3] != '.') {  
33         again = true;  
34         continue;  
35     } while (++iN < ilength) {  
36         if (isdigit(sinput[iN])) {  
37             continue;  
38         } else if (iN == (ilength - 3)) {  
39             continue;  
40         }  
41     }  
42 }
```

Fino ad oggi ogni programma è stato visto come un insieme di moduli che interagiscono tra di loro. Tutte le funzioni di un modulo sono scritte nello stesso file.

- questa cosa era propria del linguaggio C/C++ dove esiste un sorgente in un file e poi vengono implementate le funzioni.
- Concetto di black box

MODULO



La programmazione modulare(2)

```
17 string sinput;  
18 int ilength, iN;  
19 double dblTemp;  
20 bool again = true;  
21  
22 while (again) {  
23     iN = -1;  
24     again = false;  
25     getline(cin, sinput);  
26     system("cls");  
27     stringstream(sinput) >> dblTemp;  
28     stringstream(sinput).length();  
29     ilength = sinput.length();  
30     if (ilength < 4) {  
31         again = true;  
32         continue;  
33     } else if (sinput[ilength - 3] != '.') {  
34         again = true;  
35         continue;  
36     }  
37     while (++iN < ilength) {  
38         if (isdigit(sinput[iN])) {  
39             if (isalnum(sinput[iN])) {  
40                 continue;  
41             } else if (iN == (ilength - 3)) {  
42                 continue;  
43             }  
44         }  
45     }  
46 }
```

Importante è la programmazione modulare basata su più files di codice sorgente, ognuno dei quali dedicato alla risoluzione di un determinato problema e messi insieme alla fine risolvono globalmente il problema iniziale.

In C lo sviluppo di moduli (multi-file che svolgono compiti specifici) viene gestito attraverso dei files di supporto chiamati files d'interfaccia o header, tali files (con estensione .h) contengono la definizione/dichiarazione di strutture dati e funzioni che devono essere accessibili agli altri file che includono il modulo in questione. La definizione completa di strutture dati e funzioni è invece contenuta nei file di codice sorgente (con estensione .c).

Durante la fase di compiling i files vengono compilati separatamente e successivamente linkati insieme dal linker del compilatore.

La programmazione modulare (3)

```
17 string input;  
18 int ilength, iN;  
19 double dblTemp;  
20 bool again = true;  
21  
22 while (again) {  
23     iN = -1;  
24     again = false;  
25     getline(cin, input);  
26     system("cls");  
27     stringstream(input) >> dblTemp;  
28     ilength = input.length();  
29     if (ilength < 4) {  
30         again = true;  
31         continue;  
32     } else if (input[ilength - 3] != '.') {  
33         again = true;  
34         continue;  
35     } while (++iN < ilength) {  
36         if (isdigit(input[iN])) {  
37             continue;  
38         } else if (iN == (ilength - 3)) {  
39             continue;  
40         }  
41     }  
42 }
```

- I moduli vengono creati a parte rispetto al linguaggio
- Tecnica di organizzazione del codice molto seguita dai programmatori.

Vantaggi:

- al crescere della complessità dei programmi da sviluppare, consente rendere più semplici ed efficaci le operazioni di modifica e di testing del codice prodotto.
- permette la suddivisione del lavoro;
- permette del riuso del software
- rende facile la manutenzione;

MA

nessun software “grande” è scritto SOLO con la programmazione modulare

Punti ostici

- difficoltà nell'interazione tra i funzioni (moduli)
- separazione dei moduli,
- compatibilità tra moduli software,
- dipendenza tra moduli software,
- limiti delle strutture e rapporto tra strutture e moduli (funzioni)

```
17 string sInput;  
18 int iLength, iN;  
19 double dblTemp;  
20 bool again = true;  
21  
22 while (again) {  
23     iN = -1;  
24     again = false;  
25     getline(cin, sInput);  
26     system("cls");  
27     stringstream(sInput) >> dblTemp;  
28     iLength = sInput.length();  
29     if (iLength < 4) {  
30         again = true;  
31         continue;  
32     } else if (sInput[iLength - 3] != '.') {  
33         again = true;  
34         continue;  
35     } while (++iN < iLength) {  
36         if (isdigit(sInput[iN])) {  
37             continue;  
38         } else if (iN == (iLength - 3)) {  
39             continue;  
40         }
```

Nuovo paradigma OOP

- La programmazione orientata agli oggetti (Object Oriented Programming, OOP) è un paradigma di programmazione, in cui un programma viene visto come un insieme di oggetti che interagiscono tra di loro.
- La programmazione ad oggetti rappresenta un ulteriore sviluppo rispetto alla programmazione modulare.
- Nei linguaggi OOP esiste un nuovo tipo di dato, la classe. Questo tipo di dato serve appunto a modellare un insieme di oggetti dello stesso tipo.

In generale, un oggetto è caratterizzato da un insieme di attributi (**dati**) e da un insieme di funzionalità (**metodi**)

Esempio di definizione di classe

Pensiamo ad astrarre un'automobile, individuiamo attributi e funzionalità

attributi

potenza marca modello marce stato velocità	accelera frena cambio cambio direzione
---	--

funzionalità



È possibile interagire con l'automobile, per modificare il suo comportamento attraverso la sua interfaccia che definisce le operazioni consentite.

Classi e oggetti

OOP cioè Object oriented programming è un paradigma di programmazione dove è possibile definire dei nuovi tipi di dati, dette classi che rappresentano qualsiasi cosa, reale o astratta, nella quale si possono immagazzinare *dati* ed *operazioni*.

Le variabili che vengono definite partendo dalla definizione classe vengono chiamate oggetti (o istanze della classe)

Dati è una classe.

Un possibile oggetto in un programma potrebbe essere

```
Dati variabile=new Dati();
```

attributi



metodi



ADT

Le classi sono esempi di Abstract Data Type.

La classe è il mezzo naturale per tradurre l'astrazione di un tipo definito dall'utente che combina la rappresentazione dei dati (attributi) con le funzioni (metodi) che manipolano i dati.

La collocazione di dati e funzioni in una sola entità, la classe, è l'idea centrale dell' OOP.

Perchè usare gli ADT (e le classi) e non le funzioni?

Tre grandi motivi:

- Spesso nei programmi è necessario rendere **modulare** il codice,
- **Riusabilità del codice:** perchè dover definire di nuovo in ogni software, determinate funzionalità già scritte per risolvere problemi simili?
- Creare classi permette di rendere facile quella che viene chiamata **astrazione** del problema. (astrazione significa semplificare/nascondere)

Esempio di astrazione dobbiamo scrivere un programma che implementi il sistema di decollo di un aereo: complesso vero? sì ma tutta la difficoltà può essere astratta in una classe dove ci saranno all'interno tutti i dettagli realizzativi, nei metodi e funzioni e tutte le variabili necessarie al funzionamento.

Definizione di classe

ha due parti:

dichiarazione: descrive i dati e l'interfaccia ("metodi")

definizioni dei metodi: descrive l'implementazione delle funzioni membro

```
class NomeClasse
{
    //dichiarazione degli attributi
    //dichiarazione dei metodi (e implementazione)
}
```

Gli attributi sono delle variabili di qualsiasi tipo base: int, char, double, puntatori, o altre classi.

I metodi sono delle funzioni definite allo stesso modo e con le stesse regole (parametri e valori di ritorno)

Definizione di classe (2)

```
class Persona {           // Classe
    private:               // Specificatore di accesso
        int myNum;         // attributi della classe
        string myString;
    public:
        void setNumero(int numero)
        {
            myNum=numero;
        }
        void setNome(string nome)
        {
            myString=nome;
        }
        void stampa()
        {
            cout<<"mi chiamo "<<myString<<" e il mio numero e'"<<myNum<<endl;
        }
};
```

```
int main()
{
    cout<<"Primo esempio di
    classe";
    Persona P;
    P.setNome("Emanuele");
    P.setNumero(3);
    P.stampa();
    return 0;
}
```

P è una **istanza** della
classe Persona !

Dichiaratori di accesso

Le parole `public` e `private` e `protected` che puoi vedere nel codice di una classe servono a limitare l'accesso ai metodi e agli attributi e sono chiamati dichiaratori di accesso.

Per default, i metodi di una classe sono nascosti all'esterno, cioè, i suoi dati ed i suoi metodi sono *private*

è possibile controllare la *visibilità* esterna mediante specificatori d'accesso:

la sezione `public` contiene membri a cui si può accedere dall'esterno della classe

la sezione `private` contiene membri ai quali si può accedere solo dall'interno della classe

ai membri che seguono lo specificatore `protected` si può accedere anche da metodi di classi *derivate* della stessa

Dichiaratori di accesso

E' buona norma:

- tutti gli attributi (variabili) definirli private,
- tutti i metodi (funzioni) per accedere agli attributi e modificare i dati sono definite public o protected.

Metodi get/set

Avendo definito privati gli attributi, è buona norma definire i così detti metodi get/set che permettono di leggere e scrivere gli attributi dall'esterno

```
class misura{          // Classe
    private:            // Specificatore di accesso
        int valore;     // attributi della classe
    public:

        void setValore(int numero)
        {
            valore=numero;
        }
        int getValore()
        {
            return valore;
        }
};
```

Metodi get/set

Avendo definito privati gli attributi, è buona norma definire i così detti metodi get/set che permettono di leggere e scrivere gli attributi dall'esterno

```
class misura{          // Classe
    private:            // Specificatore di accesso
        int valore;     // attributi della classe
    public:

        void setValore(int numero)
        {
            valore=numero;
        }
        int getValore()
        {
            return valore;
        }

};
```

L'attributo valore è modificabile solo tramite i metodi get e set. All'esterno della classe non sarà visibile in alcun modo!

Information Hiding

Definire tutti gli attributi privati, definisce il concetto di information hiding: il concetto di classe “nasconde” gli attributi all'esterno della classe stessa.

```
class misura{          // Classe
    private:            // Specificatore di accesso
        int valore;     // attributi della classe
    public:

        void setValore(int numero)
        {
            valore=numero;
        }
        int getValore()
        {
            return valore;
        }
};
```

Parola chiave this

Un metodo può avere un argomento o una variabile locale con lo stesso nome di un attributo.

In questo caso si usa la parola chiave this per distinguere le due variabili.

```
class Punto{           // Classe
    private:            // Specificatore di accesso
        int x,y;        // attributi della classe
    public:

        void setValore(int x, int y)
        {
            this->x=x;
            this->y=y;
        }
        ....
};
```

Definizione esterna dei metodi

Non è obbligatorio definire subito nella definizione della classe tutti i metodi, si può fare nello stesso file all'esterno della classe.

```
class Punto{           // Classe
    private:           // Specificatore di accesso
        int x,y;       // attributi della classe
    public:

        void setValore(int x, int y);

};

Punto::SetValore(int x, int y)
{
    this.x=x;
    this.y=y;
}
```

E' possibile scrivere all'interno della classe solo il prototipo dei metodi e specificare l'implementazione dei metodi all' esterno della classe.

Non farlo: il compilatore vi darà errore e non andrà nulla.

Costruttori e distruttori

Un costruttore concettualmente inizializza e alloca le risorse (eventualmente dinamiche) che userà l'oggetto

```
class Rettangolo{           // Classe
private:                    // Specificatore di accesso
    int base,altezza;      // attributi della classe
public:

    Rettangolo()
    {
        base=3;
        altezza=4;
    }
    Rettangolo(int b, int h)
    {
        base=b;
        altezza=h;
    }
};
```

Costruttori e distruttori

Un distruttore concettualmente libera e dealloca le risorse (eventualmente dinamiche) che userà l'oggetto

```
class Rettangolo{           // Classe
private:                     // Specificatore di accesso
    int *base,altezza;      // attributi della classe
public:

    Rettangolo()
    {
        base=(int *) malloc(sizeof(int));
        *base=3;
        altezza=4;
    }
    Rettangolo(int b, int h)
    {
        base=(int *) malloc(sizeof(int));
        *base=b;
        altezza=h;
    }
    ~Rettangolo()
    { // libera le risorse dinamiche, attenzione ai puntatori!!!!
        free(base)
    }
};
```

Domanda aperte: può esistere una classe che usa memoria dinamica e non avere un distruttore?
Può esistere un costruttore con un solo parametro in questo esempio?

Costruttori e distruttori

Il distruttore è sempre uno solo e non può avere parametri.

Il nome DEVE essere: ~NomeClasse

Il costruttore è una funzione con lo stesso nome della classe e senza tipo di ritorno. Es. NomeClasse

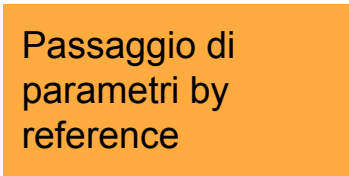
Si possono definire più costruttori (diversi per tipo e/o il numero dei parametri)

Costruttore di copia (copy constructor)

Il costruttore copia è uno speciale costruttore che serve per realizzare l'operazione di copia del contenuto informativo di un oggetto all'interno di un altro oggetto.

data una classe Punto (omesso il costruttore vuoto e con parametri)

```
class Punto{  
    public:  
        int x;  
        int y;  
    Punto(const Punto& p1)  
    {  
        x = p1.x;  
        y = p1.y;  
    }  
}
```



Passaggio di
parametri by
reference

Nel main, dati due oggetti

```
Punto a=Punto(1,2),b;
```

```
a=b;
```