

Analisi e Progettazione del Software

Classi

```
1  class Complesso
2  {
3      double re, im;    //campi
4      double Modulo(){return sqrt(re*re + im*im);}    //metodo
5      Complesso(double r, double i){re=r; im=i;}    //costruttori
6      Complesso(){re=0; im=0;}    //dichiaro un costruttore di default
7      double Fase();    //posso dichiarare prototipo qui
8  };
9  Complesso C1(2.4, 1.2);    //dichiaro oggetti e li inizializzo con
    costruttori
10
11  Complesso::Fase()    //e poi dichiaro il corpo fuori dalla classe
12  {
13      ...
14  }
```

Trucchetto : tutte le librerie standard del c sono accessibili in c++
es: #include <math.h> diventa #include <cmath>

Privatizzazione nelle classi

Ci sono 3 possibili livelli di privatizzazione

- public: il valore è accessibile a tutti, in generale ci mettiamo i metodi
- private: il valore non è accessibile all'esterno, in genere ci mettiamo i dati
- protected: noi non lo useremo

Trucchetto: è possibile rendere pubblici gli oggetti creando un metodo che restituisce il valore (*selettori*), così siamo protetti in scrittura ma non in lettura

Costanti nelle classi

Posso definire degli oggetti come costanti es:

```
1  const Data d(g,m,a);
```

Per poter utilizzare questi oggetti abbiamo bisogno di garantire al compilatore che se la variabile viene passata a una funzione non verrà modificata. Esempio selettore:

```
1 unsigned Giorno() const {return giorno;} //devo aggiungere const
```

All'interno di una classe posso avere un campo costante (in genere pubblico) che posso inizializzare direttamente nella dichiarazione della classe oppure inizializzarlo in un costruttore:

```
1 Data::Data(unsigned g, unsigned m, unsigned a)
2     : GIORNI(365) //riga di inizializzazione
3     {...}
```

⇒ quando chiamo il costruttore per un oggetto me lo inizializza

Passaggio dei valori alle funzioni

In c esiste solo il passaggio per valore, mentre in c++ esiste anche il passaggio per *riferimento*.

Passaggio per riferimento

Passo alla funzione il puntatore alla variabile.

Problema: vogliamo passare un dato grosso (inefficiente copiare) ma vogliamo anche che il dato non venga modificato ⇒ passaggio per *riferimento costante* (molto usato)

```
1 int F(const A& a){...}
```

⇒ ovviamente dovremo utilizzare metodi che non modificano il dato (*const*)

Funzione che restituisce un puntatore

Posso creare un metodo che mi restituisce per riferimento:

```
1 int& Max(int v[], unsigned n) //trova val max in un vettore
2 {
3     unsigned i, m = 0;
4     for(i = 1; i < m; i++)
5         if(v[i] > v[m])
6             m = i;
7     return v[m]; //si dice che restituisco un L-value
8 }
```

Overloading degli operatori

Gli operatori convenzionali perdono senso quando li andiamo ad utilizzare su degli oggetti creati dai noi. Dato che questo semplificherebbe l'utilizzo delle classi in c++ esiste un metodo per farlo: ⇒ aggiungiamo significato agli operatori (*overloading degli operatori*)

Esempio overloading

Scriviamo l'overloading per l'operatore `+=` per una data (somma giorni) invece di utilizzare la funzione scritta in precedenza

```
1 void operator+=(int n)
2 {
3     if(n > 0)
4         for(int i = 0; i < n; i++)
5             ++(*this);    //++ ha precedenza su * => devo mettere
                             parentesi
6     ...
7 }
```

Alla riga 5 stiamo utilizzando l'operatore di incremento prefisso che avevamo già dichiarato, lo stiamo utilizzando sull'elemento *this* che indica l'oggetto chiamante della funzione che andiamo a passare per riferimento alla funzione ⇒ in questo modo non devo restituire niente

Operatore freccia (→):

Viene utilizzato per brevità quando vogliamo utilizzare un membro di un puntatore

```
1 (*puntatore).membro == puntatore->membro
```

puntatore deve essere ovviamente un puntatore a un oggetto di una classe. Se ad esempio volessi accedere a un membro dell'oggetto chiamante:

```
1 this->giorno = 1;
```

esempio: overloading operatore `+=` per la classe *Complesso*

```
1 void Complesso::operator+=(const Complesso& c) const    //c non si
    modifica
2 {
3     return Complesso(re + c.re, im + c.im);
4 }
```

⇒ utilizzo il costruttore per ritornare un tipo Complesso

Operatore di confronto

Problema: dato che stiamo confrontando due oggetti di una stessa classe vogliamo che il primo e il secondo oggetto vengano trattati allo stesso modo:
⇒ per ovviare a questo dobbiamo creare una *funzione esterna* alla classe

esempio: sovraccarichiamo l'operatore per vedere se due date sono uguali

```
1  class Data
2  {
3      friend bool operator==(const Data& d1, const Data& d2);
4      ...
5  }
6
7  bool operator==(const Data& d1, const Data& d2)
8  {
9      return d1.giorno == d2.giorno &&
10             d1.mese == d2.mese &&
11             d1.anno == d2.anno;
12 }
```

Riga 3: dato che sto dichiarando una funzione non potrei utilizzare gli oggetti privati della classe, quindi devo dichiarare la funzione come *friend* all'interno della classe.

Perché facciamo questo?

esempio:

```
1  Complesso operator+(const Complesso& c1, const Complesso& c2) {...}
2  ...
3  c1 = c2 + 3.4;    //il compilatore esegue operator+(c2, Complesso(3.4))
4  c1 = 3.4 + c2;
```

Riga 4: se non avessi dichiarato la funzione come esterna non avrei potuto utilizzarla in questo caso: un numero, non classe Complesso, non può essere il chiamante di un metodo di classe

Operatore di output

Esempio: vogliamo stampare una data senza dover fare una sequenza di cout

```

1  friend ostream& operator<<(ostream& os, const Data& d);
2  ...
3  ostream& operator<<(ostream& os, const Data& d)
4  {
5      os << d.giorno << '/' << d.mese << '/' << d.anno;
6      return os;
7  }

```

Devo **restituire os** per poter concatenare cout (es: cout << num << "ciao")

Operatore di input

```

1  istream& operator>>(istream& is, Data& d)
2  {
3      char ch;  //legge senza salvare il carattere di separazione
4      is >> d.giorno >> ch >> d.mese >> ch >> d.anno;
5      return is;
6  }

```

⇒ è buona norma avere operatore di input e output che usano lo stesso formato:
(dd/mm/aa)

Pile (LIFO)

Andremo a gestire questa struttura dati attraverso una *classe Pila*
Primitive che andremo a definire:

- **Push:** inserisce un elemento in cima alla pila
- **Pop:** elimina il primo elemento dalla pila
- **Top:** restituisce il primo elemento della pila
- **IsEmpty:** mi comunica se è vuoto o no

⇒ per come andremo a definire le pile non ci sarà possibile saper quanti dati ci sono all'interno

Le *primitive* verranno dichiarate come metodi pubblici mentre la pila vera e propria sarà privata.

```
1  class Pila
2  {
3      friend ostream& operator<<(ostream& os, const Pila& p);
4      friend istream& operator>>(istream& is, Pila& p);
5
6      public:
7          Pila();
8          void Push(int elem);
9          void Pop(){top--;}
10         int Top() const {return vet[top]};
11         bool IsEmpty const {return top == -1};
12
13         private:
14         int* vet;
15         int dim;    //dimensione del vettore
16         int top;    //posizione dell'elemento affiorante
17     };
```

Dato che (in teoria), la pila può essere vista come infinita andremo ad utilizzare un *vettore dinamico*. Iniziamo allocando un vettore di dimensione arbitraria di 100 elementi.
⇒ quando avremo bisogno di nuovo spazio andremo a raddoppiarne la dimensione
Questo lo facciamo perché *riallocare memoria è costoso* ⇒ faremo nella Push()

```
1  Pila:Pila()
2  {
3      dim = 100;
4      vet = new int[dim];
5      top = -1;
6  }
7
8  void Pila::Push(int elem)
9  {
10     if(top == dim - 1)    //se top è 99 allora il vettore è pieno
11     {
12         int* aux_vet;
13         aux_vet = new int[dim*2];
14         for(int i = 0; i <= top; i++)    //trasferisco gli elementi
15             aux_vet[i] = vet[i];
16         delete[] vet;    //cancello il vecchio vettore
17         vet = aux_vet;    //aggiorno il puntatore al nuovo vettore
18         dim *= 2;
19     }
20     top++;
21     vet[top] = element;
22 }
23
24 ostream& operator<<(ostream& os, const Pila& p)
25 {
26     os << '(';
27     for(int i = 0; i < p.top; i++)
28         os << p.vet[i] << ", ";
29     if(p.top != -1)    //in questo modo non vado a stampare uno spazio
30         os << p.vet[top];
31     os << ')';
32     return os;
33 }
34
35 istream& operator>>(istream& is, Pila& p)
36 {
37     p.top = -1;    //anche se la pila è piena, gli impongo di essere
38     //vuota
39     char ch; int elem;
40     is >> ch;    //leggo la prima parentesi
```

```
40     ch = is.peek();    //funzione di libreria che guarda l'elemento
41     while(ch != ')')
42     {
43         is >> elem >> ch;
44         p.Push(elem);
45     }
46     return is;
47 }
```