Analisi e Progettazione del Software

Classi

```
class Complesso
{
    double re, im;   //campi
    double Modulo(){return sqrt(re*re + im*im);}   //metodo
    Complesso(double r, double i){re=r; im=i;}   //costruttori
    Complesso(){re=0; im=0;}   //dichiaro un costruttore di default
    double Fase();   //posso dichiarare prototipo qui
};
Complesso C1(2.4, 1.2);  //dichiaro oggetti e li inizializzo con costruttori

Complesso::Fase()   //e poi dichiaro il corpo fuori dalla classe
{
    ...
}
```

> **Trucchetto** : tutte la 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:

```
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:

```
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:

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

⇒ 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)

```
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:

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

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: ⇒ *overloading degli operatori*

Esempio overloading

Scriviamo l'overloading per l'operatore *+=* per una data (somma giorni)

```
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

```
friend ostream& operator<<(ostream& os, const Data& d);
...
ostream& operator<<(ostream& os, const Data& d)
{
    os << d.giorno << '/' << d.mese << '/' << d.anno;
    return os;
}
```

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

Operatore di input

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

⇒ 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.

Pila.hpp

```
class Pila
{
    friend ostream& operator<<(ostream& os, const Pila& p);
    friend istream& operator>>(istream& is, Pila& p);

    public:
    Pila();
    void Push(int elem);
    void Pop(){top--;}
    int Top() const {return vet[top]};
    bool IsEmpty const {return top == -1};

    private:
    int* vet;
    int dim;   //dimensione del vettore
    int top;   //posizione dell'elemento affiorante
};
```

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()

Pile.cpp

```cpp
Pila:Pila()
{
    dim = 100;
    vet = new int[dim];
    top = -1;
}

void Pila::Push(int elem)
{
    if(top == dim - 1)   //se top è 99 allora il vettore è pieno
    {
        int* aux_vet;
        aux_vet = new int[dim*2];
        for(int i = 0; i <= top; i++)   //trasferisco gli elementi
            aux_vet[i] = vet[i];
        delete[] vet;    //cancello il vecchio vettore
        vet = aux_vet;   //aggiorno il puntatore al nuovo vettore
        dim *= 2;
    }
    top++;
    vet[top] = element;
}

ostream& operator<<(ostream& os, const Pila& p)
{
    os << '(';
    for(int i = 0; i < p.top; i++)
        os << p.vet[i] << ", ";
    if(p.top != -1)   //in questo modo non vado a stampare uno spazio in +
        os << p.vet[top];
    os << ')';
    return os;
}

istream& operator>>(istream& is, Pila& p)
{
    p.top = -1;   //anche se la pila è piena, gli impongo di essere vuota
    char ch; int elem;
    is >> ch;     //leggo la prima parentesi
```

```
    ch = is.peek();   //funzione di libreria che guarda l'elemento
    while(ch != ')')
    {
        is >> elem >> ch;
        p.Push(elem);
    }
    return is;
}
```

Copia di due variabili di tipo pila

Se faccio:

```
Pila p1, p2;
p2 = p1;
```

All'interno della pila p2 avremo lo stesso puntatore ⇒ vogliamo creare una copia vere e propria

```
Pila& Pila::operator=(const Pila& p)
{
    //se l'indice del top è maggiore di dim devo riallocare
    if(p.top >= dim)
    {
        delete[] vet;
        dim = p.dim;
        vet = new int[dim];
    }
    top = p.top;
    for(int i = 0; i < top; i++)
    vet[i] = p.vet[i];
    return *this;
}
```

Costruttore di copia

```
Pila p2(p1);   //vogliamo che il costruttore copi la pila
```

⇒ se voglio scrivere una cosa del genere devo garantire che la copia non avvenga in maniera superficiale (caso precedente) ⇒ costruttore di copia (buona norma con strutture dinamiche)

```cpp
Pila::Pila(const Pila& p)
{
    //in questo caso non devo chiedermi se riallocare, la pila non esiste
    dim = p.dim;
    top = p.top;
    vet = new int[dim];
    for(int i = 0; i < dim; i++)
    vet[i] = p.vet[i];
}
```

Distruttore

Ogni volta che creo una pila della memoria viene allocata, devo andare a creare un metodo che me la liberi ⇒ *distruttore* (viene chiamato in automatico alla fine della funzione)

hpp

```cpp
~Pila();
```

cpp

```cpp
Pila::~Pila()
{
    delete[] vet;   //distruggo solo la parte dinamica
}
```

Funzioni di libreria

Classe String

```cpp
String s, t;
```

Sono definite di libreria i metodi e le funzioni:

```cpp
if(s == t);          s = t;
s += t;              if(s < t);
String s("ciao");    s = "ciao";
s += 'A';            a = s.size();
char ch = s[n];      s[n] = ch;
```

Lettura e scrittura di un file

Vado a dichiarare un oggetto di classe *ofstream*

```
os.open("pippo.txt");   //può essere un percorso o una stringa
os << p;   //scrivo su un file
os.close;  //chiudo file

int main(int argc, char* argv[])   //d'ora in poi andremo ad usarli
```