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*

```
#include <fstream>

int main(int argc, char* argv [])
{
    ifstream is;   //lettura di un file
    ofstream os;   //scrittura su un file

    is.open(argv[1]);
    is >> x;
    ...
    is.close()
    os.open(argv[1]);
    os << x;
    ...
}
```

Composizione tra classi

Utilizzo in una classe di oggetti appartenenti ad altre classi:

```
class A
{...};
class B
{A a;};
...
B b1, b2;
b2 = b1;
```

Quando faccio la copia di due elementi vado ad utilizzare l'operatore uguale definito per la classe B e quando viene copiato l'oggetto appartenente alla classe A utilizza l'operatore per A.

> **N.B.**
> Se ho una classe statica che utilizza un oggetto dinamico non devo peroccuparmi

Invocazione alternativa dei costruttori

Facendo riferimento all'esempio delle due classi precedenti si può voler fare:

```
B:B(int k, int j)
    :a(j)   //notazione già vista per la creazione di costanti
{...}
```

⇒ in questo modo vado a specificare che per la creazione dell'oggetto appartenente ad A devo utilizzare il costruttore con un elemento altrimenti utilizzerebbe quello vuoto di default

Creare campi vuoti

Per vari motivi posso volere che un campo della classi resti vuoto (es:classe impegno)

```
class Impegno
{
    ...
    private:
        string nome;
        Data inizio;
        Data* p_fine;   //puntatore di tipo data che posso lasciare vuoto
};
Impegno::Impegno(string n, Data d1, Data d2)
    :nome(n), inizio(d1)
{
    if(d2 == Data())   //se d2 è uguale a 01/01/1970
        p_fine = nullptr
    else
        p_fine = new Data(d2);   //puntatore di tipo Data
}
```

Tipi di allocazione dinamica

```
p = new int[n];      ->     delete[] p;
p = new Data();      ->     delete p;
p = new Data[n];     ->     delete[] p;
```

1. Allocazione di un vettore di 10 elementi interi (nel delete non servirebbe la [ ])
2. Allocazione di un puntatore di tipo Data
3. Allocazione di un vettore di oggetti Data
   ⇒ in questo caso [ ] serve per distruggere tutti gli elementi dell oggetto

> **Trucchetto**
> È possibile bloccare l'utilizzo di una funzione o metodo, ad esempio possiamo voler che nella nostra classe non sia possibile utilizzare il costruttore di copia
>
> ```
> A(const A& a) = delete;
> ```

Template

Vogliamo poter creare una classe generica che supporti l'utilizzo di campi di più tipi.
Esempio: classe Pila che accetti elementi di più tipi:

```
template <typename T>   //T sarà il nostro tipo generico
class Pila
{
    ...
    Pila(const Pila<T>& p);
    ...
    T Top() const{return vet[top];}
    ...
    private:
        T* vet;
        int dim;      // dimensione del vettore
        int top;      // elemento affiorante
};
template <typedef T>
Pila <T>::Pila()      //costruttore
{
    dim = 100;
    vet = new T[dim];
    top = -1;
}
int main()
{
    Pila<int> p1;
    Pila<double> p2;
    Pila<Data> p3;
    Pila<Pila<Data>> p4;   //posso utilizzare qualunque tipo
}
```

Attenzione
Quanto vado ad utilizzare un tipo che è una classe (es. rig 25) sto dando per scontato che la classe contenga tutti i metodi che mi servono
⇒ es. operatori di input e output

Utilizzo dei template della libreria standard

```
1  x = min(a, b);
```

min restituisce il minimo tra a e b che possono essere di qualsiasi tipo, l'importante è che sia stato *definito l'operatore <* ⇒ andremo a definirlo quasi sempre

Classe vector

```
1  #include <vector>   //utilizzeremo come statico ma dinamico con
   template
2
3  //Costruttori
4  vector<int> v;         //vuoto, vettore con dimensione zero
5  vector<int> v(10);     //10 elementi di valore 0 o utilizza costruttre
   vuoto
6  vector<int> v(20, 4); //20 elementi di valore 4
7  vector<int> v{3,4,5}; //costruttore con lista
```

Attenzione!
Ogni costruttore va comunque ad allocare una certa quantità di memoria (definita da libreria std) in maniera dinamica ⇒ si utilizza il metodo reserve() per decidere quanto

```
1  v.reserve(1000);
```

Questo viene fatto specialmente quando voglio essere sicuro che non vengano effettuate riallocazioni all'interno di cicli (rallenta esecuzione ⇒ *riduzione performance*)

Operatori e metodi

```
a = v[n];       //non esegue controllo di posizione (possibile segm. fault)
a = v.at(n);    //esegue controllo di posizione (lancia eccezione)
v.size();       //restituisce dimensione (unsigned)
v.push_back(10);//aggiungo 10 in fondo al vettore
//esiste anche versione push_front ma devo spostare tutti gli elementi
v.pop_back();   //elimino ultimo elemento (pop_front())
v.resize(40);   //fornisco nuova dimensione del vettore (iniz. a 0)
v.resize(40,-1);//nuova elementi inizializzati a -1
a = v.back();   //restituisce ultimo elemento
v.begin();      //restituisce primo indirizzo v.end() per l'ultimo
//inserire o togliere un elemento
v.erase(v.begin()+2);    //elimina elemento in posizione 2
v.insert(v.begin()+3,2); //inserisce 2 in posizione 3
```

Esempio utilizzo begin()/end()

```
#include <algorithm>
#include <vector>
sort(v.begin(),v.end());   //riordina il vettore
```

Matrice

⇒ sfruttiamo il fatto che la matrice è un vettore di vettore

```
vector<vector<float>> m(10, vector<float>(20,-1));  //crea m 10x20
```

Nuovi metodi classe iostream

Esempio classe persona

```
istream& operator<<(istream& is, Persona& p)
{
    is.ingnore(256, '-');   //ingora fino a -, per max 256 caratteri
    getline(is,p.nome,','); //scrivo in p.nome() fino alla ,
    is.get();               //leggiamo lo spazio
    getline()is,p.cognome); //se non specificato leggo fino a endl
    is >> p.data_nascita() >> p.citta_nascita();
    //ovviamente devo aver definito nella classi l'operatore <<
}
```

Restituire più valori

```
return make_pair(a,b);
pair<int,float>;   //classe con 2 elementi template (c'è anche tuple)
p.first;           //primo elemento
p.second;          //secondo elemento
```

Esempio ordinamento file

OrdinaFile

```
void OrdinaFile(const string& nome_file)
{
   ifstream is(nome_file);
   vector<Persona> v;
   Persona pers;
   while(is >> pers)
      v.push_back(pers);
      is.close();                //devo chiuderlo prima di scrivere
      sort(v.begin(),v.end());   //funzione di libreria algorithm
      ofstream os(nome_file);
   for(unsigned i = 0; i < v.size(); i++)
   {
      os << v[i] << endl;
      if(i != v.size())         //evita di stampare l'ultimo invio
         os << endl;
   }
   os.close();
}
```

Namespace

Aggiungo modularità al codice, specialmente alle librerie ⇒ non utilizzeremo nel dettaglio

```
namespace esempio
{
    class A
    {...};
};
esempio::A a;
```

namespace è un contenitore (lo posso chiamare più volte nello stesso codice) viene utilizzato per evitare conflitti di nome tra le librerie (es: std e gsl hanno conflitti)
⇒ se sono sicuro di non avere conflitti tra librerie posso usare:

```
using namespace esempio;
```

⇒ stessa cosa che abbiamo fatto fino ad ora con la libreria standard (*std*)

> **Buona norma**
> Quando andiamo a scrivere delle librerie per terzi si può andare a scrivere il .hpp utilizzando i namespace mentre nel .cpp utilizzo *using*.
> In questo modo, dato che gli invierò solamente il .hpp, non avrà conflitti.

Eccezioni

```
try
{
    if(...)
       throw ...;
}
catch(...){...}
```

- *try*: contiene del codice che potrebbe fallire
- *throw*: lancia l'eccezione
- *catch*: cattura l'eccezione ed esegue del codice

⇒ noi ci limiteremo a predisporre il codice alle eccezioni, quindi andremo solamente a lanciarle

> **Esempio catch**:
>
> ```
> catch(runtime_error& e)
> {
>     cerr << e.what() << endl;   //stampo errore
>     return 1;
> }
> ```

{esempio

```
ifstream is(nome_file);
if(!is)
    throw runtime_error("Non riesco ad aprire: " + nome_file);
```

⇒ se un eccezione viene lanciata ma non viene catturata stampa il messaggio del throw

Assert

```
assert(n > 0);   //se falsa do errore (parte del c)
```

- utilizziamo *throw* ad esempio per errori di input, errori non gestibili da parte dell'utente
- utilizziamo *assert* per evitare malfunzionamenti nel codice

Librerie da includere

```
#include <stdexcept>   //eccezioni
#include <cassert>     //assert
```

Nozioni in +

⇒ non richieste nei codici, viste solo per conoscenza

Iteratori

Supponiamo di avere un vettore di elementi. Solitamente per scorrerlo utilizziamo un for
⇒ alternativa più generale (funziona con ogni tipo di vettore della std, anche liste)

```
vector<int> v{1,3,7,-5};

vector<int>::iterator p;
for(p = v.begin(); p < v.end(); p++)
    cout << *p << ' ';
```

auto

```
auto x = f(y);   //x prende il tipo che restituisce x
```

⇒ poco usato perché rende il codice poco chiaro

Range loop

⇒ deriva a altri linguaggi in cui è molto utilizzato

```
for(auto e:v)
    cout << e << ' ';   //e contiene già il valore, eq a e = v[i]
```

Dizionari

Molto utili quando devo conteggiare degli elementi che non sono numeri

```
map<string,int> m;
m["pippo"] = 12;
```

pippo è detta chiave e *12* è il contenuto. *m* è gestito come un vettore dinamico

Set

Utilizzato per inserire valori, non già esistenti, in un vettore in maniera molto efficiente (log)

```
set<int> mySet;      // Dichiara un set di interi
    mySet.insert(5); // Inserisce l'elemento 5
    mySet.insert(2); // Inserisce l'elemento 2
    mySet.insert(8); // Inserisce l'elemento 8
    mySet.insert(2); // Questo non verrà inserito, perché 2 è già presente

    cout << "Elementi nel set (ordinati): ";
    for (int element : mySet)
        cout << element << " ";
    cout << std::endl;
```

Ereditarietà

Ci dà la possibilità di creare una *sottoclasse* (figlia) partendo da una *classe madre*

```
class Studente : Persona {};   //studente figlia, persona madre
```

⇒ la classe figlia eredita tutti i metodi e i campi della classe

Smart pointers

Gli **smart pointer** in C++ sono classi che gestiscono automaticamente la memoria allocata dinamicamente (sullo heap), prevenendo errori comuni come e fughe di risorse.

Lambda expressions

Posso passare una funzione come parametro ad un altra funzione
(es: applicare funzione a un vettore)

Move statements

Quando facciamo:

```
1  p = Polinomio(2.3,2) + Polinomio(-3.4,1)+Polinomio(2.5);
```

Per questa apparentemente assegnazione vado a creare un sacco di oggetti temporanei
Esistono due operatori per fare queste assegnazioni senza creare oggetti temporanei
⇒ *costruttore di spostamento* e *operatore di spostamento* (chiamati da elementi morenti)