

Lezione 1-2

COMPILAZIONE

La compilazione prevede 2 step: nel primo si produce un codice oggetto e nel secondo, detto *linking*, un file eseguibile. Nella fase di linking vengono rese disponibili tutte le funzioni create dall'utente e tutte quelle delle librerie usate.

Un *makefile* permette di compilare in maniera più comoda e di poter gestire più facilmente aggiornamenti di un solo file sorgente facente parte di un insieme di file. Questo è utile ad esempio quando un'applicazione è condivisa tra più file.

ES:

creiamo un file Welcome.cpp, scritto in c++ tramite un qualsiasi editor. Questo è il nostro file sorgente. Da questo, mediante una prima compilazione, otteniamo un file .o che è il file oggetto. Quest'ultimo file fungerà da “ingrediente” per la ricetta successiva che porterà alla generazione di un file eseguibile.

Sintassi:

Welcome: Welcome.o

```
g++ -o Welcome Welcome.o
```

Welcome.o: Welcome.cpp

```
g++ -c Welcome.cpp –std=c++11 –pedantic –Wall
```

A sinistra dei 2 punti ci sono dei target → obiettivi a cui vogliamo arrivare. A destra ci sono gli ingredienti, ovvero i prerequisiti per raggiungere il target. Sotto c’è la ricetta. Notiamo che un target può essere a sua volta un requisito (Welcome.o).

Nella fase di compilazione g++ è il comando; tutte le cose che cominciano con “-“ sono opzioni del comando g++.

- g++ è l’operazione di compilazione;
- “-o” permette di ottenere un file di output non necessariamente di nome di default “a.out” bensì del nome specificato dopo;
- “-c” è l’opzione del compilatore che permette di generare il codice oggetto (senza fare linking, quindi) di ciascun file sorgente, senza ripetere l’operazione su tutti i file;
- -std indica lo standard utilizzato; noi utilizziamo la versione 11 di c++ perché utilizzeremo delle opzioni presenti in questa versione.

- -Wall dice al compilatore di analizzare tutti i warnings possibili e quindi di visualizzare errori che possono essere riconosciuti in maniera automatica come la modalità di inizializzazione di una variabile, loop infiniti, etc.

Con **g++ --h** nel terminale si visualizzano tutte le opzioni del compilatore.

Il vantaggio di un makefile è che viene salvata per ogni ingrediente la data di modifica; quindi non viene generato necessariamente ogni volta un target tramite compilazione e quindi non c'è bisogno di compilare sempre. Tipicamente in un makefile si parte dal target finale.

Con il comando “touch” nel terminale si può modificare la data di creazione di un file.

Da MobaXTerm per selezionare una certa cartella ‘Z’ va inserito il comando *cd Z*, se Z è una sottocartella del percorso che si sta seguendo. Altrimenti bisogna andare a ritroso con “*cd ..* ” fino ad arrivare al percorso più ampio che include Z, per poi andare nuovamente verso Z. Scrivendo poi **g++ --h** nel terminale si possono visualizzare tutte le opzioni del compilatore.

Se come editor si usa Vim allora per uscire bisogna premere ESC e successivamente scrivere :*q* ,se non vogliamo salvare; in tal caso esce un messaggio di avviso che richiede la conferma dell’operazione e va scritto :*q!* ; altrimenti bisogna scrivere :*x*.

Se come editor si usa Emacs allora per uscire bisogna premere CTRL+x, CTRL+c (\rightarrow **C-x C-c**). Per salvare **C-x C-s**. Per chiedere aiuto **C-h**. Per stoppare e tornare indietro **C-z**. Per creare un nuovo file usare **C-x C-f**: verrà chiesto il nome del file. Mettendo un nome che non combacia con quello di alcun file già esistente si crea un nuovo file emacs. Per tornare indietro di una schermata premere ALT-v (simbologia \rightarrow **M-v**), per andare avanti di una schermata premere **C-v**. Per cancellare lo schermo e riscrivere il testo premere **C-l**. Per interrompere un comando premere **C-g**. Per tagliare **C-w**, per copiare **M-w**, per incollare **C-y**.

In un makefile si possono anche definire delle variabili. Ad esempio:

GCC= g++

In tal caso quando si utilizza g++ bisogna usare la sintassi:

`$(GCC)`

In questo modo stiamo dicendo di considerare il contenuto della variabile GCC e non la variabile in sé.

La stessa cosa potremmo fare definendo una variabile:

CFLAGS = -std=c++11 -pedantic -Wall

E richiamandola sempre con la notazione del dollaro nel momento opportuno.

Si possono utilizzare dei target “simbolici” che non hanno ricette ma solo prerequisiti. Questo è utile, ad esempio, quando si vogliono ottenere più file eseguibili.

ES: vogliamo ottenere Welcome, Welcome1 e Welcome2 (eseguibili), avendo a disposizione i rispettivi file .cpp. Allora definiamo a monte un target ‘all’ fatto così:

`all: Welcome Welcome1 Welcome2`

`Welcome: Welcome.o`

.....

`Welcome.o: Welcome.cpp`

.....

`Welcome1: Welcome1.o`

.....

`Welcome1.o: Welcome1.cpp`

.....

.

.

.

Etc.

All è un target simbolico poiché non ha anche una ricetta, ma ha solo i prerequisiti in modo tale che questi siano generati. Questo perché se un target non è prerequisito di qualcosa non viene compilato.

Quando nel terminale scriviamo:

make -f <Makefile>

con “-f” stiamo dicendo di interpretare il file come makefile.

Quando scriviamo un programma in C++ EVITIAMO ASSOLUTAMENTE di usare **using namespace std**

perché ci porta all’interno del programma tutte le famiglie di identificativi. Non è un errore, ma è una *bad practice*.

Bisogna allora usare:

- **using std::cout;** //se si vuole che il programma usi la famiglia di identificativi cout (della libreria standard)
- **using std::cin;** //se si vuole che il programma usi la famiglia di identificativi cin (della libreria standard)
- **using std::endl;** //se si vuole che il programma usi la famiglia di identificativi endl (della libreria standard), utilizzati alla fine di ogni comando di tipo cout (forse anche cin) per svuotare i buffer che collegano il nostro sistema con l’input/output.

E così via.

OSS:

>> è l’operatore di *estrazione*.

Esistono vari tipi definiti nella libreria standard.

INT è usato per gli interi e ci sono tantissime tipologie (Short, Long, Unsigned, etc.) ed occupa un numero di byte in memoria che dipende sia dal tipo specifico (Short etc.), sia dal sistema; quindi non lo si può dire a priori.

FLOAT, DOUBLE etc.

CHAR

STRING : per usare il tipo stringa bisogna includere la libreria <string> (string è di C++, cstring è propria di C e potremmo usarla solo che fa sì che le stringhe non siano degli oggetti ma dei vettori di caratteri). Si possono fare varie operazioni sul tipo stringa.

- string s1= “world” opp. string s2(“world”)
- string s3(3, ‘a’) → s3 contiene 3 ripetizioni del carattere ‘a’
- string s4 → s4 inizializzata a valore vuoto
- string s5(s2) → copia s2 in s5
- string s[5] → accede al 6th carattere della stringa

In realtà per accedere ad un carattere di una stringa c’è anche un altro modo, ovvero tramite s.at (5); il ‘.’ permette in generale di accedere agli elementi di un certo oggetto. Di fatto la stringa è una funzione oggetto e tramite il punto accediamo alle funzioni membro di questo oggetto.

Sulle stringhe si possono fare anche operazioni “aritmetiche”:

- s1+s2 fa la concatenazione di 2 stringhe s1 e s2
- s1=s2 copia s2 in s1
- s1+=s2 corrisponde a s1=s1+s2
- ==, !=, <, >, >=, <= sono operazioni di relazione che seguono l’ordine alfabetico

Ci sono poi tante altre operazioni:

- s.length() calcola la lunghezza della stringa s
- s.capacity() dà il numero di byte necessari per rappresentare la lunghezza della stringa s
- s.substr(1,3) restituisce una sottostringa della stringa s di 3 caratteri a partire da quello di indice 1, quindi dal secondo carattere.

Con CTRL+d si dice al programma che si è finito di inserire l’input da tastiera. In realtà si può anche fare in modo che il programma prenda l’input non dalla tastiera ma da un file esterno, ad esempio, ma anche che l’input di un programma sia l’output di un altro programma (tramite l’operatore pipe ||).

Se noi scriviamo da tastiera 2 parole separate da uno spazio, la variabile di tipo stringa da noi definita sarà automaticamente solo la prima parola.

Es:

```
#include <iostream>
#include <string>
using std::string;
using std::cin;
using std::cout;
int main() {
    string word;
    cin>> word;
    cout<<word;
    return 0;
}
```

Se noi da tastiera scriviamo Michele Ceccarelli a video sarà stampato solo Michele.
Per considerare tutta la stringa bisogna utilizzare la funzione getline().

Esercizi

LEZIONE 3

Viene detto *custom* il passaggio da una variabile di un certo tipo ad una di un altro tipo, ad esempio da float ad int. Il customing porta sicuramente una perdita di informazioni e potrebbe portare ad errori non sintattici bensì ad “errori” semanticici, che non sono veri e propri errori in realtà. Per questo si preferisce usare una lista di inizializzazione piuttosto che inizializzare una variabile con l’`=`: nella lista di inizializzazione si deve restituire un qualcosa che ha lo stesso tipo della variabile che si sta definendo ed inizializzando, e se ciò non accade viene comunicato attraverso un warning o un errore.

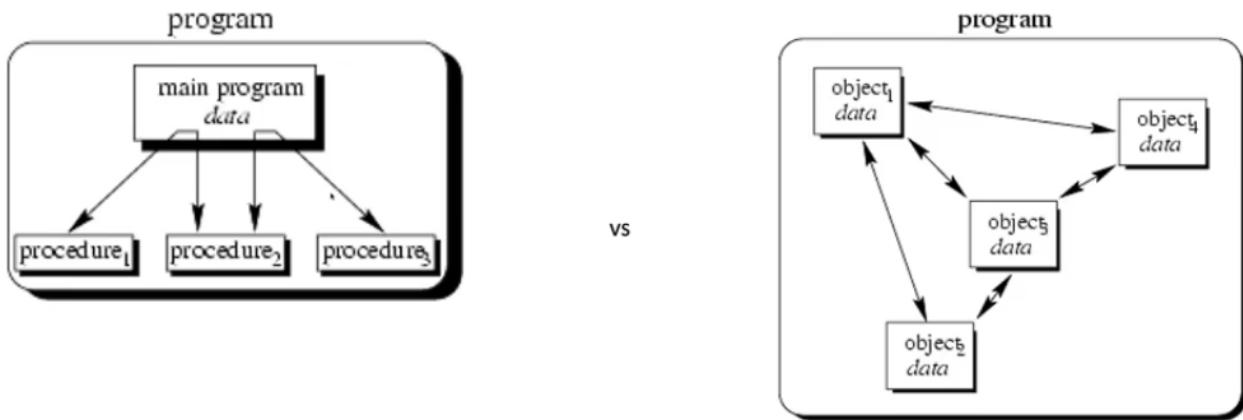
CLASSI & OGGETTI

Prima di chiederci cosa è una classe, chiediamoci cosa non è. Bene, la classe non è acqua.

In generale le classi sono i tipi a cui afferiscono gli oggetti e sono delle componenti software riutilizzabili, come le funzioni nelle librerie .h che utilizziamo. Una caratteristica della programmazione basata sulle classi e sugli oggetti è quella che gli oggetti racchiudono non solo dei dati (come vale per le funzioni, caratterizzate da parametri di input e di output) ma anche dei comportamenti. Ovvero, gli oggetti sono una rappresentazione di concetti reali che possono essere di tipo audio, video, record, etc., e ciascuno di questi oggetti può essere rappresentato in software mediante un dato che contiene sia degli attributi, che noi chiamiamo **membri** della classe e che rappresentano i valori possibili che possono essere memorizzati in quel tipo di dato, sia delle **funzioni** che possono operare sugli attributi della classe. Il vantaggio di usare una programmazione orientata agli oggetti è che un programmatore che utilizza un certo oggetto non necessariamente conosce i dettagli di come sono stati rappresentati i dati all’interno di questo oggetto ma conosce solo i comportamenti; questo semplifica lo scambio di componenti software tra diversi programmatore e quindi semplifica il processo di sviluppo. Inoltre i programmi che definiscono dei componenti sotto forma di oggetti e classi sono più facili da comprendere, correggere e modificare.

Ci sono anche altri vantaggi quali l’ereditarietà ed il polimorfismo: *ereditarietà* vuol dire che possiamo costruire oggetti a partire da altri oggetti ed alcuni oggetti possono ereditare delle proprietà e rappresentare una specializzazione di un oggetto più generale; *polimorfismo* vuol dire che lo stesso comportamento può generare procedure diverse in base al contesto.

Differenza tra programmazione procedurale e programmazione orientata agli oggetti:



The main program coordinates calls to procedures and hands over appropriate data as parameters.

Objects of the program interact by sending messages to each other

Vediamo che mentre in un linguaggio procedurale c'è il main che invoca un certo numero di procedure che possono invocare a loro volta altre procedure, in un linguaggio *object oriented* ci sono più componenti, gli oggetti, che comunicano tra di loro inviandosi messaggi per richiedere funzioni

Di fatto gli oggetti non sono altro che dei componenti che racchiudono (incapsulano) delle funzioni e dei dati; non tutto ciò che compone un oggetto è visibile dagli altri oggetti, ma solo le parti che questo oggetto espone all'esterno e quindi i messaggi che l'oggetto può ricevere sotto forma di invocazione di un certo comportamento. Quindi gli oggetti aumentano il livello di astrazione per semplificare la collaborazione durante il processo di sviluppo e cercano di diminuire l'insorgenza di errori mascherando i dettagli e cercando di descrivere le funzioni in una maniera più astratta.

Quando parliamo di classi, una classe è un tipo di dati e gli oggetti sono le istanze di quella classe e quindi variabili di quel tipo che contengono, durante il loro ciclo di vita nell'esecuzione di un processo, un particolare **stato** (stato= valori assunti dai dati contenuti in un oggetto). Quindi un oggetto incapsula sia dati che funzioni; lo stato rappresenta i dati contenuti nell'oggetto, e l'oggetto ha degli insiemi di operazioni che operano su quei dati, che prendono il nome di metodi o funzioni membro dell'oggetto e che descrivono come quelle operazioni devono essere effettuate.

Un programmatore che utilizza un oggetto, o una certa funzione, non deve sapere quella funzione come è stata realizzata, un po' come nella vita reale non c'è bisogno di conoscere com'è stata progettata la macchina per guidarla.

Le caratteristiche della programmazione orientata agli oggetti sono:

- *Incapsulamento*: definire oggetti che siano combinazione di dati e azioni. La struttura dati rappresenta le proprietà dell'oggetto nonché il suo stato; le azioni rappresentano i comportamenti possibili che sono controllati tramite funzioni.
- *Ereditarietà*: la capacità di derivare nuovi oggetti da altri vecchi di cui sono una specializzazione e di generare quindi una gerarchia tra oggetti che va dagli oggetti più astratti a quelli più specifici.
- *Polimorfismo*: la capacità di interpretare la stessa funzione in maniera diversa tra i diversi oggetti.

In C++ noi definiamo dei nuovi tipi che sono le classi e che servono a contenere un insieme di funzioni che ne implementano i comportamenti. Quindi per realizzare un comportamento di una classe bisogna scrivere una funzione specifica per quel comportamento. Di fatto una classe la possiamo immaginare come una struttura (struct) che contiene sia dati che funzioni. Le funzioni sono i messaggi esposti dalle classi.

Esempio:

vogliamo rappresentare un conto in banca tramite degli oggetti. La classe deve contenere le informazioni (dati) legate al conto corrente, quindi il nome del cliente e la somma depositata, e deve darci la possibilità di ottenere la somma depositata, fare un deposito o fare un prelievo (funzioni).

```
1 // Account.h
2 // Account class that contains a name data member
3 // and member functions to set and get its value.
4 #include <string> // enable program to use C++ string data type
5
6 class Account {
7 public:
8     // member function that sets the account name in the object
9     void setName(std::string accountName) {
10         name = accountName; // store the account name
11     }
12
13     // member function that retrieves the account name from the object
14     std::string getName() const {
15         return name; // return name's value to this function's caller
16     }
17 private:
18     std::string name; // data member containing account holder's name
19 }; // end class Account
20
```

```
1 //AccoutTest.cpp
2 // Creating and manipulating an Account object.
3 #include <iostream>
4 #include <string>
5 #include "Account.h"
6
7 using std::cout;
8 using std::cin;
9 using std::endl;
10 using std::string;
11
12 int main() {
13     Account myAccount; // create Account object myAccount
14
15     // show that the initial value of myAccount's name is the empty string
16     cout << "Initial account name is: " << myAccount.getName();
17
18     // prompt for and read name
19     cout << "\nPlease enter the account name: ";
20     string theName;
21     getline(cin, theName); // read a line of text
22     myAccount.setName(theName); // put theName in myAccount
23
24     // display the name stored in object myAccount
25     cout << "Name in object myAccount is: "
26         << myAccount.getName() << endl;
27 }
```

```
1 AccountTest: AccountTest.o
2         g++ -o AccountTest AccountTest.o
3 AccountTest.o: AccountTest.cpp Account.h
4         g++ -Wall -pedantic -std=c++11 -c AccountTest.cpp
5
```



Tipicamente le definizioni delle classi sono fatte in file .h; non è una regola, ma è buona norma.

Quando gli header file sono presenti in cartelle predefinite allora si utilizza

```
#include <file.h>;
```

quando invece sono nella stessa cartella del file che stiamo compilando allora si utilizza
#include “file.h”; .

Con l'operatore ‘.’ accedo alle funzioni membro di una certa classe. Nell'esempio di sopra si usa, per esempio, myAccount.getname() per utilizzare la funzione membro getname() definita nella classe da noi creata Account.h.

È sempre buona prassi fare iniziare per maiuscole le classi (es. class Account) e per minuscole i nomi delle variabili. Se poi un identificativo contiene più parole allora lo si fa iniziare per minuscola e come divisorì si utilizzano le prime lettere delle parole scritte in maiuscolo (es. getName). Questa convenzione è chiamata *camel case*.

Un oggetto ha degli attributi che sono implementati dai suoi dati membri; nel nostro esempio *std::string name* è il data member. Ogni oggetto ha una sua copia dei dati membri della classe. Ovviamente un oggetto contiene anche una o più funzioni membro che manipolano i dati membro appartenenti ad oggetti specifici della classe.

Ciò che contraddistingue una classe da una struct è, soprattutto, il fatto che c'è la possibilità di decidere cosa far vedere all'esterno e cosa no, ovvero la possibilità di creare una sezione pubblica ed una privata. La sezione privata è visibile solo alla classe stessa. Per default se non si specifica niente allora si sta intendendo “privato”.

Per ragioni sottili la classe, che è un header, non dovrebbe contenere direttive e quindi *using....;* per questo motivo mettiamo *std::* prima di ogni variabile di tipo string, nell'esempio. Questo perché tecnicamente non sappiamo chi importerà il file.h ma vogliamo invece che sia più riutilizzabile possibile.

LEZIONE 4

Parliamo di una funzione membro particolare che ha lo stesso nome della classe, che non ritorna nulla e che è detta **costruttore**.

Il costruttore è del tipo:

explicit NomeClasse (...) .

Explicit viene utilizzato se non c'è più di un parametro in ingresso al costruttore.

La funzione costruttore viene invocata ogni volta che si alloca un nuovo oggetto di questo tipo.

Se non definiamo nessun costruttore allora le variabili membro vengono inizializzate con i loro costruttori di default; ad ogni modo è consigliabile specificare un costruttore all'interno della classe.

Vediamo come si mette il costruttore nella classe creata per l'esercizio della scorsa lezione:

```
1 //Account.h
2 //class Account with a constructor
3 #include <string>
4 class Account {
5 private:
6     std::string name; // account name data member
7 public:
8     // constructor initializes data member name
9     // with parameter accountName
10    explicit Account(std::string accountName)
11        : name{accountName} { // member initializer
12            // empty body
13    }
14
15    // function to set the account name
16    void setName(std::string accountName) {
17        name = accountName;
18    }
19
20    // function to retrieve the account name
21    std::string getName() const {
22        return name;
23    }
24}; // end class Account
25
```

Nel nostro esempio il costruttore ha un parametro di ingresso che è di tipo stringa (accountName) e che serve per assegnare alla variabile membro 'nam il contenuto proprio del parametro 'accountNam attraverso una lista di inizializzazione.

Normalmente i costruttori sono visibili all'esterno e quindi sono pubblici. In generale la zona di visibilità di una variabile viene detta *scope*.

La parola chiave *explicit* viene utilizzata quando il costruttore ha un solo parametro in ingresso ed evita che vengano invocati dei costruttori automatici “di conversione” (poi vedremo meglio).

I costruttori non possono essere dichiarati “const” poiché devono modificare l’oggetto.

Nello specifico com’è fatta la sintassi di un costruttore? C’è un parametro in ingresso (o anche più) e poi:

: variabile_membro {parametro_in_ingresso} //lista di inizializzatori



```
10 explicit Account(std::string accountName)
11   : name{accountName} { // member initializer
12     // empty body
13   }
14 }
```

Ma sarebbe andato bene anche così:



```
10 explicit Account(std::string accountName) {
11   name = accountName;
12 }
13 }
```

Ma meglio utilizzare l’approccio della lista che funziona in ogni situazione a differenza del primo.

Se ci sono più variabili membro vanno separate le inizializzazioni tramite una virgola.

Quando richiamiamo una classe istanziandone 1 o più oggetti bisogna anche porre una lista di inizializzazione col parametro del costruttore.

Es:

```
1 //AccountTest.cpp
2 // Using the Account constructor to initialize the name data
3 // member at the time each Account object is created.
4 #include <iostream>
5 #include "Account2.h"
6
7 using std::cout;
8 using std::endl;
9
10 int main() {
11   // create two Account objects
12   Account account1{"Jane Green"};
13   Account account2{"John Blue"};
14
15   // display initial value of name for each Account
16   cout << "account1 name is: " << account1.getName() << endl;
17   cout << "account2 name is: " << account2.getName() << endl;
18 }
19
20 }
```

Notiamo poi che alle righe 17-18 c'è la stessa funzione `getName()` che assume però significati diversi perché facente riferimento a 2 oggetti diversi.

Se non definiamo esplicitamente un costruttore per una classe allora il compilatore definirà automaticamente un costruttore di default senza parametri in ingresso. Il costruttore di default non inizializza i dati membro della classe appartenenti ai tipi fondamentali (int, float, char, etc.) ma invoca il costruttore di default per ciascuna variabile membro che a sua volta è un oggetto di un'altra classe. Ad esempio nel nostro caso la variabile membro `name` è a sua volta un oggetto di tipo stringa.

Se in una classe abbiamo definito un costruttore allora non ci sarà più, per quella classe, un costruttore di default (il compilatore non lo creerà più in automatico) e quindi non possiamo dichiarare un oggetto della classe senza invocare il costruttore. Nel nostro esempio questo si traduce nel fatto che non possiamo scrivere:

`Account account1;`

poiché il costruttore richiede esattamente un parametro di tipo stringa. Invece ad esempio:

`Account account1 {"aaaaah bbbbb"};`

andrebbe bene.

Rappresentazione in UML di una classe che contiene un costruttore:



Ora concentriamoci sulle funzioni membro `set` e `get`. Queste funzioni sono utilizzate molto spesso nella creazione di una classe: tipicamente per ogni variabile membro scriviamo una funzione set ed una get; questo perché le funzioni set e get fanno da interfaccia con le variabili di tipo privato contenute nella classe e permettono quindi di accedere in maniera sicura e soprattutto controllata ai dati contenuti nell'oggetto. L'alternativa all'utilizzo delle funzioni set e get sarebbe rendere pubbliche le variabili membro, ovvero i dati, facendo sì che chiunque istanzi un oggetto di quella specifica classe possa assegnare a piacimento i valori delle variabili membro per quell'oggetto. Specialmente la funzione `set` fa da strato di interfacciamento con le variabili membro, facendo in modo da validarne i contenuti prima di poterle istanziare. Ad esempio, se cerco di istanziare un oggetto assegnando un valore privo di senso ad una variabile

membro, come una temperatura corporea negativa o il 13esimo mese dell'anno, la funzione set se ne accorge e non permette l'assegnazione.

Quindi le funzioni set e get (soprattutto la prima) servono anche per fare una validazione dei valori assegnati ad una variabile membro. Se voglio accedere alle variabili membro (che posso immaginare come il nucleo della classe) devo necessariamente passare attraverso lo strato costituito dalle funzioni get e set. Questa funzione di “controllo” riduce di molto gli errori ed aumenta la sicurezza, la robustezza e la riutilizzabilità dei programmi.

ESEMPIO dell'account bancario con saldo:

```

1 // Account.h
2 // Account class with name and balance data members, and a
3 // constructor and deposit function that each perform validation.
4 #include <string>
5
6 class Account {
7 public:
8     // Account constructor with two parameters
9     Account(std::string accountName, int initialBalance)
10    : name(accountName) { // assign accountName to data member name
11
12    // validate that the initialBalance is greater than 0; if not,
13    // data member balance keeps its default initial value of 0
14    if (initialBalance > 0) { // if the initialBalance is valid
15        balance = initialBalance; // assign it to data member balance
16    }
17 }
18
19 // function that deposits (adds) only a valid amount to the balance
20 void deposit(int depositAmount) {
21     if (depositAmount > 0) { // if the depositAmount is valid
22         balance = balance + depositAmount; // add it to the balance
23     }
24 }
25
26 // function returns the account balance
27 int getBalance() const {
28     return balance;
29 }
30
31 // function that sets the name
32 void setName(std::string accountName) {
33     name = accountName;
34 }
35
36 // function that returns the name
37 std::string getName() const {
38     return name;
39 }
40 private:
41     std::string name; // account name data member
42     int balance(0); // data member with default initial value
43 }; // end class Account
44 
```

```

1 // AccountTest.cpp
2 // Displaying and updating Account balances.
3 #include <iostream>
4 #include "Account.h"
5
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 int main()
11 {
12     Account account1("Jane Green", 50);
13     Account account2("John Blue", -7);
14
15     // display initial balance of each object
16     cout << "account1: " << account1.getName() << " balance is $" 
17     << account1.getBalance();
18     cout << "\naccount2: " << account2.getName() << " balance is $" 
19     << account2.getBalance();
20
21     cout << "\n\nEnter deposit amount for account1: "; // prompt
22     int depositAmount;
23     cin >> depositAmount; // obtain user input
24     cout << "adding " << depositAmount << " to account1 balance";
25     account1.deposit(depositAmount); // add to account1's balance
26
27     // display balances
28     cout << "\n\naccount1: " << account1.getName() << " balance is $" 
29     << account1.getBalance();
30     cout << "\naccount2: " << account2.getName() << " balance is $" 
31     << account2.getBalance();
32
33     cout << "\n\nEnter deposit amount for account2: "; // prompt
34     int depositAmount;
35     cin >> depositAmount; // obtain user input
36     cout << "adding " << depositAmount << " to account2 balance";
37     account2.deposit(depositAmount); // add to account2 balance
38
39     // display balances
40     cout << "\n\naccount1: " << account1.getName() << " balance is $" 
41     << account1.getBalance();
42     cout << "\naccount2: " << account2.getName() << " balance is $" 
43     << account2.getBalance() << endl;
44 }
```

Oss: nel programma che definisce la classe sia la funzione membro set che il costruttore ricevono in ingresso una stringa, accountName; tuttavia si tratta di 2 variabili string differenti, poiché entrambe sono variabili locali e vivono all'interno delle 2 funzioni.

Oltre alle funzioni set e get sono state scritte anche le funzioni per il deposito e per il prelievo, che vanno ad alterare la variabile membro *balance*. La funzione *getBalance* (così come quella “*getName*”) è dichiarata come *const* perché non va a modificare l’oggetto della classe *account* sul quale è stata chiamata.

Se vogliamo inserire anche il prelievo va controllato che la cifra inserita per il prelievo sia >0 , sicuramente, ma anche minore o uguale al saldo attuale. Nel caso in cui non sia così va mostrato un messaggio di errore.

ESEMPIO di prima con ritiro (“withdraw”)

```
1 // Account.h
2 // Account class with name and balance data members, and a
3 // constructor and deposit function that each parameter
4 // function returns
5 // Function withdraw
6
7 class Account {
8 public:
9     // Account constructor with two parameters
10    // initializes accountName, but not balance
11    Account(string accountName, int initialBalance);
12    // Name of account
13    string accountName;
14    // Initial balance
15    int initialBalance;
16    // Current balance
17    int balance;
18
19    // Function that deposits amount into a valid account to the balance
20    void deposit(int depositAmount);
21    // Deposit amount > 0; if the depositAmount is valid
22    // balance + depositAmount; if not in the balance
23    // balance = initialBalance
24    // Assign it to data member balance
25
26
27    // Function that withdraws amount from a valid account to the balance
28    void withdraw(int withdrawAmount);
29    // WithdrawAmount > 0; if the withdrawAmount is valid
30    // withdrawAmount <= balance
31    // balance - withdrawAmount; if less than the balance
32    // balance = "Withdraw amount exceed account balance" is returned
33    // else withdraw amount should be > 0 or greater
34
35    // Function returns the account balance
36    int getBalance() const;
37    // Return balance
38
39
40    // Function that sets the name
41    // sets accountName to accountName
42    // Name = accountName;
43
44
45    // Function that deletes the name
46    // destroying getName();
47    // ensure name;
48
49
50 #include
51 #include <iostream>
52 // account.h
53 // app name acc default initial value
54 #include <iomanip>
```

1

La funzione membro *withdraw* restituisce un void, prende un parametro in ingresso che è un intero (*withdrawAccount*) e che ha come scope proprio il corpo della funzione membro.

Altro esercizio:

INVOICE (fattura)

Exercise 2: Invoice

- (**Invoice class**) Create a class called **Invoice** that a hardware store might use to represent an invoice for an item sold at the store. An **Invoice** should include six data members-a part number (type **string**), a part description (type **string**), a quantity of the item being purchased (type **int**), a price per item (type **int**) a value-added tax (VAT) rate as a decimal (type **double**) and a discount rate as a decimal(type **double**).
- Your class should have a constructor that initializes the six data members.
- The constructor should initialize the first four data members with values from parameters and the last two data members to default values of 0.20 per cent and zero respectively.
- Provide a set and a get functions for each data member.
- In addition, provide a member function named **getInvoiceAmount** that calculates the invoice amount (i.e., multiplies the quantity by the price per item and applies the tax and discount amounts), then returns the amount.
- Have the set data members perform validity checks on their parameters—if a parameter value is not positive, it should be left unchanged. Write a driver program to demonstrate **Invoice**'s capabilities.

In sostanza abbiamo bisogno di 6 funzioni set e di 6 funzioni get, ed ovviamente di una funzione costruttore.

```
1 // File: Invoice.h
2 // An invoice class to represent an invoice of sold items
3
4 #include <iostream>
5
6 class Invoice {
7 public:
8     Invoice(string partNumber, string description, int quantity, int price,
9             float vatRate, float discountRate);
10    void setPartNumber(string partNumber);
11    void setDescription(string description);
12    void setQuantity(int quantity);
13    void setPrice(int price);
14    void setVatAmount(float vat);
15    void setDiscountAmount(float discount);
16    string getPartNumber();
17    string getDescription();
18    int getQuantity();
19    int getPrice();
20    int getVatAmount();
21    int getDiscountAmount();
22    float getInvoiceAmount();
23 private:
24    string partNumber;
25    string description;
26    int quantity;
27    int price;
28    float vatRate;
29    float discountRate;
30};

31 // Preamble for the account example
32 testInvoice: testInvoice.o Invoice.o
33 g++ -o testInvoice testInvoice.o Invoice.o
34 testInvoice: testInvoice.o Invoice.o
35 g++ -Wall -pedantic -std=c++11 -o testInvoice.o
36 Invoice.o Invoice.o Invoice.o
37 g++ -Wall -pedantic -std=c++11 -o Invoice.o
38
39
40
```

```
1 // File: Invoice.cpp
2 // Source code implementing the Invoice class
3
4 #include "Invoice.h"
5
6 Invoice::Invoice(string partNumber, string description, int quantity, int price,
7                   float vatRate, float discountRate) {
8     partNumber = partNumber;
9     description = description;
10    quantity = quantity;
11    price = price;
12    vatRate = vatRate;
13    discountRate = discountRate;
14}
15
16 void Invoice::setPartNumber(string partNumber) {
17     partNumber = partNumber;
18}
19
20 void Invoice::setDescription(string description) {
21     description = description;
22}
23
24 int Invoice::getQuantity() {
25     if (quantity > 0) {
26         return quantity;
27     }
28     else {
29         return 0;
30     }
31}
32
33 int Invoice::getPrice() {
34     if (price > 0) {
35         return price;
36     }
37     else {
38         return 0;
39     }
40}
41
42 int Invoice::getVatAmount() {
43     if (vatRate > 0.0 & vatRate < 0.4) {
44         return (int)(vatRate * price);
45     }
46     else {
47         return 0;
48     }
49}
50
51 int Invoice::getDiscountAmount() {
52     if (discountRate == 0.0) {
53         return 0;
54     }
55     else {
56         return (int)(discountRate * price);
57     }
58}
59
60 float Invoice::getInvoiceAmount() {
61     if (quantity > 0 & price > 0) {
62         float amount = quantity * price;
63         amount += amount * vatRate;
64         amount -= amount * discountRate;
65         return amount;
66     }
67     else {
68         return 0;
69     }
70 }
```

Può convenire mettere insieme nella parte public della classe tutte le intestazioni (signatures) delle funzioni ed andare poi a sviluppare le funzioni in un altro file, ad esempio in quello che è stato chiamato **Invoice.cpp**.

Quindi l'applicazione sarà costituita da 3 file: il file .h che contiene le intestazioni, il file .cpp che contiene l'implementazione delle funzioni ed un file (testInvoice.cpp) che istanzia alcuni di questi oggetti ed invoca le funzioni membro. Di seguito il file testInvoice.cpp:

```
1 // File: testInvoice.cpp
2 // Written: program by Yann Viennot
3
4 #include <iostream> // I/O
5 #include "Invoice.h" // Our class
6
7 using namespace std;
8
9 int main() {
10     // Create 2 instances of Invoice
11     //    - one for 100% of invoice
12     //    - one for 50% of invoice
13     Invoice inv1("Bill-01", "Belgium white wine", 5, 100);
14     // value is 100% of 100 = 100
15     Invoice inv2("Bill-02", "Bordeaux red wine", 5, 50);
16     // value is 50% of 100 = 50
17
18     // Print total amount
19     cout << inv1.getSubtotal() << endl << inv1.getGrandTotal() << endl;
20     cout << inv2.getSubtotal() << endl << inv2.getGrandTotal() << endl;
21
22     return 0;
23 }
```

Ci sono altri 2 esercizi, 1 sui motori ed 1 sulle date.

LEZIONE 5

Tutor- VS Code, MobaxTerm

LEZIONE 6

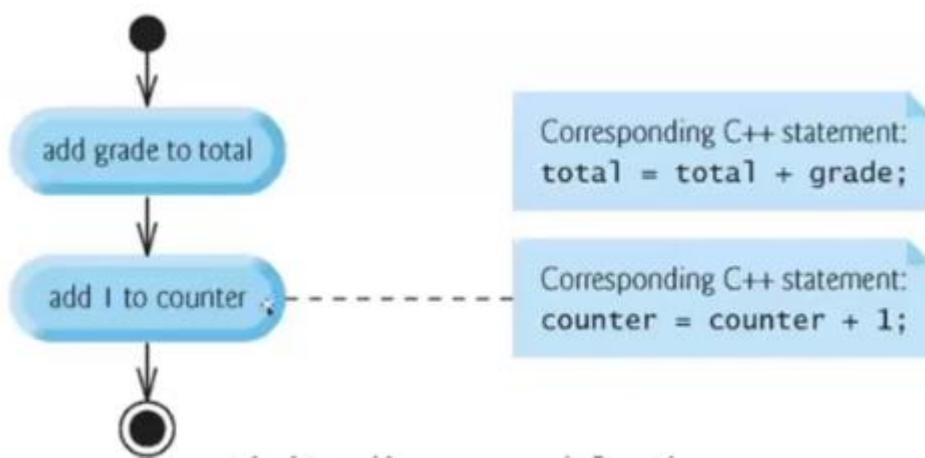
Vediamo le strutture fondamentali per il controllo del flusso.

Qualsiasi algoritmo può essere scritto utilizzando le 3 strutture di controllo fondamentali:

- Sequenza
- Selezione
- Iterazione

Sulla sequenza non c'è molto da dire. In UML la struttura di sequenza la si rappresenta con dei blocchi che contengono i processi fondamentali e delle frecce che stabiliscono la precedenza; ci sono 2 nodi poi che indicano la partenza e la fine. Ai blocchi si possono associare anche delle note che ne descrivono meglio le istruzioni.

Es:



La struttura di selezione la si realizza tramite le istruzioni *if*, *if...else* e *switch*.

La struttura di iterazione la si realizza tramite le istruzioni *while*, *do...while* e *for*.

Tutte le parole if, else, switch, while e do sono dette **keywords**, o parole chiave, e sono riservate dal linguaggio C++, per cui non possono essere utilizzate come identificativi (ad esempio come nomi di variabili). Ce ne sono molte altre di keywords:

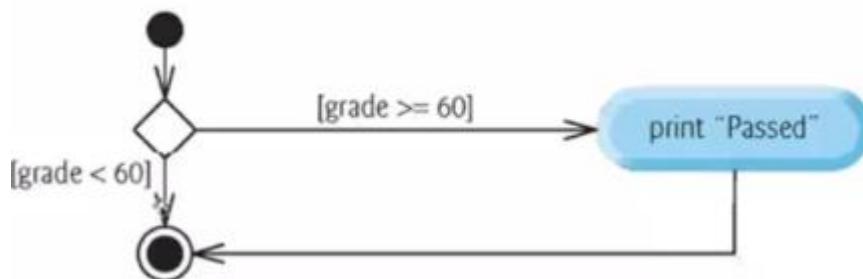
C++ Keywords				
<i>Keywords common to the C and C++ programming languages</i>				
asm	auto	break	case	char
const	continue	default	do	double
else	enum	extern	float	for
goto	if	inline	int	long
register	return	short	signed	sizeof
static	struct	switch	typedef	union
unsigned	void	volatile	while	
<i>C++-only keywords</i>				
and	and_eq	bitand	bitor	bool
catch	class	compl	const_cast	delete
dynamic_cast	explicit	export	false	friend
mutable	namespace	new	not	not_eq
operator	or	or_eq	private	protected
public	reinterpret_cast	static_cast	template	this
throw	true	try	typeid	typename
using	virtual	wchar_t	xor	xor_eq

C++ Keywords				
<i>C++11 keywords</i>				
alignas	alignof	char16_t	char32_t	constexpr
decltype	noexcept	nullptr	static_assert	thread_local

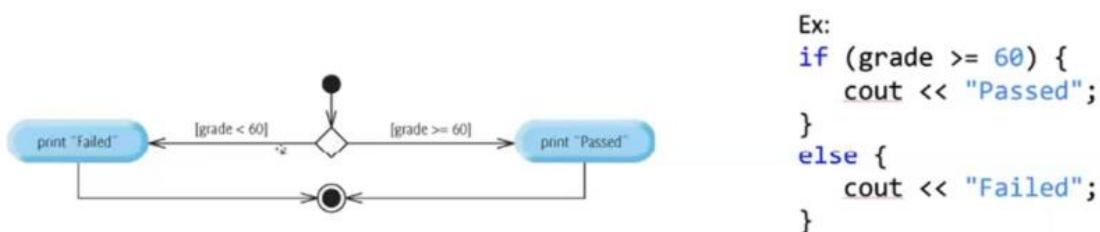
La selezione in UML si rappresenta con un rombo vuoto da cui partono 2 frecce: una è quella del caso in cui la condizione dell'if sia vera, l'altra del caso in cui la condizione sia falsa.

Es:

Ex:
`if (studentGrade >= 60) {
 cout << "Passed";
}`



Similmente nel caso dell'if-else:



C'è poi un altro operatore che non abbiamo visto a fondamenti di informatica e che si chiama:

Operatore Condizionale (? :)

L'anatomia del condizionale è quella di una condizione, punto interrogativo e due punti.

Es:

```
1 #include <iostream>
2 using std::cin;
3 using std::cout;
4
5 int main () {
6     // Local variable declaration:
7     int age=0;
8     std::string message;
9     cout << "What's your age? ";
10    cin >> age;
11    message = (age <= 25) ? "Youth is wonderful, enjoy it!":"Age is a mental state";
12    cout << message << "\n";
13
14    return 0;
15 }
```

Equivalent to:

```
1 #include <iostream>
2 using std::cin;
3 using std::cout;
4
5 int main () {
6     // Local variable declaration:
7     int age=0;
8     std::string message;
9     cout << "What's your age? ";
10    cin >> age;
11    if (age <= 25)
12        message = "Youth is wonderful, enjoy it!";
13    else
14        message = "Age is a mental state";
15    cout << message << "\n";
16    return 0;
17 }
```

Se la condizione tra parentesi è vera, allora message prende la prima clausola, ovvero quella prima dei 2 punti. Se la condizione è falsa allora message assume il valore della sequenza di caratteri dopo i 2 punti. Sostanzialmente questo operatore non è altro che un operatore di assegnazione condizionale di tipo if-else, all'interno di una sola espressione.

Di seguito invece un esempio di programma con utilizzo del while:

```
1 //ClassAverage.cpp
2 // Computing class-average using sentinel-controlled iteration.
3 #include <iostream>
4 #include <iomanip> // parameterized stream manipulators
5 using namespace std;
6 int main() {
7     // initialization phase
8     int total{0}; // initialize sum of grades
9     unsigned int gradeCounter{0}; // initialize # of grades entered so far
10    cout << "Enter grade or -1 to quit: ";
11    int grade;
12    cin >> grade;
13    // loop until sentinel value read from user
14    while (grade != -1) {
15        total = total + grade; // add grade to total
16        gradeCounter = gradeCounter + 1; // increment counter
17        // prompt for input and read next grade from user
18        cout << "Enter grade or -1 to quit: ";
19        cin >> grade;
20    }
21    // termination phase
22    // if user entered at least one grade...
23    if (gradeCounter != 0) {
24        // use number with decimal point to calculate average of grades
25        double average{static_cast<double>(total) / gradeCounter};
26        // display total and average (with two digits of precision)
27        cout << "\nTotal of the " << gradeCounter
28            << " grades entered is " << total;
29        cout << setprecision(2) << fixed;
30        cout << "\nClass average is " << average << endl;
31    }
32    else { // no grades were entered, so output appropriate message
33        cout << "No grades were entered" << endl;
34    }
35 }
36 }
```

Alla riga 25 notiamo l'utilizzo del casting, ovvero della variazione di tipo all'interno di un'espressione. Le variabili *total* e *gradeCounter* sono entrambe di tipo intero, per cui se effettuassimo una loro divisione uscirebbe un intero poiché la parte decimale sarebbe troncata prima dell'assegnazione del risultato alla variabile *average*, di tipo double. Quindi la variabile *average* uscirebbe di tipo intero anche se definita come double. Per questo si utilizza un operatore di static cast, che fa una conversione di tipo esplicita. Grazie a *static_cast<double>(total)* viene creata una copia temporanea di *total* di tipo double, anche se *total* resta un intero, ma in questa maniera il risultato sarà effettivamente calcolato in virgola mobile; infatti, visto che le operazioni si effettuano tra variabili dello stesso tipo, una volta che *total* è diventato double anche *gradeCounter* lo diventa, ricevendo una “promozione” automatica al tipo double.

Notiamo poi che alla riga 29 c'è un comando particolare che è *setprecision(2)*: questo comando serve per visualizzare 2 cifre decimali per i valori a virgola mobile e quindi a specificare la volontà di lavorare in doppia precisione. Per utilizzare questa funzionalità bisogna includere una libreria che si chiama *iomanip* e che sta per “input-output manipulation”. Se non specifichiamo la precisione con cui vogliamo visualizzare valori reali automaticamente questa sarà fissata a 6. Invece *fixed* è definita

all'interno di ***iostream***: serve a visualizzare i valori float e double in notazione, per l'appunto, *fixed*, che si oppone alla notazione scientifica che usa mantissa ed esponente.

Quindi in definitiva alle righe 29-30 si sta dicendo: “Stampa a video il valore della variabile *average* con virgola fissa e con 2 cifre decimali.

Alla riga 14 abbiamo utilizzato un valore sentinella per terminare il loop del while.

ESERCIZIO:

- A **palindrome** is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in an integer and determines whether it's a palindrome.
-

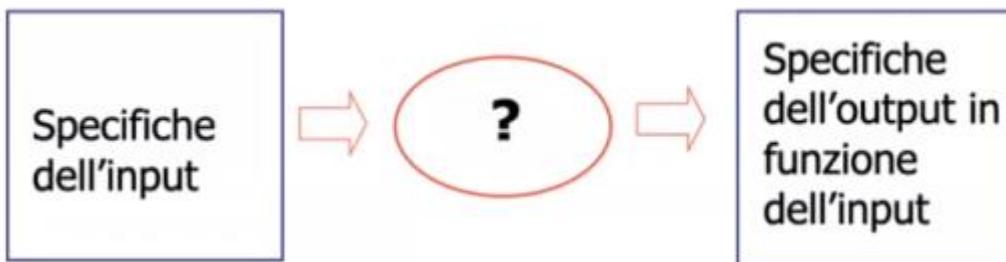
MALE

LEZIONE 7

ANALISI DEGLI ALGORITMI

Va distinto il problema algoritmico dall'algoritmo.

Quando definiamo un problema algoritmico definiamo quali sono le specifiche di input e quali sono le specifiche dell'output in funzione dell'input:



Ad esempio il problema dell'ordinamento consiste in un insieme di valori confrontabili tra di loro, come specifiche di input, mentre in una permutazione dei valori in ingresso secondo un criterio che soddisfi il criterio dell'ordinamento, come specifica di output.

Per risolvere il problema algoritmico definito si possono utilizzare infiniti algoritmi.

L'efficienza di un algoritmo consiste nel suo tempo di esecuzione.

ORDINAMENTO

Nel caso dell'ordinamento una soluzione potrebbe essere quella di considerare tutte le possibili permutazioni fino a trovare quella che soddisfa il problema dell'ordinamento. Tuttavia, però, esplorare tutte le possibili permutazioni vuol dire, dati n numeri, effettuare " $n!$ " operazioni! Si intuisce quindi che al crescere del numero di elementi da ordinare l'algoritmo risulta poco efficiente, poiché il numero di operazioni da effettuare aumenta in maniera esponenziale.

Un possibile algoritmo di ordinamento è il seguente:

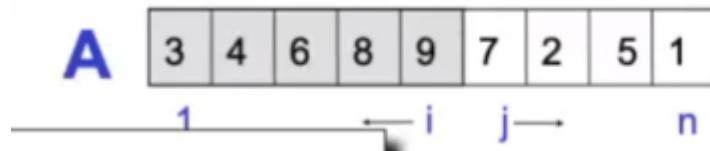
INSERTION SORT

È quello che usiamo a nostra insaputa quando ordiniamo le carte durante una partita a scala 40, ad esempio. Una parte è già ordinata e l'altra va ordinata. La parte ordinata cresce di un elemento ad ogni iterazione, quella da ordinare decresce allo stesso modo.

```
for (int j = 1; j < numbers.length; j++) {  
    int key = numbers[j];  
    int i = j;  
    //inserisci numbers[j] in  
    //numbers[0,...,j-1]  
    while (i > 0 && numbers[i-1] > key) {  
        numbers[i] = numbers[i-1];  
        i--;  
    }  
    numbers[i] = key;  
}
```

KEY=ELEMENTO DA ORDINARE

Si parte dal primo elemento, che può essere considerato ordinato, e si procede nell'ordinamento; per questo il ciclo for parte da 1 e non da 0. Mano a mano la lista degli elementi ordinati cresce e l'elemento j-esimo viene aggiunto ai precedenti j-1, che sono già ordinati, confrontandolo con questi. Ad esempio:



Se gli elementi in grigio sono già ordinati, il 7 lo si confronta con 9: visto che è minore li si scambiano di posto; a questo punto si confronta il 7 con l'elemento ancora precedente, ovvero 8, ed essendo 7 minore di 8 allora si sposta 7 ancora di uno a sinistra. A questo punto 7 è maggiore di 6 e quindi rispetta il criterio dell'ordinamento e la lista degli elementi ordinati si accresce di un elemento. Si passa quindi al 2 e si fa la stessa cosa, poi al 5 e poi all'1. Volendoci riferire all'algoritmo, se consideriamo proprio 7 allora:

```
int key=numbers [j]=7;
```

```
int i=j=5 (quinta posizione);
```

Nel while poi "i" viene decrementato di volta in volta.

Quanto è efficiente questo algoritmo? Dobbiamo valutare:

- **Complessità computazionale**
- **Caso peggiore, caso medio, caso migliore**
- **Notazione asintotica**
- **Esempi di calcolo della complessità**

Nello sviluppare un programma poi ci deve interessare

Sviluppare algoritmi che utilizzano in maniera efficiente le ricorse del calcolatore.

Analizzare un algoritmo vuol dire essere capaci di prevedere le risorse che l'algoritmo richiede; e quindi: il tempo richiesto, lo spazio utilizzato, lo sforzo del programmatore, la facilità d'uso, etc. Queste quantità possono essere misurate in maniera empirica o analitica, seppur approssimata. Nel primo caso, ad esempio, si cronometra il tempo di esecuzione; lo svantaggio di questo approccio è che il tempo misurato dipenderà dal sistema, dal linguaggio, etc. Nel secondo caso, invece, si farà dipendere il tempo t dalla

dimensione dell'input n e si valuta quanto il numero di operazioni cresce all'aumentare della dimensione dell'input. Ovviamente un approccio di questo tipo per essere massimamente preciso richiederebbe un livello di dettaglio notevole: bisognerebbe valutare ogni operazione per vedere se richieda accessi in memoria o se utilizzi registri interni del processore, bisognerebbe valutare le prestazioni del computer sul quale l'algoritmo gira, e così via. Noi ci fermiamo al livello di dettaglio detto:

MODELLO RAM (RANDOM ACCESS MACHINE)

Questo modello assume che tutte le operazioni hanno un tempo costante e sono aritmetiche, di movimento dei dati (assegnamenti) o di controllo (salti, chiamate a subroutine, return) e che tutti i tipi di dati sono interi e float. Facendo queste assunzioni si riesce a trovare effettivamente $T(n)$, ovvero il tempo di esecuzione dell'algoritmo in funzione della dimensione dell'input. T non è espresso in unità di tempo, come millisecondi, ma come numero di operazioni elementari eseguite per il calcolo dell'output. Ogni singola operazione elementare è detta anche *passo* e l'assunzione di base è che ciascun passo richieda un tempo costante che eventualmente può variare da istruzione a istruzione.

Vediamo l'analisi dell'*insertion sort*.

INSERTION SORT

Analisi di Insertion Sort

- Calcolo del **tempo richiesto per l'esecuzione dell'algoritmo** espresso in funzione della dimensione dell' **input**

	costo	passi
<code>for (int j = 1; j < numbers.length; j++) {</code>	c_1	n
<code> int key = numbers[j];</code>	c_2	$n-1$
<code> int i = j;</code>	c_3	$n-1$
<code> //inserisci numbers[j] in</code>	0	
<code> //numbers[0,...,j-1]</code>	0	
<code> while (i > 0 && numbers[i-1] > key) {</code>	c_4	$\sum_{j=2}^n t_j$
<code> numbers[i] = numbers[i-1];</code>	c_5	$\sum_{j=2}^n (t_j - 1)$
<code> i--;</code>	c_6	
<code>}</code>	0	
<code> numbers[i] = key;</code>	c_7	$n-1$
<code>}</code>	0	

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

Quindi il tempo di esecuzione è dato da:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

Questa funzione dipende da t_j : nel caso migliore, ovvero quello in cui la lista è già ordinata, non si entra mai nel while ed è quello per cui si ha $t_j=1$.

Il caso peggiore è quello in cui le carte sono ordinate in maniera decrescente. Analizziamo i 2 casi separatamente.

CASO FORTUNATO:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

▪ Caso migliore:

- Se l'array è già ordinato il corpo del ciclo while non è mai eseguito: $t_j=1$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

$T(n) = an+b$

Il caso migliore quindi prevede un tempo di elaborazione *LINEARE*, ovvero il numero di operazioni cresce con il numero di elementi.

CASO SFORTUNATO:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

▪ Caso peggiore:

- Se l'array è ordinato in ordine inverso: $t_j=j$

Infatti il j-esimo elemento va spostato fino alla prima posizione, e questo vale per tutti gli elementi.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

• Caso peggiore:

- Se l'array è ordinato in ordine inverso: $t_j = j$

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

Quindi:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left(\frac{n(n+1)}{2} - 1 \right) + \\ &+ c_5 \left(\frac{n(n-1)}{2} \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7(n-1) \end{aligned}$$

$$T(n) = an^2 + bn + c$$

16

Stavolta T cresce come n^2 , ovvero in maniera parabolica.

CASO MEDIO:

Quello che ci può interessare è MEDIANTE cosa ci possiamo aspettare: beh, nel caso medio $t_j = j/2$. In tal caso:

• Caso medio:

- Se l'array è casuale, in media metà degli elementi saranno maggiori dell'elemento da inserire: $t_j = j/2$

$$T(n) = a'n^2 + b'n + c'$$

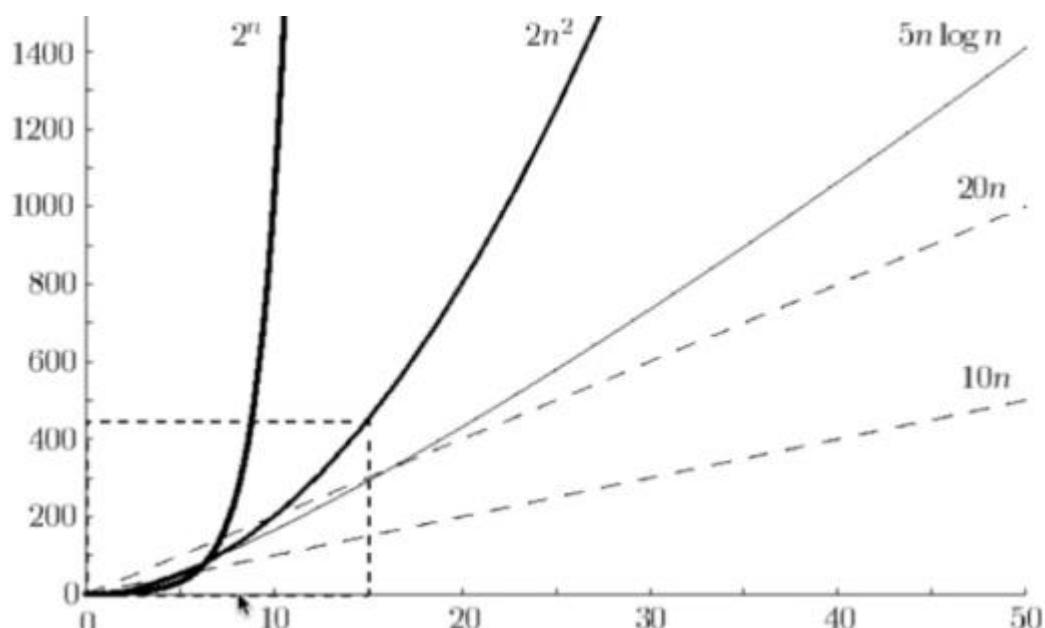
$$\sum_{j=2}^n t_j = \sum_{j=2}^n \frac{j}{2} = \frac{1}{2} \left(\sum_{j=1}^n j - 1 \right) = \frac{n^2 + n - 2}{4}$$

$$\sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n \left(\frac{j}{2} - 1 \right) = \frac{n^2 - 3n + 2}{4}$$

18

Insomma, ci troviamo ancora una dipendenza quadratica. Questa è una brutta notizia.

Il tempo di elaborazione medio quindi non è molto buono, e questo fa sì che questi altri algoritmi di ordinamento siano preferiti a questo. In generale gli algoritmi che hanno un tempo medio di esecuzione più vicino al caso peggiore che a quello migliore non sono molto efficienti. Tipicamente nell'analizzare un algoritmo si fa sempre un'analisi del tipo *worst case*, poiché definisce un limite superiore da non superare del numero di passi. Il caso medio poi non è sempre facile da studiare perché richiede anche un'analisi statistica. Ci interessano tutti e tre i casi, ma ha una certa priorità quello peggiore. Abbiamo poi capito che non ci interessa il numero di secondi ma il numero di operazioni da effettuare e quindi alla forma della curva dell'algoritmo nel piano: (numero di operazioni, tempo di esecuzione).



Naturalmente i tempi di esecuzione si riducono di 10 volte se prendiamo un calcolatore 10 volte più veloce. Però a noi non interessa tanto questo, quanto un altro quesito, ovvero di quanto aumenta la dimensione del problema che riusciamo a risolvere in un certo tempo. Avere un'idea della complessità dell'algoritmo ci dice a che livello di dettaglio possiamo risolvere il problema in termini della dimensione dell'input che riusciamo a trattare.

In particolare, se "n" è la dimensione che può essere processata in 10.000 passi, se compro un calcolatore 10 volte più veloce invece di 10.000 passi posso farne 100.000 di passi; la dimensione che posso processare con 100.000 passi la chiamo "n'". Quello che ci interessa è il rapporto che c'è tra n ed n', ovvero: se compro un computer più veloce di quanto aumenta la dimensione del problema che riesco a risolvere, nella stessa unità di tempo?

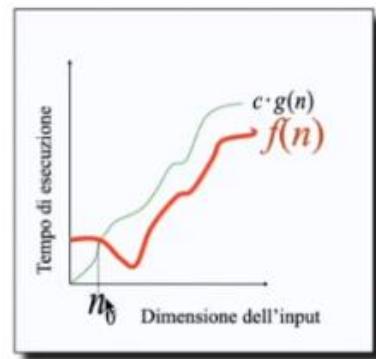
$T(n)$	n	n'	Incremento	n'/n
$10n$	1,000	10,000	$n'=10n$	10
$20n$	500	5,000	$n'=10n$	10
$5n\log n$	250	1,842	$\sqrt{10n} < n' < 10n$	7.37
$2n^2$	70	223	$n'=\sqrt{10n}$	3.16
2^n	13	16	$n'=n+3$	--

Se $T(n)$ è lineare allora $(n'/n)=10 \rightarrow$ riesco a risolvere un problema 10 volte più grande. Se però $T(n)$ ha andamento logaritmico vedo che $(n'/n)=7.37$, quindi con un calcolatore 10 volte più veloce riesco a risolvere un problema 7 volte più grande; e ancora può andare bene. Tuttavia se $T(n)$ cresce come n^2 allora la dimensione trattabile cresce ma di poco, poiché è pari a $n^*(\sqrt{10})$ e quindi tra n' ed n c'è un rapporto di circa 3. La cosa più drammatica è che nel caso in cui $T(n)$ abbia andamento esponenziale, non c'è nemmeno più un rapporto tra n' ed n ! Si ha infatti $n'=n+3$ e quindi sostanzialmente comprare un computer 10 volte più veloce quasi non reca vantaggi. Questo fa sì che algoritmi con complessità dell'ordine 2^n non vengano considerati affatto.

Dei fattori costanti non tanto ci interessa, perché dipendono dalla tecnologia e da altre questioni e la relazione $T(n)$, come abbiamo già accennato, non è facile da calcolare. Quello che ci interessa per analizzare un algoritmo è vedere “asintoticamente” cosa succede, e quindi come varia il tempo di esecuzione in funzione della dimensione in input, ovvero n . Per questo parliamo di *complessità asintotica*.

Come notazione asintotica usiamo la notazione O - “O-Grande”:

- **The notazione O - “O-Grande”**
 - Limite superiore asintotico
 - $f(n) = O(g(n))$, esistono una costante c ed un intero n_0 , tali che
 $f(n) \leq c g(n)$ per ogni $n \geq n_0$
 - $f(n)$ e $g(n)$ sono funzioni definite sugli interi positivi
- Utilizzata per l'analisi del caso peggiore (worst-case)



La notazione $f(n)=O(g(n))$ sta a significare che $f(n)$ da un certo punto in poi è dominata da $g(n)$ e tradotta vuol dire “ f è dell’ordine di g ”. In realtà f è dominata non da g ma da g^*c , dove c è un termine costante. Ad esempio nel caso di:

$$f(n) = n^2 + 100n + \log_{10}n + 1000$$

da un certo punto in poi conta solo n^2 , e lo si può vedere fissando dei valori di n :

n	$f(n)$	n^2		100n		$\log_{10}n$		1000		
		Valore	Valore	%	Valore	%	Valore	%	Valore	%
1	1101	1	0.1		100	9.10	0	0.0	1000	90.8
10	2101	100	4.76		1000	47.6	1	0.05	1000	47.6
100	21002	10000	47.6		10000	47.6	2	0.99	1000	4.76
1000	1101003	1000000	90.8		100000	9.10	3	0.000	1000	0.09
10 000	101001004	100000000	99.0		1000000	0.99	4	0.0	1000	0.00
100 000	10010001005	10000000000	99.0		10000000	0.09	5	0.0	1000	0.00

Da $n_0=1000$ in poi possiamo dire che di fatto n^2 domina su tutto il resto, e quindi:

$$f(n) = O(n^2)$$

e da $n=n_0$ $f(n)$ sarà sempre maggiorata da c^*n^2 , con $c=100$ ad esempio. Infatti n^2 starà al di sotto di $100 n^2$, e sicuramente lo stesso vale per tutti gli altri termini che già di per sé da un certo punto in poi stanno sotto a n^2 .

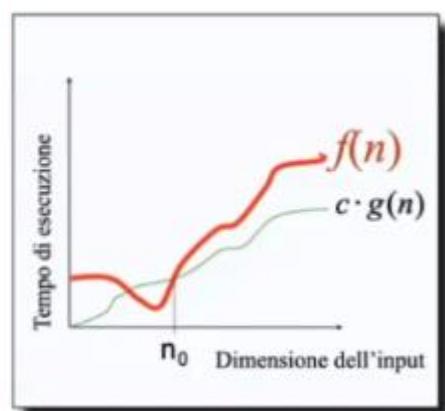
Si utilizza altresì la notazione Omega-Grande per il limite inferiore asintotico di $f(n)$:

La notazione Ω - “Omega Grande”

- Limite inferiore asintotico
- $f(n) = \Omega(g(n))$ se esistono una costante c ed un intero n_0 , tali che $c g(n) \leq f(n)$ per ogni $n \geq n_0$

Utilizzata per descrivere il tempo di esecuzione del caso migliore, o per descrivere limiti inferiori di problemi algoritmici

- Ad es., il limite inferiore per la ricerca in un array non ordinato è $\Omega(n)$.



Stavolta questa notazione sta ad indicare che $f(n)$ sta al di sopra di $c^*g(n)$ da un certo n_0 in poi. Questo è più interessante per il caso migliore di un algoritmo. Ad esempio, per l'insertion sort il caso migliore era $T(n)=an+b$ e quindi possiamo dire che l'insertion sort ha una complessità di tempo almeno $\Omega(n)$, a meno di un fattore costante.

Semplice regola: Trascura termini di ordine inferiore e fattori costanti.

- $50n \log n$ è $O(n \log n)$
- $7n - 3$ è $O(n)$
- $8n^2 \log n + 5n^2 + n$ è $O(n^2 \log n)$
- Una costante è $O(1)$

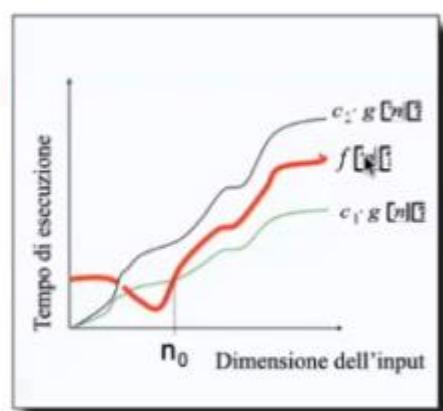
Nota: Anche se $(50n \log n)$ è $O(n^5)$, ci si aspetta che tale approssimazione sia quanto più stretta possibile

C'è poi un'altra notazione asintotica, ovvero Theta-Grande:

La notazione Θ - "Theta Grande"

- Limite asintoticamente stretto
- $f(n) = \Theta(g(n))$ esistono due costanti c_1, c_2 , ed un intero n_0 , tali che $c_1g(n) \leq f(n) \leq c_2g(n)$ per ogni $n \geq n_0$

$f(n) = \Theta(g(n))$ se e solo se
 $f(n) = \Omega(g(n))$ e $f(n) = O(g(n))$
 $O(g(n))$ è spesso
erroneamente utilizzata al
posto di $\Theta(g(n))$



Questa notazione si chiama limite asintoticamente stretto ed indica che, a partire da un certo n_0 , $f(n)$ è compresa tra $g(n)*c_1$ e $g(n)*c_2$, e quindi $g(n)$ è sia limite inferiore che superiore di $f(n)$. Quindi ad esempio per quanto riguarda la funzione di prima:

$$f(n) = n^2 + 100n + \log_{10}n + 1000$$

Possiamo dire, più correttamente di prima, che:

$$f(n) = \Theta(n^2), n_0 = 1000, c_1 = 100, c_2 = 0.5$$

E quindi in generale se f è sia “Omega” che “O” di $g(n)$ allora vale la relazione di limite asintoticamente stretto, ovvero “Theta”. L’analisi fatta con Theta è ovviamente più precisa rispetto a considerare solo O, ad esempio. Teniamo in mente allora questo:

$$\begin{array}{ll} f(n) = O(g(n)) & \Leftrightarrow f \leq g \\ f(n) = \Omega(g(n)) & \Leftrightarrow f \geq g \\ f(n) = \Theta(g(n)) & \Leftrightarrow f = g \end{array}$$

Come si combinano espressioni di tipo O-grande? C’è una regola: sostanzialmente la somma è ancora un O-grande del massimo tra i 2 ordini. La “dimostrazione” è questa:

- Supp. Una parte dell’algoritmo prende $O(n^2)$ ed un’altra $O(n^3)$
 - Come sommare i due limiti superiori per avere il tempo totale dell’algoritmo?
- Supp. $T_1(n)=O(f_1(n))$ e $T_2(n)=O(f_2(n))$ se inoltre $f_2(n)=O(f_1(n))$ allora $T_1(n)+T_2(n)=O(f_1(n))$

Esistono delle costanti $n_1, n_2, n_3, c_1, c_2, c_3$

- Se $n \geq n_1$ allora $T_1(n) \leq c_1 f_1(n)$
- Se $n \geq n_2$ allora $T_2(n) \leq c_2 f_2(n)$
- Se $n \geq n_3$ allora $f_2(n) \leq c_3 f_1(n)$

Sia $n_0 \geq n_1, n_2, n_3$ allora per $n \geq n_0$

$$\begin{aligned} T_1(n)+T_2(n) &\leq c_1 f_1(n)+c_2 f_2(n) \leq \\ &c_1 f_1(n)+c_2 c_3 f_1(n) \end{aligned}$$

PROPRIETA' di O-GRANDE:

- Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$, allora $f(n)$ è $O(h(n))$.
- Se $f(n)$ è $O(kg(n))$ per qualche costante $k > 0$, allora $f(n)$ è $O(g(n))$.
- Se $f_1(n)$ è $O(g_1(n))$ e $f_2(n)$ è $O(g_2(n))$, allora $(f_1 + f_2)(n)$ è $O(\max(g_1(n), g_2(n)))$.
- Se $f_1(n)$ è $O(g_1(n))$ e $f_2(n)$ è $O(g_2(n))$, allora $f_1(n)f_2(n)$ è $O(g_1(n)g_2(n))$.

La 2 permette di trascurare le costanti, la 4 è importante ad esempio nei cicli innestati; se ho $f_1(n) = O(n^2)$ e $f_2(n) = O(n)$, allora $f_1(n)*f_2(n) = O(n^3)$.

Vediamo qualche esempio:

Calcolo della complessità

a = b; //assegnazione

Un'istruzione di assegnazione ha complessità $\Theta(1)$

```
sum = 0;
for (i=1; i<=n; ++i)
    sum += n;
```

Il corpo del **for** ha complessità $\Theta(1)$,
è eseguito n volte, risultato: $\Theta(n)$

```
sum = 0;
for (j=1; j<=n; ++j)
    for (i=1; i<=j; ++i)
        sum++;
for (k=0; k<n; ++k)
    A[k] = k;
```

Il doppio ciclo **for** è $\sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2)$

Il terzo ciclo **for** è $\Theta(n)$
Risultato: $\Theta(n^2)$



Calcolo della complessità (2)

```
sum1 = 0;
for (j=1; j<=n; ++j)
    for (i=1; i<=n; ++i)
        sum1++;
sum2 = 0;
for (j=1; j<=n; ++j)
    for (i=1; i<=j; ++i)
        sum2++;
```

Il primo doppio ciclo è n^2 ,
il secondo è $n(n+1)/2$: Risultato $\Theta(n^2)$

```
sum1 = 0;
for (k=1; k<=n; k*=2)
    for (j=1; j<=n; ++j)
        sum1++;
sum2 = 0;
for (k=1; k<=n; k*=2)
    for (i=1; i<=k; ++i)
        sum2++;
```

Il primo doppio ciclo è $\sum_{k=1}^{\log n} n = \underbrace{n + n + \dots + n}_{\log n \text{ volte}} = \Theta(n \log n)$

Il secondo doppio ciclo è $\Theta(n)$
Risultato: $\Theta(n^2)$

$$\sum_{m=1}^{\log n - 1} 2^m = 2^1 + 2^2 + \dots + 2^{\log n - 1} = \frac{2^{\log n} - 1}{2 - 1} = \Theta(n) \quad 40$$

LEZIONE 8

Oggi vedremo come argomento le:

FUNZIONI

Abbiamo già visto le funzioni nel corso di fondamenti e le abbiamo già viste parlando di classi, però ora entreremo più nel dettaglio sullo scambio dei parametri e sull'utilizzo dello stack, in virtù del fatto che vorremo lavorare con funzioni ricorsive.

In generale usiamo le funzioni soprattutto per la riutilizzabilità del codice e per l'utilizzo di pezzi di codice quando sono necessari all'interno di programmi più grandi, e questo rende possibile l'hiding (ovvero il mascheramento) dei dettagli dell'implementazione del pezzo di codice stesso. Una funzione è invocata da una funzione chiamante e quando termina il suo compito restituisce eventualmente un risultato e certamente il controllo alla funzione chiamante.

Non tutte le funzioni sono membri di una classe, ed è il caso delle *funzioni globali*. Queste funzioni sono specificate in alcuni header particolari, come `<cmath>`, per cui possono essere utilizzate una volta incluso l'header in questione all'interno del programma.

Come già detto, le funzioni le abbiamo viste anche parlando delle classi: noi definiamo ogni classe in un file header (.h) e la includiamo prima del main nel programma contenente il codice sorgente. Fare questo assicura che la classe (e quindi le funzioni membro in essa definite) sia definita prima che il main crei e manipoli gli oggetti di quella classe. Il compilatore poi si assicura che chiamiamo correttamente i costruttori e le funzioni membro di ogni classe (ad esempio si assicura che passiamo il numero corretto di parametri ed il loro giusto tipo). Il preprocessore ha il compito di aggiungere il contenuto del file.h all'interno del nostro file; il compilatore genera il codice oggetto ed infine il linker genera il vero e proprio file eseguibile.

Tipicamente una funzione la si definisce prima con una signature, ovvero con un prototipo che ne esplica solo i parametri.

Nell'esempio della pagina successiva il prototipo sta alla riga 9 ed è il prototipo di una funzione che calcola il massimo tra 3 valori.

Es:

```
1 // maximum.cpp
2 // maximum function with a function prototype.
3 #include <iostream>
4 #include <iomanip>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int maximum(int x, int y, int z); // function prototype
10
11 int main() {
12     cout << "Enter three integer values: ";
13     int int1, int2, int3;
14     cin >> int1 >> int2 >> int3;
15
16     // invoke maximum
17     cout << "The maximum integer value is: "
18         << maximum(int1, int2, int3) << endl;
19 }
20
21 // returns the largest of three integers
22 int maximum(int x, int y, int z) {
23     int maximumValue{x}; // assume x is the largest to start
24
25     // determine whether y is greater than maximumValue
26     if (y > maximumValue) {
27         maximumValue = y; // make y the new maximumValue
28     }
29
30     // determine whether z is greater than maximumValue
31     if (z > maximumValue) {
32         maximumValue = z; // make z the new maximumValue
33     }
34
35     return maximumValue;
36 }
37
```

Naturalmente il prototipo deve essere identico alla definizione vera e propria della funzione, anche se il C++ ci permette di scrivere diversi prototipi della stessa funzione con argomenti diversi (function overloading(?)).

La funzione riceve uno o più parametri in ingresso e restituisce eventualmente un parametro in uscita. I parametri in ingresso vanno dichiarati premettendo sempre il loro tipo, ad ognuno di loro.

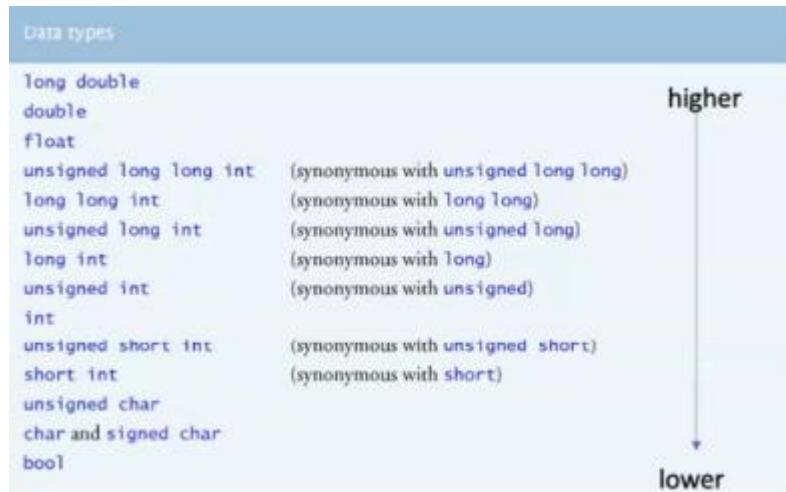
Tipicamente le funzioni vanno fatte sintetiche in modo tale che ci siano varie funzioni, ognuna corrispondente ad un task.

Il prototipo serve SEMPRE tranne quando definiamo la funzione prima di utilizzarla. Infatti nei file.h che includiamo, come `<cmath>`, non c'è l'implementazione di `log(x)` o `cos(x)` ma ci sono solo i prototipi che vengono linkati all'atto del linking.

Lo scopo (*scope*) di una funzione è la regione di programma in cui la funzione è riconosciuta ed accessibile.

Una cosa importante da sapere è la *coercione degli argomenti*, cioè il cambiamento dinamico del tipo degli argomenti che può essere di promozione o di restrizione. Infatti quando richiamo una funzione per utilizzarla potrei dichiarare uno dei parametri di ingresso come float invece che int, ad esempio, o viceversa: nel primo caso otterrei un errore di compilazione, nel secondo caso l'intero sarebbe promosso a reale

automaticamente e non si avrebbe alcun errore di compilazione. La restrizione, quindi, va fatta tramite un cast esplicito, ovvero un cambiamento di tipo; la promozione è invece automatica e dinamica. La gerarchia dei tipi è questa:



La libreria standard di C++ (<cstdlib>) è divisa in tante parti, ognuna con il suo header file. Gli header files contengono i prototipi delle funzioni che formano ogni parte della libreria. In base alle funzioni che ci servono dobbiamo importare l'header opportuno all'interno del nostro file .cpp.

Vediamo un esempio in cui viene utilizzata una funzione della libreria standard detta *RANDOM* e che serve per la generazione di numeri casuali. Questa funzione viene utilizzata per calcolare un integrale al computer, tramite il cosiddetto metodo Montecarlo, o per studi statistici, come la simulazione di un lancio di dadi o di una monetina. In realtà è una funzione pseudo-casuale, nel senso che genera una sequenza casuale di valori tra 0 ed un limite massimo, ma in realtà ogni volta che la richiamo genera la stessa sequenza casuale. Questo a meno che non si specifichi un seme iniziale casuale che verrà selezionato da un'altra funzione contenuta nella libreria standard che è *srand*. Se usassi *srand=10* in un programma allora si genererebbe una sequenza casuale di valori che sarà sempre la stessa a partire dal seme pari a 10. Ciò per preservare la riproducibilità del programma. Bisogna quindi dare un seme “casuale” e non deterministico per poter avere sequenze diverse di esecuzione del programma, come ad esempio l'istante di tempo di esecuzione ottenibile tramite *time*. La funzione *time* serve per far variare eventualmente il comportamento di un programma in base al momento in cui lo si esegue.

C'è poi un tipo importante che è ENUM: è un tipo di dato che può assumere un numero finito e limitato di valori e ci si riferisce nel C++ alle CLASSI ENUMERATIVE. Vediamo un esempio in cui si fa uso anche di questo, ed è l'esempio di craps, un gioco statunitense in cui si fa uso di 2 dadi.

```

1 // Craps.cpp
2 // Craps simulation.
3 #include <iostream>
4 #include <cstdlib> // contains prototypes for functions srand and rand
5 #include <ctime> // contains prototype for function time
6 using std::cout;
7 using std::endl;
8 unsigned int rollDice(); // rolls dice, calculates and displays sum
9
10 int main() {
11     // scoped enumeration with constants that represent the game status
12     enum class Status {CONTINUE, WON, LOST}; // all caps in constants
13
14     // randomize random number generator using current time
15     srand(static_cast<unsigned int>(time(0)));
16
17     unsigned int myPoint{0}; // point if no win or loss on first roll
18     Status gameStatus; // can be CONTINUE, WON or LOST
19     unsigned int sumOfDice{rollDice()}; // first roll of the dice
20
21     // determine game status and point (if needed) based on first roll
22     switch (sumOfDice) {
23         case 7: // win with 7 on first roll
24         case 11: // win with 11 on first roll
25             gameStatus = Status::WON;
26             break;
27         case 2: // lose with 2 on first roll
28         case 3: // lose with 3 on first roll
29         case 12: // lose with 12 on first roll
30             gameStatus = Status::LOST;
31             break;
32         default: // did not win or lose, so remember point
33             gameStatus = Status::CONTINUE; // game is not over
34             myPoint = sumOfDice; // remember the point
35             cout << "Point is " << myPoint << endl;
36             break; // optional at end of switch
37     }
38
39     // while game is not complete
40     while (Status::CONTINUE == gameStatus) { // not WON or LOST
41         sumOfDice = rollDice(); // roll dice again
42
43         // determine game status
44         if (sumOfDice == myPoint) { // win by making point
45             gameStatus = Status::WON;
46         }
47         else {
48             if (sumOfDice == 7) { // lose by rolling 7 before point
49                 gameStatus = Status::LOST;
50             }
51     }
52 }

```

Vediamo che alla riga 12 c'è l'utilizzo di enum class: definisco una classe di tipo enum e di nome Status che può assumere solo 3 valori costanti.

Le regole per il craps sono queste:

- Rules of Craps game
- A player rolls two dice. Each die has six faces.
- Faces contain 1, 2, 3, 4, 5 and 6 spots.
- After the dice have come to rest, the sum of the spots on the two upward faces is calculated.
- If the sum is 7 or 11 on the first roll, the player wins.
- If the sum is 2, 3 or 12 on the first roll (called “craps”), the player loses (i.e., the “house” wins).
- If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, then that sum becomes the player’s “point.”
- To win, continue rolling until you “make your point.”
- You lose by rolling a 7 before making the point.

Praticamente un giocatore ha 2 dadi con 6 facce: se lanciandoli una prima volta fa 7 o 11 vince, se fa 2,3 o 12 perde (\rightarrow vince il banco), se fa uno tra i numeri restanti allora il numero che esce diventa il punteggio da ottenere: lancia fin quando non ottiene quel numero a meno che prima non esca un 7, caso in cui il giocatore perde.

Le variabili di tipo *Status* possono assumere solo un valore tra i possibili elencati, scritti tipicamente in maiuscolo. Nel nostro caso allora la classe enumerativa *Status* può assumere 3 valori CONTINUE, WON, LOST, che saranno poi sostituiti da degli interi.

Per utilizzare le classi enumerativi si usa come sintassi, ad esempio, quella alla riga 25:

```
gameStatus = Status::WON;
```

dove *gameStatus* è una variabile di tipo *Status* precedentemente definita.

Quindi in generale si usa *variabile::nome_della_costante*.

Dalla riga 63 c'è poi un'altra funzione che utilizza a sua volta la funzione *rand*, di cui abbiamo già parlato; dobbiamo aggiungere però che il valore massimo che può restituire questa funzione è un intero molto grande che si chiama *RAND_MAX* contenuto nella libreria standard, e questo ci interessa ad esempio quando vogliamo rappresentare un numero in virgola mobile tra 0 ed 1: lo facciamo dividendo *RAND* per *RAND_MAX*. A noi interessa rappresentare un numero tra 1 e 6 e per questo abbiamo usato:

```
int die1{1 + rand() % 6}; // first die roll
```

Infatti *rand()* modulo 6 se ci pensiamo un attimo cos'è che fa? *Rand(1)* è un intero, quindi quando ne facciamo il “% 6” uscirà un numero compreso tra 0 e 5. Aggiungendo 1 si ottiene un numero casuale tra 1 e 6.

Nel programma poi abbiamo fatto uso dello *switch-case*, che permette di fare una selezione multipla utile nel nostro caso per poter valutare se il giocatore ha vinto, perso o deve continuare a lanciare.

Importante è l'uso di *static*. Vediamolo in un esempio sullo scope delle variabili:

```
1 // sco.cpp
2 // Scoping example.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7
8 void useLocal(); // function prototype
9 void useStaticLocal(); // function prototype
10 void useGlobal(); // function prototype
11
12 int x{1}; // global variable
13
14 int main() {
15     cout << "global x in main is " << x << endl;
16
17     int x{5}; // local variable to main
18
19     cout << "local x in main's outer scope is " << x << endl;
20
21     { // block starts a new scope
22         int x{7}; // hides both x in outer scope and global x
23
24         cout << "local x in main's inner scope is " << x << endl;
25     }
26
27     cout << "local x in main's outer scope is " << x << endl;
28
29     useLocal(); // useLocal has local x
30     useStaticLocal(); // useStaticLocal has static local x
31     useGlobal(); // useGlobal uses global x
32     useLocal(); // useLocal reinitializes its local x
33     useStaticLocal(); // static local x retains its prior value
34     useGlobal(); // global x also retains its prior value
35
36 }
37
38 cout << "\nlocal x in main is " << x << endl;
```

```
39 // useLocal reinitializes local variable x during each call
40 void useLocal() {
41     int x{25}; // initialized each time useLocal is called
42
43     cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44     ++x;
45     cout << "local x is " << x << " on exiting useLocal" << endl;
46 }
47
48 // useStaticLocal initializes static local variable x only the
49 // first time the function is called; value of x is saved
50 // between calls to this function
51 void useStaticLocal() {
52     static int x{50}; // initialized first time useStaticLocal is called
53     cout << "\nlocal static x is " << x << " on entering useStaticLocal"
54     << endl;
55     ++x;
56     cout << "local static x is " << x << " on exiting useStaticLocal"
57     << endl;
58 }
59
60 // useGlobal modifies global variable x during each call
61 void useGlobal() {
62     cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
63     x *= 10;
64     cout << "global x is " << x << " on exiting useGlobal" << endl;
65 }
66
67
68 }
```

Notiamo innanzitutto che abbiamo tre variabili di tipo intero di nome *x*, una inizializzata a 1, una a 5 ed una a 7. La prima è globale, la seconda è definita nel main e la terza è definita in un blocco interno al main. Quella più interna maschera tutte le altre, per cui alla riga 24 il cout sarà riferito proprio alla *x* del blocco interno e sarà stampato a video “7”.

Per quanto riguarda l'uso di *static*, lo troviamo alla riga 52. A che serve?

Una funzione io la posso invocare migliaia di volte ed ogni volta il Program Counter (PC) prende l'indirizzo della prima istruzione della funzione. Ovviamente all'interno della funzione io dichiaro delle variabili con cui lavorerò prima di restituire il controllo al programma chiamante. Nella funzione *useLocal()* notiamo che quello che facciamo non è altro che definire una variabile, incrementarla e stampare il risultato a video; fatto ciò restituiamo il controllo. Quando invoco di nuovo la stessa funzione ridefinisco una nuova variabile che di nuovo parte da 25, viene incrementata a 26 e poi dopo la stampa a video di questa variabile il controllo viene restituito nuovamente alla funzione chiamata. Ogni volta che invoco questa funzione viene allocata una casella di memoria per contenere la variabile di tipo intero *x* che sarà poi deallocata alla fine della chiamata e reistanziata alla prossima chiamata. Nella funzione *useStaticLocal()*, invece, viene dichiarata una variabile di tipo statico ed inizializzata a 50: questa variabile continua ad esistere anche dopo che la funzione finisce di fare il suo lavoro e quando si rientra nella funzione si troverà la variabile allo stato in cui era stata lasciata precedentemente, e quindi incrementata di volta in volta di 1, nel nostro caso.

INLINE FUNCTION

una funzione dichiarata come inline e il preprocessore automaticamente la riscrive nel momento in cui la funzione viene richiamata. Quindi non avviene il meccanismo dello scambio dei parametri ma semplicemente la funzione viene scritta nuovamente quando è chiamata. Alcuni compilatori fanno automaticamente un'operazione del genere se la valutano più conveniente.

ES:

```
1 // inlinecpp
2 // inline function that calculates the volume of a cube.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definition of inline function cube. Definition of function appears
9 // before function is called, so a function prototype is not required.
10 // First line of function definition serves as the prototype.
11 inline double cube(const double side) {
12     return side * side * side; // calculate cube
13 }
14
15 int main() {
16     double sideValue; // stores value entered by user
17     cout << "Enter the side length of your cube: ";
18     cin >> sideValue; // read value from user
19
20     // calculate cube of sideValue and display result
21     cout << "Volume of cube with side "
22         << sideValue << " is " << cube(sideValue) << endl;
23 }
24
```

Nell'esempio notiamo anche l'utilizzo di variabili *const*: questo sta a significare che le variabili in questione non possono essere modificate.

PASSAGGIO DEI PARAMETRI

Il passaggio dei parametri può avvenire per riferimento o per valore.

Quando passo un parametro per valore allora viene fatta una copia del valore del parametro e passata alla funzione chiamata. Questo vuol dire che la funzione non potrà cambiare il valore della variabile originale utilizzata per invocare la funzione.

Nel caso del passaggio per riferimento viene passato alla funzione un riferimento della variabile e quindi viene data alla funzione l'autorità a cambiare il suo valore. Quindi passo dei parametri per riferimento quando voglio che la funzione cambi il loro valore, ovvero quando voglio che la funzione abbia un “side effect”. Un parametro lo si passa per riferimento utilizzando l'operatore “*&*” che assume il ruolo di “riferimento” della variabile.

Es:

```
1 // <sstream>
2 // Passing arguments by value and by reference.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int squareByValue(int); // function prototype (value pass)
8 void squareByReference(int&); // function prototype (reference pass)
9
10 int main() {
11     int x{2}; // value to square using squareByValue
12     int z{4}; // value to square using squareByReference
13
14     // demonstrate squareByValue
15     cout << "x = " << x << " before squareByValue\n";
16     cout << "Value returned by squareByValue: "
17         << squareByValue(x) << endl;
18     cout << "x = " << x << " after squareByValue\n" << endl;
19
20     // demonstrate squareByReference
21     cout << "z = " << z << " before squareByReference" << endl;
22     squareByReference(z);
23     cout << "z = " << z << " after squareByReference" << endl;
24 }
25
26 // squareByValue multiplies number by itself, stores the
27 // result in number and returns the new value of number
28 int squareByValue(int number) {
29     return number *= number; // caller's argument not modified
30 }
31
32 // squareByReference multiplies numberRef by itself and stores the result
33 // in the variable to which numberRef refers in function main
34 void squareByReference(int& numberRef) {
35     numberRef *= numberRef; // caller's argument modified
36 }
```

Le referenze possono essere utilizzate anche come alias, ossia come sinonimi per altre variabili all'interno di una funzione. Ad esempio:

```
int count{1}; // declare integer variable count
int& cRef{count}; // create cRef as an alias for count
++cRef; // increment count (using its alias cRef)
```

In questo caso la variabile *count* viene inizializzata ad 1, la variabile *cRef* viene inizializzata a *count* ma viene dichiarata con l'operatore &: questo vuol dire che quando viene incrementata verrà incrementato anche *count*. Quindi si è utilizzato *cRef* come alias di *count*.

Il riferimento può essere molto utile quando vogliamo passare un argomento grande ad una funzione, come una stringa che rappresenta un testo lungo, senza che la funzione alteri la variabile e senza passare la variabile per valore, in modo da non fare una copia; allora in tal caso passo la variabile per riferimento ma in un modo particolare, ossia utilizzando *const* prima del parametro.

Es:

```
void setName(const std::string& accountName);
```

Instead of

```
void setName(std::string accountName);
```

or

```
const std::string& getName() const {...}
```

Instead of

```
std::string getName() const { ... };
```

indicates that the string should be returned by reference
(eliminating the overhead of copying a string) and that
the caller cannot modify the returned string

In questi 2 modi dell'esempio si vede appunto come sfruttare i vantaggi del riferimento evitando però side effects.

Esistono poi gli argomenti di default: se una funzione ha vari parametri, alcuni di questi possono non essere specificati e si possono dare dei valori di default. I valori di default possono essere sfruttati solo per le variabili più a destra nella lista dei parametri della funzione.

Interessante poi è l'utilizzo dell'operatore:

::

per fare riferimento alle variabili globali piuttosto che quelle locali.

Es:

```
1 // unary.cpp
2 // Unary scope resolution operator.
3 #include <iostream>
4 using namespace std;
5
6 int number{7}; // global variable named number
7
8 int main() {
9     double number{10.5}; // local variable named number
10
11    // display values of local and global variables
12    cout << "Local double value of number = " << number
13    << "\nGlobal int value of number = " << ::number << endl;
14 }
```

In questo esempio vediamo che ci sono 2 variabili con stesso nome, una globale ed una nel main. Se nel main stampiamo a video la variabile ovviamente quella stampata a video sarà la variabile definita nel main. Antecedendo l'operatore :: alla variabile, però, si può stampare a video la variabile globale.

FUNCTION OVERLOADING

In realtà esiste anche l'overloading delle operazioni: il +, ad esempio, assume significati differenti in base al contesto in cui lo si utilizza (intero, stringa, etc).

Stessa cosa può avvenire per le funzioni: possiamo definire due funzioni con lo stesso nome ma con signature diversa. Si sceglie a questo punto quale delle 2 funzioni invocare in base al tipo di parametro con cui avviene l'invocazione.

Es:

```
1 // overloading.cpp
2 // Overloaded square functions.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // function square for int values
8 int square(int x) {
9     cout << "square of integer " << x << " is ";
10    return x * x;
11}
12
13 // function square for double values
14 double square(double y) {
15     cout << "square of double " << y << " is ";
16     return y * y;
17}
18
19 int main() {
20     cout << square(7); // calls int version
21     cout << endl;
22     cout << square(7.5); // calls double version
23     cout << endl;
24 }
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

La funzione *square* è definita in 2 modi diversi, ovvero come due funzioni: una prende in ingresso un double e restituisce un double, l'altra prende in ingresso un int e restituisce un int. Scelgo quale funzione invocare in base ai parametri con cui invoco la funzione. Questo è l'overloading delle funzioni.

FUNCTION TEMPLATES

Questo è importante e ci fa capire la potenza del C++. Il template è un modo per effettuare l'overloading in maniera più compatta; permette di scrivere un'unica funzione con tante firme. I template sono funzioni speciali che possono operare con tipi generici e questo ci permette di creare un unico template le cui funzionalità possono essere adattate a più di un tipo o di una classe. Ad esempio, se devo calcolare il massimo tra 2 interi ed il massimo tra 2 reali, invece che creare 2 funzioni posso utilizzare un template che ha un doppio comportamento. E quindi si capisce che un template altro non è che uno scheletro che viene riscritto dinamicamente in base a delle parole chiave, in questo caso in base al nome di un tipo.

```

//maximum of two int
int max(int x, int y) {
    return (x > y)? x: y;
}

//maximum of two float
float max(float x, float y) {
    return(x > y)? x:y;
}

//template for the maximum
template <typename T>
T max(T x, T y) {
    return(x > y)? x:y;
}

```

Syntax:

```

template <class identifier> function_declaration;
template <typename identifier> function_declaration;

```

Il C++, dati i tipi degli argomenti forniti nella chiamata alla funzione, genera automaticamente delle funzioni separate e specializzate al tipo con cui viene chiamata la funzione. Ciascun parametro (nel nostro esempio ce n'è solo uno) viene anteceduto dalla parola chiave *typename* o dalla parola *class* (lo vedremo poi meglio). Vediamo lo stesso esempio ma più specifico e con passaggio per riferimento, cosa che conviene sempre fare quando si passano classi vere e proprie.

```

1 // max.cpp
2 //Use of templates
3 #include <iostream>
4 using std::cout;
5
6 template <typename T>
7 const T& max(const T& x, const T& y) {
8     return(x > y)? x:y;
9 }
10
11 int main()
12 {
13     int i = max(3, 7); // returns 7
14     std::cout << i << '\n';
15
16     double d = max(5.34, 18.623); // returns 18.623
17     std::cout << d << '\n';
18
19     char ch = max('a', 'b'); // returns 'a'
20     std::cout << ch << '\n';
21
22     return 0;
23 }
24

```

STACK

I meccanismi per la chiamata di funzioni nel C++ sono basati sulla struttura dati stack.

Quando chiamiamo una funzione:

1. il sistema deve sapere dove restituire il controllo alla fine della funzione;
2. i parametri sono “passati” alla funzione;
3. i valori di ritorno sono ottenuti dalla funzione chiamante.

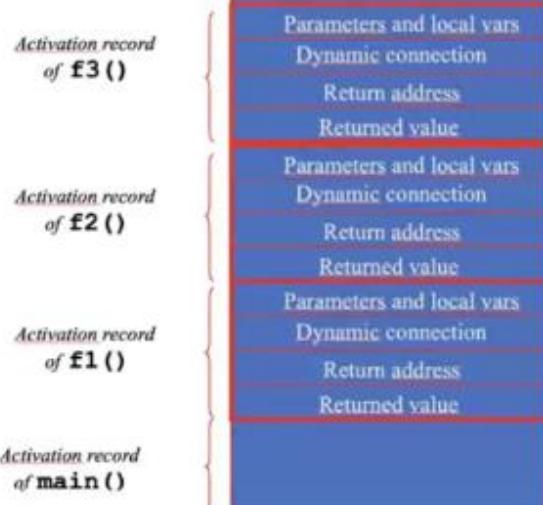
Ogni volta che una funzione chiama un’altra funzione, viene inserito sopra lo stack (push) un entry point, detto anche “stack frame” o “activation record” (record di attivazione) e che contiene l’indirizzo di ritorno di cui la funzione chiamata necessita per ridare il controllo alla funzione chiamante. Quando una funzione chiamata finisce il suo lavoro viene prima deallocated lo stack (pop) e poi ceduto il controllo alla funzione chiamante, sfruttando il “return address” scritto sempre nello stack.

The activation record typically contains:

- The parameters of the function
- Local variable
 - May be stored somewhere else, but the record contains descriptions and pointers to their locations
- Return address
- Dynamic connection
 - Reference to the activation record of the calling function
- Returned value
 - Unless the function returns a void

The activation records are dynamically stored in a Stack data structure

main() → f1() → f2() → f3()



Quindi come prima cosa alla chiamata di una funzione viene salvato il record di attivazione nello stack, che contiene le informazioni necessarie per eseguire la funzione e per restituire il controllo alla funzione chiamante, ovvero in ordine:

- parametri con cui è stata invocata la funzione;
- variabili locali definite all’interno della funzione;
- dynamic connection, ovvero un link all’activation record della funzione chiamante;
- return address;
- valore di ritorno (eventuale) restituito.

Ciò che caratterizza la chiamata di una funzione allora è il record di attivazione associato a quella chiamata, e questo permette in effetti che una funzione possa chiamare se stessa con parametri differenti. In tal caso si parla di **chiamata ricorsiva**.

Nel linguaggio comune cerchiamo di evitare le definizioni ricorsive, mentre nell'informatica le apprezziamo. Vediamo un esempio di ricorsione con le liste. Una lista è una cosa fatta da:

- un elemento
- oppure da un elemento e una lista.

Es:

(24, 88, 40, 37)

```
element  comma  LIST
24      ,      (88, 40, 37)

          element  comma  LIST
          88      ,      (40, 37)

          element  comma  LIST
          40      ,      (37)

          element
          37
```

Le definizioni ricorsive sono costituite da:

- una parte non ricorsiva chiamata base case (caso base);
- un'altra parte, ricorsiva, detta recursive base (caso ricorsivo).

Il caso base, non ricorsivo, ritorna un risultato ed è necessario perché altrimenti la definizione potrebbe essere applicata infinitamente, invece c'è bisogno di una terminazione.

Es: FATTORIALE

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n - 1)! & \text{Recursive case} \end{cases}$$

LEZIONE 9

Continuiamo con le funzioni ricorsive.

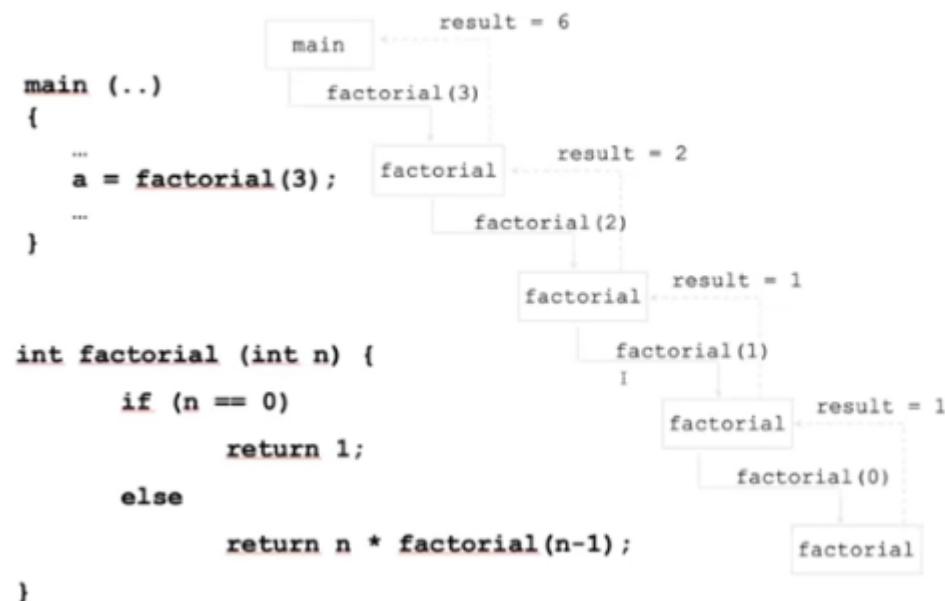
RICORSIONE

Abbiamo detto che un esempio possibile è il fattoriale; vediamone un esempio di programma:

```
1 // factorial.cpp
2 // Recursive function factorial.
3 #include <iostream>
4 #include <iomanip>
5 using std::cout;
6 using std::endl;
7 using std::setw;
8
9 unsigned long factorial(unsigned long); // function prototype
10
11 int main() {
12     // calculate the factorials of 0 through 10
13     for (unsigned int counter{0}; counter <= 10; ++counter) {
14         cout << setw(2) << counter << "!" = " << factorial(counter)
15             << endl;
16     }
17 }
18
19 // recursive definition of function factorial
20 unsigned long factorial(unsigned long number) {
21     if (number <= 1) { // test for base case
22         return 1; // base cases: 0! = 1 and 1! = 1
23     }
24     else { // recursion step
25         return number * factorial(number - 1);
26     }
27 }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
```

La catena che possiamo immaginare è questa:



Ovvero: se chiamiamo la funzione factorial(3), questa chiamata scatenerà la chiamata di factorial(2) che scatenerà a sua volta la chiamata di factorial(1) che scatenerà la chiamata di factorial(0).

In generale ogni volta che abbiamo un procedimento iterativo possiamo utilizzare funzioni ricorsive.

Es: potenza n-esima di x

$$\text{n-th power law function } x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{otherwise} \end{cases}$$

```
double power (double x, int n) {
    if (n == 0)
        return 1.0;
    else
        return x * power(x,n-1)
}

call 1  power (x, 4)
call 2  power (x, 3)
call 3  power (x, 2)
call 4  power (x, 1)
call 5  power (x, 0)
call 5      1
call 4      x
call 3      x*x
call 2      x*x*x
call 1      x*x*x*x
```

Il ragionamento è sempre lo stesso: si “apre” il problema a partire da $\text{power}(x, 4)$ e, una volta risolto giungendo al caso base, si procede a ritroso. Vediamo bene cosa succede anche rispetto allo stack.

Let's numberate the lines of **power**

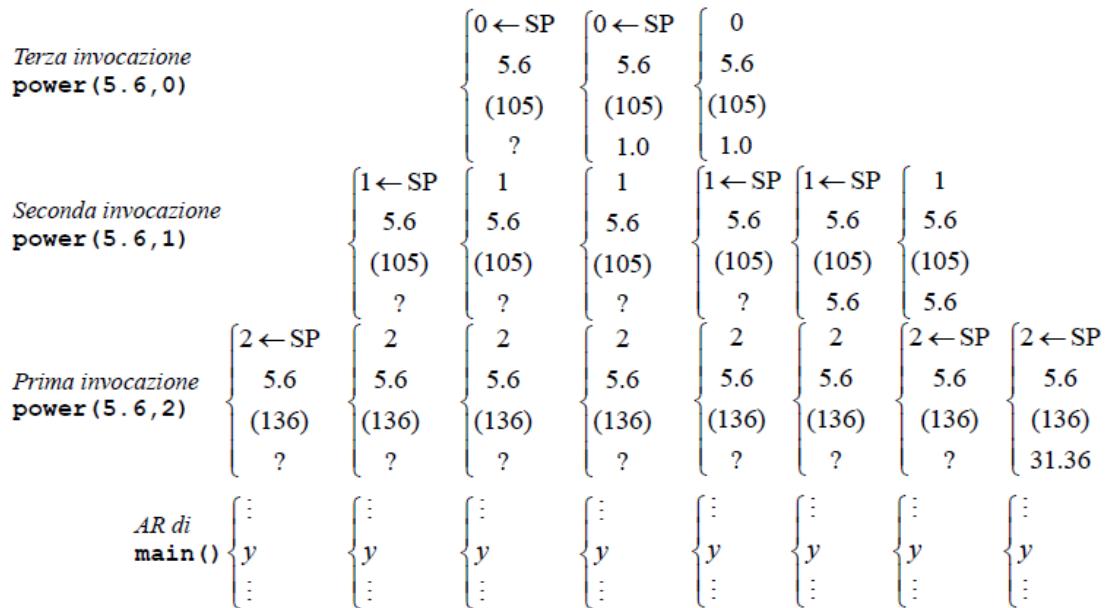
```
double power (double x, int n) {
/*102*/     if (n == 0)
/*103*/         return 1.0;
/*104*/     else
/*105*/         return x * power(x,n-1)
}
```

Being called by main():

```
public static void main(String[] args) {
    ...
/*136*/    y = power(5.6, 2);
    ...
}

invocazione 1  power (5.6, 2)
invocazione 2  power (5.6, 1)
invocazione 3  power (5.6, 0)
invocazione 3      1
invocazione 2      5.6
invocazione 1      31.36
```

Nell'esempio di sopra abbiamo supposto (in maniera fittizia) che le istruzioni della funzione **power** assumano posizioni in memoria consecutive, da 102 a 105. Cosa succede alla chiamata della funzione per calcolare il quadrato di 5.6, istruzione posta all'indirizzo 136? Succede questo:



Alla prima invocazione viene creato un record di attivazione con valore di ritorno sconosciuto, indirizzo di ritorno 136 (ultima istruzione eseguita dalla funzione chiamante prima della chiamata) e parametri della funzione pari a 5.6 e 2. Lo stackpointer SP punterà all'ultimo parametro d'ingresso della funzione, quindi 2.

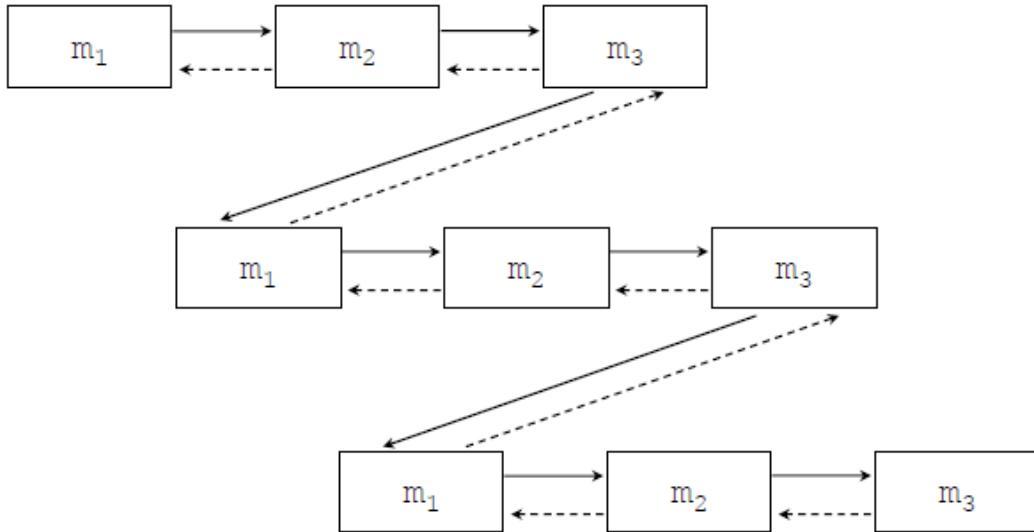
A questo punto questa funzione invoca sé stessa istanziando un secondo record di attivazione che però avrà indirizzo di ritorno differente, ovvero la funzione stessa (105) e parametri d'ingresso differenti perché questa funzione calcolerà non 5.6^2 ma 5.6^1 ; lo stackpointer SP punterà all'ultimo parametro d'ingresso di questa funzione, e quindi 1. Il valore di ritorno è ancora sconosciuto.

Si fa lo stesso con una terza invocazione e poi si procede a ritroso con varie operazioni di pop sullo stack (le precedenti erano di push), deallocando lo stack; tuttavia stavolta i valori di ritorno saranno ben noti.

Ci sono altri tipi di ricorsione, ad esempio:

RICORSIONE INDIRETTA

È più pericolosa di quella diretta. Quella diretta funzionava che una funzione chiama se stessa, quella indiretta invece funziona così: una funzione chiama un'altra funzione che chiama un'altra funzione che chiama l'originale.

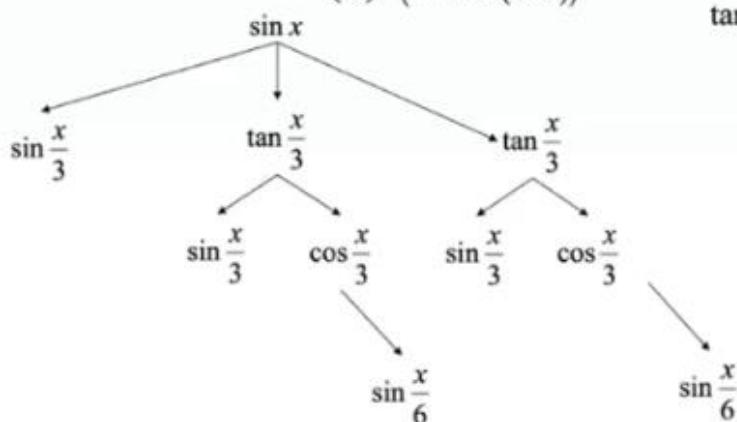


La ricorsione indiretta è più pericolosa e difficile da gestire perché non abbiamo la garanzia il caso base per qualsiasi chiamata. Per questo va utilizzata con cautela.

Esempio: calcolo di una roba brutta come $\sin() * f(\tan())$; infatti $\tan(-) = [\sin(-)/\cos(-)]$ e il coseno lo esprimiamo in funzione del seno. Il caso base è lo sviluppo di Taylor arrestato, ad esempio, al secondo termine che dovrebbe essere in realtà $x^3/3$ (esempio sbagliato).

- **Base case** $\sin(x) = x - \frac{x^2}{6}$, se $|x| < \epsilon$

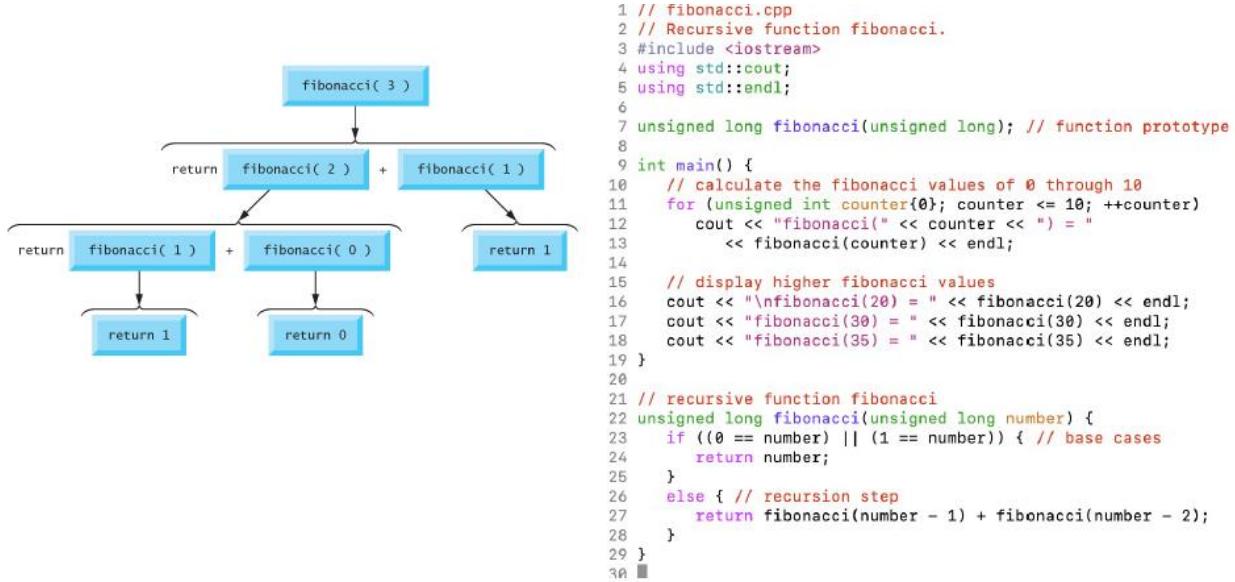
- **Recursive case:** $\sin(x) = \sin\left(\frac{x}{3}\right) \cdot \frac{(3 - \tan^2(x/3))}{(1 + \tan^2(x/3))}$ $\cos(x) = 1 - \sin(x/2)$
 $\tan(x) = \frac{\sin(x)}{\cos(x)}$



Facciamo qualche altro esempio di ricorsione. In particolare vediamo un limite della ricorsione, che è il:

SUBPROBLEM OVERLAP (sovraposizione di sottoproblemi)

Es: Fibonacci



Fare tutto in maniera ricorsiva è rischioso, poiché si può incorrere nel cosiddetto subproblem overlap (sovraposizione dei sottoproblemi). Nell'esempio di Fibonacci vediamo che $\text{fibonacci}(1)$ è invocato sia da $\text{fibonacci}(3)$ che da $\text{fibonacci}(2)$, quindi sto risolvendo lo stesso problema con 2 chiamate. In generale si può risolvere lo stesso problema con n chiamate alla stessa funzione, in maniera del tutto inutile.

Cosa conviene di più tra procedimento iterativo e procedimento ricorsivo nel caso della sequenza di Fibonacci?

```

int iterativeFib (int n){
    if (n < 2)
        return n;
    else
        int i = 2, tmp, current = 1, last = 0;
        for (; i <= n; ++i) {
            tmp = current;
            current += last;
            last = temp;
        }
}

```

Nel caso iterativo c'è un unico ciclo for, quindi il numero di operazioni è del numero di n , o meglio proporzionale ad $n \rightarrow$ num. di operazioni = $O(n)$.

Nel caso ricorsivo il costo è dell'ordine di $2^n \rightarrow$ num. di operazioni=O(2^n). Infatti la prima invocazione genera altre 2 invocazioni, alla seconda ogni invocazione ne genera altre 2 e così via.

Nello specifico:

n	$O(2^n)$	Numero di Assegnazioni	$O(n)$
	Ricorsivo	Iterativo	
6	25		15
10	177		27
15	1973		42
20	21891		57
25	242795		72
26	2692537		87

Vediamo un altro esempio di ricorsione.

Esempio: conversione da decimale a binario; l'avevamo visto in maniera iterativa, ora lo facciamo in maniera ricorsiva.

```

1#include <iostream>
2
3using std::cin;
4using std::cout;
5
6
7std::string recDec2Bin(int number){
8    if (number == 0) {
9        return("0");
10   }
11   else{
12       return recDec2Bin(number/2) + std::to_string(number%2);
13   }
14}
15
16int main(){
17    int number;
18    cout << "Enter an integer: " ;
19    cin >> number;
20    cout << "The binary representation of " << number << " is :";
21    cout << recDec2Bin(number) << "\n";
22}
~~

```

Cosa fa questo programma? Concatena man mano delle stringhe ad ogni passo ricorsivo. Nello specifico ad ogni passo ricorsivo viene calcolato il resto modulo 2 del numero e concatenato alla parte restante ancora da calcolare sulla metà del numero. Infatti se vogliamo convertire 11 quello che si fa è:

Numero	 	%2
11		1
5		1
2		0
1		1
0		0

11 in binario è 1011, ovvero la sequenza di numeri nella colonna di destra, ovvero la sequenza dei resti %2, letta al contrario. Quindi il primo %2 sarà quello relativo alla parte meno significativa del numero. Poi dividiamo il numero per 2 e ricalcoliamo il %2 su questo numero: quello che esce sarà il secondo bit da destra. E così via, fino al bit più significativo.

Ma possiamo sfruttare la ricorsione per molte altre cose, ad esempio per scoprire se una stringa è palindroma o meno: dobbiamo capire se l'ultimo elemento della stringa è uguale al primo; in tal caso facciamo lo stesso controllo sulla seconda lettera e sulla penultima e così via, fino ad ottenere una stringa vuota.

```

1 #include <iostream>
2 #include <string>
3
4 using std::string;
5 using std::cout;
6 using std::cin;
7
8
9 bool recPalindrome(string word){
10    if(word.length() <=1) {
11        return true;
12    }
13    else {
14        return word.at(0) == word.at(word.length()-1) && recPalindrome(word.substr(1,word.length()-2));
15    }
16}
17
18
19
20 int main (){
21    std::string word;
22    cout << "Enter a word: ";
23    cin >> word;
24    if (recPalindrome(word)){
25        cout << word << " IS palindrome\n";
26    }
27    else {
28        cout << word << " is NOT palindrome\n";
29    }
30    return 0;
31 }
```

LCS (Longest Common Subsequence)

È un algoritmo che risolve uno dei problemi più comuni della Computer Science riguardo il confronto tra stringhe, ovvero quello di trovare la sottosequenza più lunga comune a 2 stringhe. Una sottosequenza è diversa da una sottostringa, poiché una sottosequenza è un insieme di caratteri che appaiono nello stesso ordine rispetto alla stringa originale ma che non sono necessariamente contigui. Quindi una sottosequenza è una sequenza di caratteri che si ottiene dalla sequenza originale eliminando alcuni dei caratteri.

Es: ALTO è una sottosequenza di ALBERTO

Risolvere ricorsivamente il problema della sottosequenza comune più lunga ha come rischio quello di incorrere in una sovrapposizione dei sottoproblemi. Ovviamente tutte le stringhe hanno in comune almeno la sottosequenza vuota. Noi siamo interessati alla lunghezza della sottosequenza più lunga. Ad esempio qual è la sequenza più lunga comune alle 2 stringhe di sotto? (rappresentano sequenze di DNA)

ATACGGT	TATTCT
Sequence of length 7	Sequence of length 6

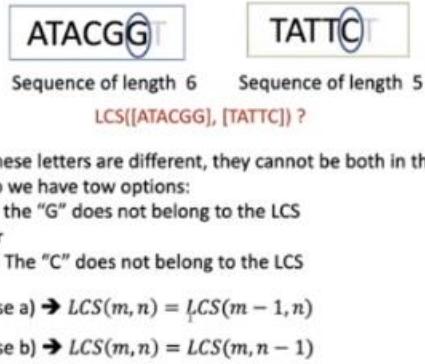
LCS([ATACGGT], [TATTCT]) ?

Cominciamo confrontando gli ultimi caratteri. Visto che sono uguali faranno sicuramente parte della LCS e quindi la lunghezza della LCS sarà sicuramente 1.

$$\boxed{\text{LCS}([\text{ATACGGT}], [\text{TATTCT}]) = 1 + \text{LCS}([\text{ATACGG}], [\text{TATTC}])}$$
$$\text{LCS}(m, n) = 1 + \text{LCS}(m - 1, n - 1)$$

Quindi dobbiamo risolvere lo stesso problema ma su delle sottostringhe, ovvero dobbiamo risolvere sottoproblemi del problema di partenza.

Procedendo ancora, vediamo che le lettere G e C sono diverse → non possono essere contemporaneamente all'interno della LCS, quindi o elimino la G o la C dalla LCS; queste sono le 2 opzioni.



In un caso dobbiamo calcolare l'LCS su 2 stringhe di lunghezza, rispettivamente, $m-1$ ed n ; nell'altro caso su stringhe di lunghezza $m, n-1$. Quale delle 2 funzioni di ricorrenza dobbiamo scegliere? Supponiamo di conoscere le 2 LCS; beh, a questo punto sceglieremmo la sottosequenza più lunga e quindi in generale sceglioamo:

choose: $\max\{LCS(m - 1, n), LCS(m, n - 1)\}$

L'idea è sempre quella di ricondursi a problemi ricorsivi sempre più piccoli.

Il caso base sarà:

$$LCS(m, 0) = 0 \text{ and } LCS(0, n) = 0$$

Ovvero procedendo diminuendo la lunghezza delle sequenze prima o poi arriveremo a calcolare la LCS tra 2 stringhe di cui una è vuota e che quindi avrà lunghezza nulla.

Allora, riassumendo:

$$LCS(m, n) = \begin{cases} 1 + LCS(m - 1, n - 1) & \text{if } x_m = y_n \\ \max\{LCS(m - 1, n), LCS(m, n - 1)\} & \end{cases}$$

$$LCS(m, 0) = 0, LCS(0, n) = 0$$

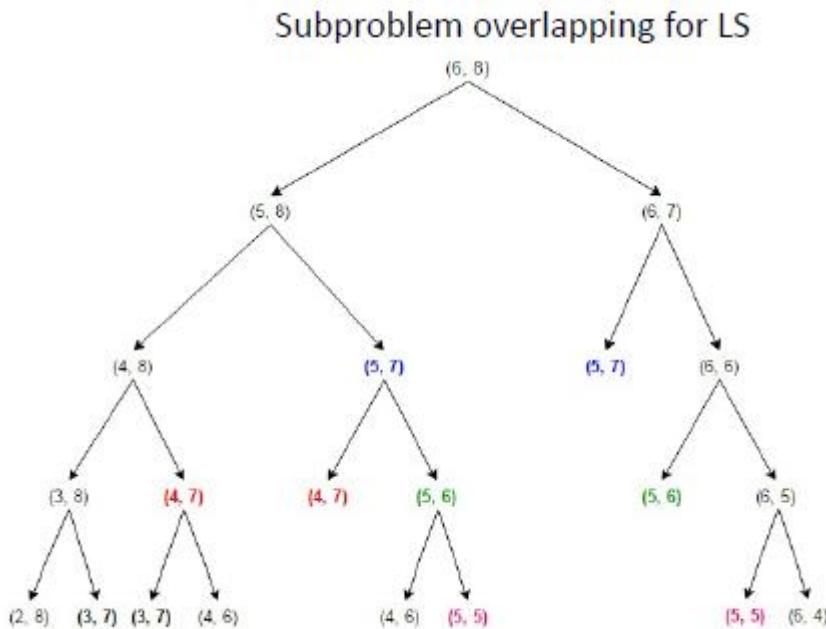
Il problema della LCS, come moltissimi problemi ricorsivi, manifesta 2 caratteristiche:

- sottostruttura ottima;
- subproblem overlap.

Queste caratteristiche riguardano in generale la programmazione dinamica.

La prima vuol dire che la soluzione del problema è ottenuta trovando e componendo soluzioni ottime a sottoproblemi dello stesso tipo. In generale il problema del percorso ottimale presenta la caratteristica della sottostruttura ottima; il percorso ottimale è il percorso che, dato un grafo in cui ci sono dei punti e delle connessioni, collega 2 punti in maniera ottimale; ad esempio può essere quello a lunghezza minima. Se voglio andare da A a C e voglio passare per B allora il percorso ottimale sarà la composizione del percorso ottimale tra A e B e quello tra B e C.

La seconda caratteristica è quella che caratterizza anche il problema di Fibonacci ed è la sovrapposizione dei sottoproblemi (subproblem overlap). Ad esempio se vogliamo calcolare LCS(6,8):



Vediamo che viene calcolata 2 volte LCS(5,7) e stesso vale per (4,7), (5,6), (5,5) e quindi viene invocata 2 volte la stessa funzione per il calcolo della LCS di tutte queste coppie. Quindi il numero di chiamate esplode nel caso peggiore (worst case) e quindi $LCS=O(2^n)$, esattamente come per Fibonacci.

A destra vediamo come si realizza il programma che trova la lunghezza della LCS. In realtà questo programma non è molto ottimizzato, sia per complessità temporale in quanto abbiamo detto che il caso peggiore è dell'ordine di 2^n per via del subproblem overlapping, sia per complessità spaziale poiché passiamo le stringhe X ed Y per valore, ed X e Y potrebbero anche contenere 1000 caratteri. Quindi potremmo in realtà passare per riferimento le stringhe ed antecedere ad esse *const* se non vogliamo alterarle.

```

1 //LCS.cpp
2 //computes the Longest common subsequence recursively
3
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 template <typename T>
8
9 //template for the max function
10 const T& max(const T& x, const T& y) {
11     return(x > y)? x:y;
12 }
13
14
15 //computes the lcs
16 int lcs(std::string X, std::string Y,int m,int n)
17 {
18
19     if (m==0|| n==0 )
20         return 0;
21     if (X[m - 1] == Y[n - 1])
22         return 1 + lcs(X, Y, m - 1, n - 1);
23     else
24         return max(lcs(X, Y, m, n - 1), lcs(X, Y, m - 1, n));
25 }
26
27
28 //Test the lcs function
29 int main()
30 {
31     std::string X{"ACGTCTGATC"};
32     std::string Y{"CTGTGTAGTA"};
33
34     cout << "Length of LCS is "<< lcs(X, Y, X.length(), Y.length()) << endl;
35     return 0;
36 }
37
38

```

Cerchiamo però di sfruttare lo svantaggio della complessità di calcolo facendo un ragionamento, ovvero piuttosto che usare la ricorsione possiamo definire ricorsivamente il valore della soluzione ma calcolare in maniera iterativa, attraverso una struttura dati apposita (multidimensionale), i valori della LCS, poiché spesso ci interessa non solo sapere la lunghezza della sottosequenza più lunga ma anche conoscere proprio la sottosequenza.

S = ATACCAGA (length $m = 8$)

T = CTCCTAG (length $n = 7$)

1. Build a table of $8+1=9$ row and $7+1=8$ columns

2. Initialize the first row and the first column to 0, :

$$LCS[m, 0] = 0$$

$$LCS[0, n] = 0$$

	*	0	1	2	3	4	5	6	7	*
*	-	0	0	0	0	0	0	0	0	T
0	-	0	0	0	0	0	0	0	0	
1	A	0								
2	T	0								
3	A	0								
4	C	0								
5	C	0								
6	A	0								
7	G	0								
8	A	0								

$$LCS(m, n) = \begin{cases} 1 + LCS(m - 1, n - 1) & \text{if } s_m = t_n \\ \max\{LCS(m - 1, n), LCS(m, n - 1)\} & \end{cases}$$

$$LCS(m, 0) = 0, LCS(0, n) = 0$$

In questa tabella inserisco tutti i valori della LCS per tutti i possibili valori di m ed n . La soluzione al problema per $(m,n)=(4,4)$, ad esempio, vedo che dipende dalla soluzione al problema per $(3,3)$, $(3,4)$, $(4,3)$. In generale ogni casella dipende da quella a sinistra, quella sopra e quella sopra a sinistra. Allora avendo i valori del caso base si può partire dalla casella $(1,1)$. Riempita questa si riempie la casella $(1,2)$, e così via.

**Longest Common Subsequence (LCS)
Example**

	*	0	1	2	3	4	5	6	7	*
*	-	0	0	0	0	0	0	0	0	T
0	-	0	0	0	0	0	0	0	0	
1	A	0	0	0	0	0	0	0	0	
2	T	0								
3	A	0								
4	C	0								
5	C	0								
6	A	0								
7	G	0								
8	A	0								

$$LCS[m, n] = \begin{cases} 1 + LCS[m - 1, n - 1] & s_m = t_n \\ \max\{LCS[m - 1, n], LCS[m, n - 1]\} & s_m \neq t_n \end{cases}$$

Quindi sfrutto inizialmente la ricorsione per arrivare al caso base, e poi calcolo i valori della LCS in maniera iterativa riempendo tutte le tabelle, fino ad arrivare al valore che mi serve che è la LCS(8,7). Mi muovo quindi prima all'indietro in maniera ricorsiva e sfrutto poi il subproblem overlap, che di per sé sarebbe negativo, per muovermi in avanti, così da trovare i valori della LCS.

	-	C	T	C	C	X	A	G
-	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	1	1
T	0	0	1	1	1	1	1	1
C	0	1	1	2	2	2	2	2
C	0	1	1	2	3	3	3	3
A	0	1	1	2	3	3	4	4
G	0	1	1	2	3	3	4	5
X	0	1	1	2	3	3	4	5

The value of the solution is in position
[n,m]

We can obtain the LCS sequence by
backtracing the sequence of choices

In our case the length of the LCS is 5
and the LCS is:

TCCAG

la lunghezza della LCS tra S e T = 5

$$LCS(m, n) = \begin{cases} 1 + LCS(m - 1, n - 1) & \text{if } s_m = t_n \\ \max\{LCS(m - 1, n), LCS(m, n - 1)\} & \end{cases}$$

$$LCS(m, 0) = 0, LCS(0, n) = 0$$

E quindi scopriamo che la sequenza della LCS è TCCAG.

LEZIONE 10

VETTORI, ARRAY & PUNTATORI

I vettori sono delle strutture dati che ci permettono di rappresentare una collezione di valori tutti dello stesso tipo con un nome ed un indice, proprio come gli array che abbiamo visto a fondamenti di informatica, ma che possono aumentare e diminuire di dimensione → non hanno dimensione costante.

ARRAY

Un array è un gruppo contiguo di locazioni di memoria che sono tutte dello stesso tipo. Per riferirci ad una locazione specifica bisogna specificare l'indice che identifica quella locazione, ovvero la distanza dalla prima locazione, quella numero 0, ad essa.

Noi abbiamo visto già a fondamenti di informatica quelli che si chiamano built-in-arrays, dove c'è una stretta corrispondenza tra puntatori ed array.

In C++ è stato introdotto un altro tipo strutturato che si chiama proprio **array** e una variabile di tipo array viene dichiarata proprio così:

```
array <typetype, arraySize > arrayName;
```

In realtà “array” non è proprio una parola chiave ma è un template definito nell’header <array>:

Defined in header <array>

```
template<
    class T,
    std::size_t N
    > struct array;
```

Quindi il tipo array è una struttura dati complessa derivata dai tipi da noi definiti, ovvero le classi, e che ci permette di identificare un insieme di elementi tutti dello stesso tipo utilizzando un nome ed un indice che identifica la posizione e che è quindi un intero. Gli array hanno dimensione fissa ed infatti al momento della dichiarazione va specificata la dimensione.

Sulle variabili di tipo array sono definite alcune funzioni membro:

Iterators	
<code>begin</code>	Return iterator to beginning (public member function)
<code>end</code>	Return iterator to end (public member function)
<code>rbegin</code>	Return reverse iterator to reverse beginning (public member function)
<code>rend</code>	Return reverse iterator to reverse end (public member function)
<code>cbegin</code>	Return const_iterator to beginning (public member function)
<code>cbegin</code>	Return const_iterator to end (public member function)
<code>crend</code>	Return const_reverse_iterator to reverse beginning (public member function)
<code>crend</code>	Return const_reverse_iterator to reverse end (public member function)

Capacity	
<code>size</code>	Return size (public member function)
<code>max_size</code>	Return maximum size (public member function)
<code>empty</code>	Test whether array is empty (public member function)

Element access	
<code>operator[]</code>	Access element (public member function)
<code>at</code>	Access element (public member function)
<code>front</code>	Access first element (public member function)
<code>back</code>	Access last element (public member function)
<code>data</code>	Get pointer to data (public member function)

Modifiers	
<code>fill</code>	Fill array with value (public member function)
<code>swap</code>	Swap content (public member function)

Gli iteratori altro non sono che dei contenitori che ci permettono di scorrere gli elementi dell'array.

Vediamo un programma di esempio sugli array:

```
1// Array2.cpp
2// Initializing an array's elements to zeros and printing the array.
3#include <iostream>
4#include <iomanip>
5#include <array>
6using std::cout;
7using std::array;
8using std::endl;
9using std::setw;
10
11int main() {
12    array<int, 5> n; // n is an array of 5 int values
13
14    // initialize elements of array n to 0
15    for (size_t i{0}; i < n.size(); ++i) {
16        n[i] = 0; // set element at location i to 0
17    }
18
19    cout << "Element" << setw(10) << "Value" << endl;
20
21    // output each array element's value
22    for (size_t j{0}; j < n.size(); ++j) {
23        cout << setw(7) << j << setw(10) << n[j] << endl;
24    }
25}
```

Per default se non si mette alcuna lista di inizializzazione allora l'array conterrà valori non prevedibili. Se invece si mette una lista di inizializzazione ma non si inizializzano tutti gli elementi, allora quelli non specificati saranno posti a 0 di default.

Vediamo un esempio.

Es: lancio di un dado

```
1 // Die.cpp
2 // Die-rolling program using an array instead of switch.
3 #include <iostream>
4 #include <iomanip>
5 #include <array>
6 #include <ctime>
7 #include <cstdlib>
8 using std::cout;
9 using std::array;
10 using std::setw;
11 using std::endl;
12 int main() {
13     // use the default random-number generation engine to
14     // produce uniformly distributed pseudorandom int values from 1 to 6
15     //default_random_engine engine[static_cast<unsigned int>(time(0))];
16     //uniform_int_distribution<unsigned int> randomInt(1, 6);
17
18     const size_t arraySize{6}; // ignore element zero
19     array<unsigned int, arraySize> frequency{}; // initialize to 0s
20
21     // roll die 60,000,000 times; use die value as frequency index
22     for (unsigned int roll{1}; roll <= 60000000; ++roll) {
23         ++frequency[rand() % 6];
24     }
25
26     cout << "Face" << setw(13) << "Frequency" << endl;
27
28     // output each array element's value
29     for (size_t face{0}; face < frequency.size(); ++face) {
30         cout << setw(4) << face+1 << setw(13) << frequency[face] << endl;
31     }
32 }
```

Alla riga 18 definisco “arraySize”=6, alla riga 19 un array “frequency” che contiene 6 (poiché arraySize=6) unsigned int che sono tutti nulli poiché ho definito l’array come “frequency{}”.

Lo scopo di questo programma è quello di verificare su 60 mln di lanci che la distribuzione di probabilità dei dadi sia effettivamente uniforme e che quindi la funzione rand faccia ciò che deve fare. Ovviamente uscirà sempre la stessa sequenza di lanci, poiché non usiamo srand per cambiare il seme iniziale.

Spesso può essere utile dichiarare degli array di tipo statico, la cui utilità è mostrata nell’esempio a destra. Le variabili di tipo statico vengono allocate soltanto la prima volta all’atto della dichiarazione; poi continuano a vivere e a contenere il valore con il quale sono state istanziate anche al termine dell’esecuzione della funzione in cui sono state definite. Un array statico, se non inizializzato con una lista, viene inizializzato automaticamente a 0. Allocare solo una volta l’array definendolo come statico può essere molto utile, soprattutto nel caso di array di grandi dimensioni, ed

```
24// function to demonstrate a static local array
25void staticArrayInit(void) {
26    // initializes elements to 0 first time function is called
27    static array<int, arraySize> array1; // static local array
28
29    cout << "\nValues on entering staticArrayInit:\n";
30
31    // output contents of array1
32    for (size_t i{0}; i < array1.size(); ++i) {
33        cout << "array1[" << i << "] = " << array1[i] << " ";
34    }
35
36    cout << "\nValues on exiting staticArrayInit:\n";
37
38    // modify and output contents of array1
39    for (size_t j{0}; j < array1.size(); ++j) {
40        cout << "array1[" << j << "] = " << (array1[j] += 5) << " ";
41    }
42}
43// function to demonstrate an automatic local array
44void automaticArrayInit(void) {
45    // initializes elements each time function is called
46    array<int, arraySize> array2{1, 2, 3}; // automatic local array
47
48    cout << "\n\nValues on entering automaticArrayInit:\n";
49
50    // output contents of array2
51    for (size_t i{0}; i < array2.size(); ++i) {
52        cout << "array2[" << i << "] = " << array2[i] << " ";
53    }
54
55    cout << "\nValues on exiting automaticArrayInit:\n";
56
57    // modify and output contents of array2
58    for (size_t j{0}; j < array2.size(); ++j) {
59        cout << "array2[" << j << "] = " << (array2[j] += 5) << " ";
60    }
61}
```

in questo modo ogni volta che richiamo la funzione in cui l'array è definito non c'è bisogno di riallocare l'array.

Cosa utilissima che non possiamo fare con i built-in-arrays del C è confrontare 2 array con gli operatori ==, <=, >, <, !=, che operano confrontando ogni coppia di elementi tra i 2 array. Ovviamente i 2 array devono avere stessa dimensione.

Es:

```
1 #include <iostream>
2 #include <array>
3 using std::cout;
4 using std::cin;
5 using std::array;
6
7
8 int main(){
9     array<int,10> a1{1,2,3,4,5,6,7,8,9,10};
10    array<int,10> a2{1,2,3,4,5,6,7,8,9,10};
11    if(a1==a2)
12        cout <<"a1 is equal to a2 \n";
13    else
14        cout <<"Different \n";
15    if(a1<=a2)
16        cout <<"a1 is less than or equal to a2\n";
17    else
18        cout <<"Different \n";
19    a2[9]=100;
20    if(a1<a2)
21        cout <<"a1 is less than a2\n";
22    else
23        cout <<"Different \n";
24
25 }
```

RANGE BASED “FOR”

È un for particolare presente nello standard 11 di C++ e ci permette di scorrere tutti gli elementi di un array con l'operatore “:” senza istanziare alcun indice.

Es:

```
-- range.cpp
1// Using range-based for to multiply an array's elements by 2.
2// Using range-based for to multiply an array's elements by 2.
3#include <iostream>
4#include <array>
5using std::cout;
6using std::array;
7using std::endl;
8
9int main() {
10    array<int, 5> items{1, 2, 3, 4, 5};
11
12    // display items before modification
13    cout << "items before modification: ";
14    for (int item : items) {
15        cout << item << " ";
16    }
17
18    // multiply the elements of items by 2
19    for (int& itemRef : items) {
20        itemRef *= 2;
21    }
22
23    // display items after modification
24    cout << "\nitems after modification: ";
25    for (int item : items) {
26        cout << item << " ";
27    }
28
29    cout << endl;
30}
```

ARRAY MULTIDIMENSIONALI

Si può costruire un array di array e quindi di fatto una matrice. Basta far sì che il tipo definito nell'array sia a sua volta un array.

Es:

```
1// mArray.cpp
2// Initializing multidimensional arrays.
3#include <iostream>
4#include <array>
5using std::cout;
6using std::array;
7using std::endl;
8
9const size_t rows{2};
10const size_t columns{3};
11void printArray(const array<array<int, columns>, rows>&);
12
13int main() {
14    array<array<int, columns>, rows> array1{1, 2, 3, 4, 5, 6};
15    array<array<int, columns>, rows> array2{1, 2, 3, 4, 5};
16
17    cout << "Values in array1 by row are:" << endl;
18    printArray(array1);
19
20    cout << "\nValues in array2 by row are:" << endl;
21    printArray(array2);
22}
23
24// output array with two rows and three columns
25void printArray(const array<array<int, columns>, rows>& a) {
26    // loop through array's rows
27    for (auto const& row : a) {
28        // loop through columns of current row
29        for (auto const& element : row) {
30            cout << element << ' ';
31        }
32
33        cout << endl; // start new line of output
34    }
35}
```

Unica attenzione: in C (e C++) una matrice la si inizializza per righe; quindi nell'esempio array1 sarà fatto così:

```
array1=  |   1   2   3   |
          |       |
          |   4   5   6   |
          |       |
```

In realtà una cosa che può convenire fare è rappresentare le matrici come array monodimensionali ma scorrere gli elementi con 2 indici.

Notiamo che nell'esempio si è usato *auto const*; "auto" in generale serve per dichiarare variabili il cui tipo è dedotto automaticamente dall'inizializzazione.

Possiamo pensare di risolvere il problema della LCS proprio tramite una struttura matriciale ed in maniera iterativa piuttosto che ricorsiva. D'altra parte abbiamo visto la scorsa lezione che possiamo considerare una matrice da riempire in maniera iterativa per trovare gli elementi della Longest Common Subsequence. In tal caso la complessità computazionale è di n^2 , nel caso in cui n sia il max tra (n,m) $\rightarrow O(n^2)$. Infatti per ogni casella dobbiamo valutare le 2 condizioni:

$$LCS[m, n] = \begin{cases} 1 + LCS[m - 1, n - 1] & s_m = t_n \\ \max[LCS[m - 1, n], LCS[m, n - 1]] & s_m \neq t_n \end{cases}$$

La valutazione di ognuna delle 2 condizioni è $O(1)$.

Fare questo esercizio:

- Given a 2D array A $m \times n$ and a 2D array B $n \times k$:
 - Compute the matrix product AxB
 - What is the computational complexity if $m=n=k$?

In generale dobbiamo sempre chiederci quanto è la complessità computazionale del programma che scriviamo.

VECTORS

I vettori sono molto simili agli array, anzi sono proprio degli array che però possono cambiare di dimensione. Possiamo accedere agli elementi di un vettore esattamente come accediamo agli elementi di un array, le funzioni di accesso sono più o meno le stesse, l'unica cosa particolare è che, per cambiare la dimensione, ogni elemento nuovo da aggiungere va inserito facendo spazio tra gli elementi già esistenti, e quindi shiftando gli elementi in un certo modo; in sostanza va ridimensionato l'array.

Proprio come gli array i vettori sono locazioni contigue di memoria, e proprio come gli array built-in, si può accedere ad un elemento utilizzando l'offset rispetto alla posizione del primo elemento. Internamente (noi non ce ne accorgiamo) i vettori sono degli array dinamicamente allocati; ovviamente non è che ogni volta che aggiungiamo un elemento viene rallocato l'intero vettore. Le diverse librerie implementano diverse strategie per la gestione della crescita, bilanciando l'uso della memoria ed il numero di rallocazioni. Le rallocazioni dovrebbero avvenire solo ad intervalli che crescono logaritmicamente. Per questo rispetto agli array i vettori possono consumare più memoria così da poter essere più flessibili sulle rallocazioni.

Come usiamo i vettori? Dobbiamo includere l'header file <vector> e dopodiché dichiarare un vettore in diversi modi in base alle esigenze:

```
1 // Create an empty vector
2 vector<int> vect1;
3 int n = 3;
4 //Creates a vector with 3 int initialized to zero
5 vector<int> vect2{3};
6 // Create a vector of size n with
7 // all values as 10.
8 vector<int> vect3(n, 10);
9 //Create a vector with initializers
10 vector<int> vect4{ 10, 20, 30 };
```

Ci sono varie funzioni che permettono di inserire elementi in un vettore o comunque di fare operazioni su di esso:

Element access:	
<code>operator[]</code>	Access element (public member function)
<code>at</code>	Access element (public member function)
<code>front</code>	Access first element (public member function)
<code>back</code>	Access last element (public member function)
<code>data</code> <small>(=)</small>	Access data (public member function)

Modifiers:	
<code>assign</code>	Assign vector content (public member function)
<code>push_back</code> <small>(=)</small>	Add element at the end (public member function)
<code>pop_back</code>	Delete last element (public member function)
<code>insert</code>	Insert elements (public member function)
<code>erase</code>	Erase elements (public member function)
<code>swap</code>	Swap content (public member function)
<code>clear</code>	Clear content (public member function)
<code>emplace</code> <small>(=)</small>	Construct and insert element (public member function)
<code>emplace_back</code> <small>(=)</small>	Construct and insert element at the end (public member function)

Vediamo un programma esempio:

```
59 // read input array and call mergesort
60 int main()
61 {
62     vector <int> myarray;
63     int num;
64     int val;
65     cout<<"Enter number of elements to be sorted:";
66     cin>>num;
67     cout<<"Enter "<<num<<" elements to be sorted:";
68     for (int i = 0; i < num; i++) {
69         cin>>val;
70         myarray.push_back(val);
71     }
72     mergeSort(myarray, 0, num-1);
73     cout<<"Sorted array\n";
74     for (int i = 0; i < num; i++)
75     {
76         cout<<myarray[i]<<"\t";
77     }
78     cout << "\n";
79 }
```

Questo esempio ci mostra proprio come allocare dinamicamente un vettore. In tal caso lo si è fatto tramite la funzione *push_back*, inserendo uno alla volta gli elementi nel vettore da tastiera. Alla fine posso scoprire la dimensione del vettore con la funzione *size*, ovvero con *myArray.size()*. Nell'esempio ci si pone anche il problema di ordinare gli elementi del vettore e lo si fa tramite l'algoritmo *mergeSort*. Vediamo come implementare questo algoritmo con un programma.

MERGESORT

```
1 #include <iostream>
2 #include <vector>
3 using std::cout;
4 using std::cin;
5 using std::vector;
6
7 void merge(vector<int> &, int , int );
8
9 //mergeSort
10 void mergeSort(vector<int>& A, int p, int r)
11 {
12     int q;
13     if (p < r){
14         //divide the array at mid and sort independently using merge sort
15         q=(p+r)/2;
16         mergeSort(A,p,q);
17         mergeSort(A,q+1,r);
18         //merge or conquer sorted arrays
19         merge(A,p,r,q);
20     }
21 }
22
23 ...
24 // Merge
25 void merge(vector<int>& A, int p, int r, int q)
26 {
27     int i, j, k;
28     static vector<int> c(A.size());
29     i = p;
30     k = p;
31     j = q + 1;
32     while (i <= q && j <= r) {
33         if (A[i] < A[j]) {
34             c[k] = A[i];
35             k++;
36             i++;
37         }
38         else {
39             c[k] = A[j];
40             k++;
41             j++;
42         }
43     }
44     while (i <= q) {
45         c[k] = A[i];
46         k++;
47         i++;
48     }
49     while (j <= r) {
50         c[k] = A[j];
51         k++;
52         j++;
53     }
54     for (i = p; i < k; i++) {
55         A[i] = c[i];
56     }
57 }
```

Il Mergesort è un algoritmo molto furbo basato sulla ricorsione. Noi fino ad ora abbiamo visto l'algoritmo Insertion Sort e ne abbiamo analizzato la complessità. Il Mergesort ha come caratteristica quella di avere una complessità di tempo di **nlog(n)** e la cosa interessante è che nel problema dell'ordinamento si è studiato che non si può fare meno di $n \cdot \log(n)$ come complessità.

Il mergesort è un esempio del paradigma di progettazione di algoritmi *Divide et Impera* che prevede 3 step:

- **Divide:** se la dimensione dell'input è troppo grande si cerca di dividere il problema in più sottoproblemi disgiunti. La disgiunzione è quello che non avevamo nel caso della LCS, problema caratterizzato proprio dalla sovrapposizione di sottoproblemi e che rientrava nella programmazione dinamica.
- **Impera:** si risolvono i sottoproblemi ricorsivamente con l'approccio Divide et Impera e se la dimensione dei sottoproblemi diventa sufficientemente piccola questi possono essere risolti in maniera diretta.
- **Combina:** le soluzioni dei sottoproblemi sono combinate per ottenere la soluzione al problema originale.

Questa è una procedura generica, naturalmente. Vediamola applicata al Mergesort.

L'idea è quella di dividere una lista molto grande di elementi in 2 liste, ordinare le due liste separatamente e poi combinarle facendone una fusione. Ovviamente se le 2 liste contengono ancora molti elementi si può applicare ricorsivamente l'approccio divide et impera.

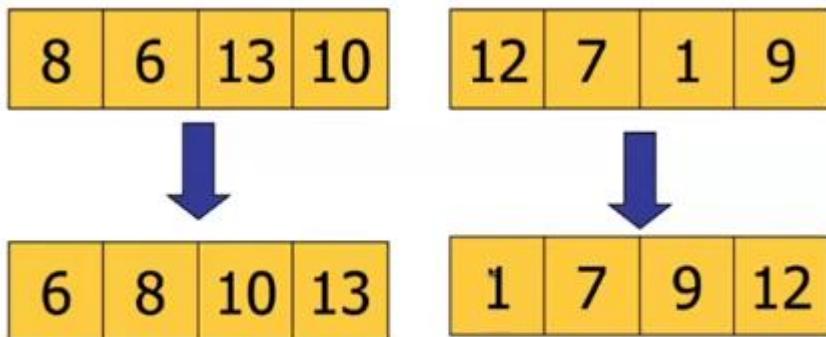
Algoritmo MergeSort

- **Divide:** Se S ha almeno due elementi (in caso contrario non c'e' nulla da ordinare), inserisci tutti gli elementi di S in due sequenze, S_1 ed S_2 , ciascuna contenente circa la metà degli elementi di S . (es. S_1 contiene i primi $\lceil n/2 \rceil$ elementi ed S_2 contiene i restanti $\lfloor n/2 \rfloor$ elementi).
- **Impera:** Ordina le sequenze S_1 ed S_2 usando MergeSort.
- **Combina:** Riposiziona gli elementi di S fondendo le due sequenze, ora ordinate, S_1 ed S_2 in un'unica sequenza ordinata

L'algoritmo Mergesort è fatto così:

```
9//mergeSort
10void mergeSort(vector<int>& A, int p, int r)
11{
12    int q;
13    if (p < r){
14        //divide the array at mid and sort independently using merge sort
15        q=(p+r)/2;
16        mergeSort(A,p,q);
17        mergeSort(A,q+1,r);
18        //merge or conquer sorted arrays
19        merge(A,p,r,q);
20    }
21}
```

La funzione *mergeSort* prende in ingresso un vettore di interi e 2 interi *p* ed *r* che sono l'indice di sinistra e quello di destra. Alla prima chiamata *p* ed *r* sono 0 ed *n*-1. Calcolo allora la metà dell'intervallo [p, r], ovvero *q*=(*p*+*r*)/2 (passo DIVIDE) ed applico l'algoritmo *mergeSort* sugli intervalli [p,q] e [q+1,r].



Chiamata iniziale: MERGE-SORT (*A*, 0, *n*-1)

Il caso base è nascosto nella condizione *p*<*r* dell'if. O meglio, il caso base è proprio quando *p* non è minore di *r* e quindi quando *p*=*r*. In tal caso ho una lista di lunghezza 1 e quindi già ordinata.

C'è poi la parte del COMBINE che è effettuata dalla funzione *merge*: questa ha lo scopo di fondere 2 array assumendo che questi siano già ordinati. La procedura *merge* è implementata così:

```

25void merge(vector<int>& A, int p, int r, int q)
26{
27    int i, j, k;
28    static vector<int> c(A.size());
29    i = p;
30    k = p;
31    j = q + 1;
32    while (i <= q && j <= r) {
33        if (A[i] < A[j]) {
34            c[k] = A[i];
35            k++;
36            i++;
37        }
38        else {
39            c[k] = A[j];
40            k++;
41            j++;
42        }
43    }
44    while (i <= q) {
45        c[k] = A[i];
46        k++;
47        i++;
48    }
49    while (j <= r) {
50        c[k] = A[j];
51        k++;
52        j++;
53    }
54    for (i = p; i < k; i++) {
55        A[i] = c[i];
56    }
57}

```

Questa procedura utilizza al suo interno uno spazio ulteriore di memoria dovuto alla dichiarazione di un vettore c . Notiamo però che questo vettore è dichiarato come statico e quindi viene allocato solo una volta e non ad ogni chiamata di $merge$; l'alternativa poteva essere quella di dichiararlo come variabile globale. Il vettore c serve proprio a contenere la fusione delle 2 liste ed il vettore così costruito viene ricopiatò nel vettore A . La funzione utilizza 3 indici i, j, k :

- i è l'indice che scorre la prima lista e quindi parte dal primo elemento della lista di “sinistra”;
- j è l'indice che scorre la seconda lista e quindi parte dal primo elemento della lista di “destra”;
- k è l'indice che scorre la lista che deve essere ordinata.

Allora l'algoritmo si chiede se l'elemento da inserire in c è un elemento della lista di destra o uno della lista di sinistra, e lo fa nell'if else all'interno del while.

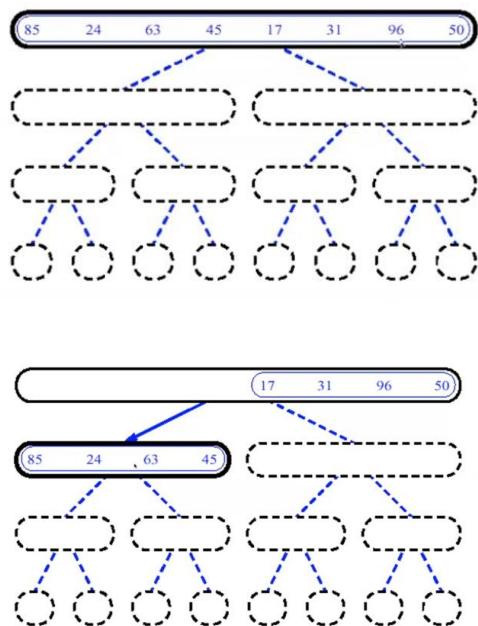
Quando il while fallisce, e prima o poi fallisce perché una delle 2 liste sicuramente arriva ad essere scorsa (si dice così?) fino alla fine, allora ci sono altri 2 while per gestire questo caso. Se viene scorsa la lista di destra completamente allora c'è un while che copia gli elementi della lista sinistra rimanenti nel vettore c ; se è la lista sinistra ad essere stata scorsa completamente allora c'è un while che copia gli elementi rimanenti della lista di destra nel vettore c . A questo punto c'è un for finale che copia il vettore c , ordinato, nel vettore A originario.

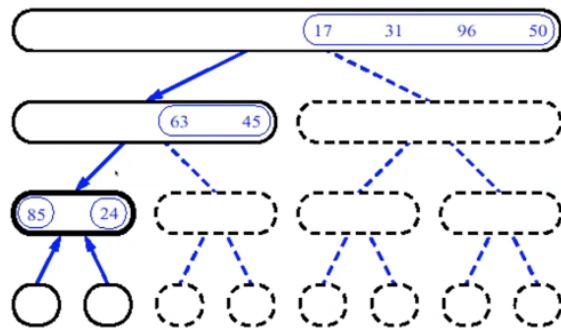
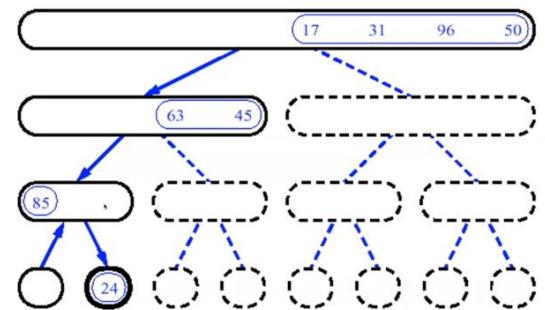
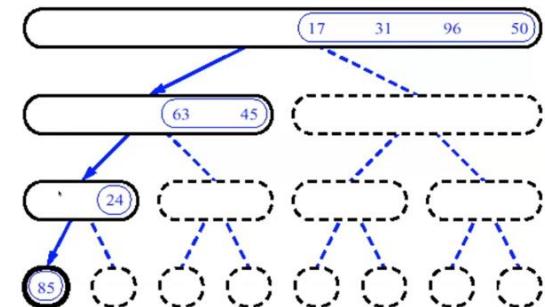
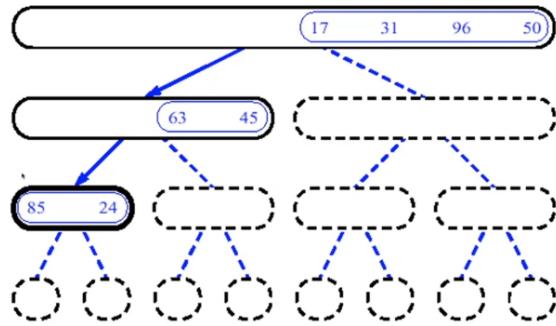
Analisi dell'algoritmo:

Mergesort scorre a sinistra e a destra e costruisce la lista ordinata. Il tempo di esecuzione è quello per scorrere la lista di partenza, nel senso che le 2 liste che si scorrono hanno metà degli elementi ovviamente. Quindi si ha:

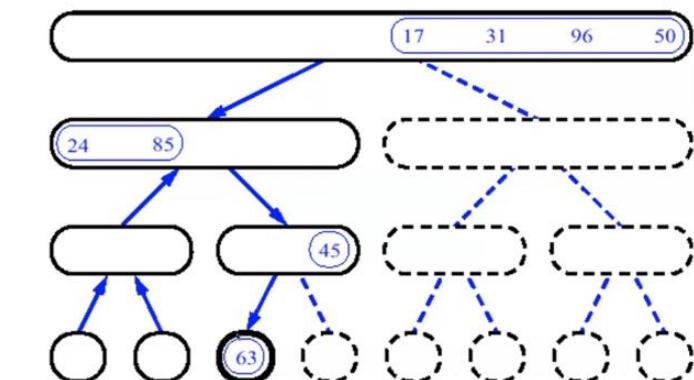
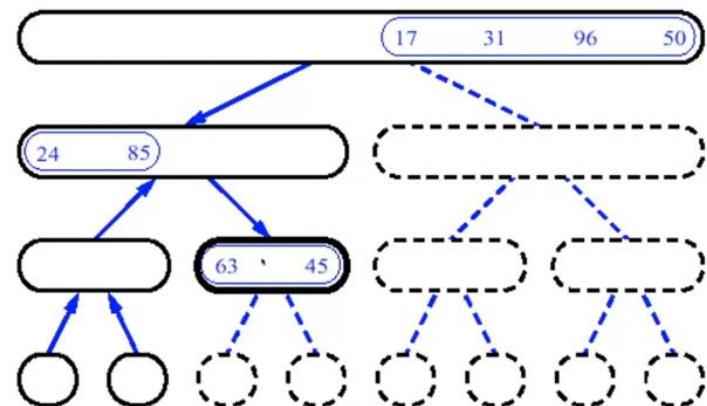
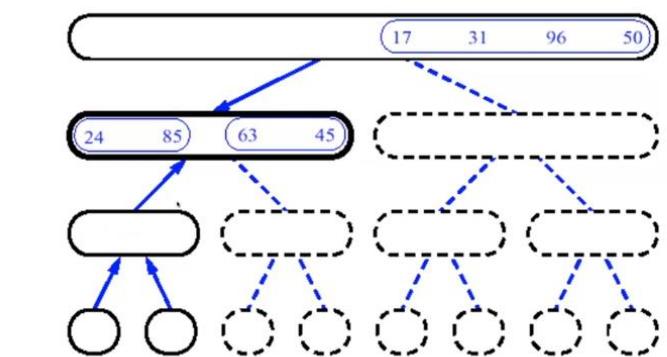
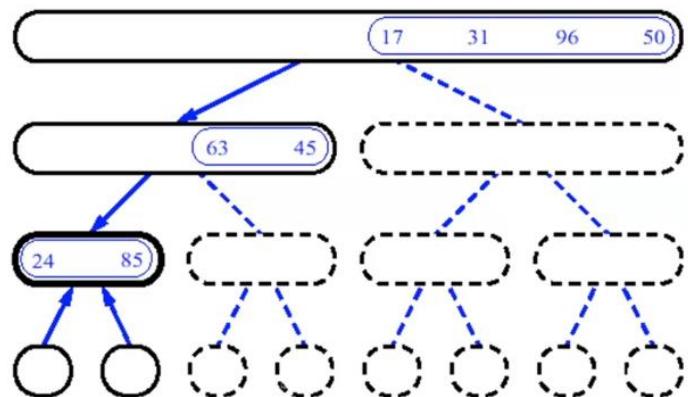
**Il tempo di
esecuzione
di MERGE è $\Theta(r-p+1)$**

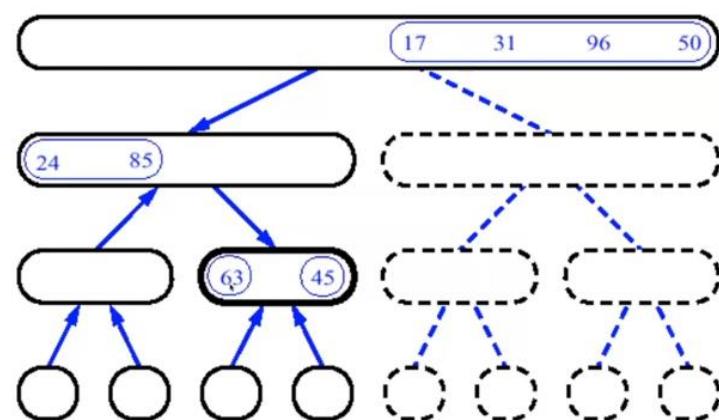
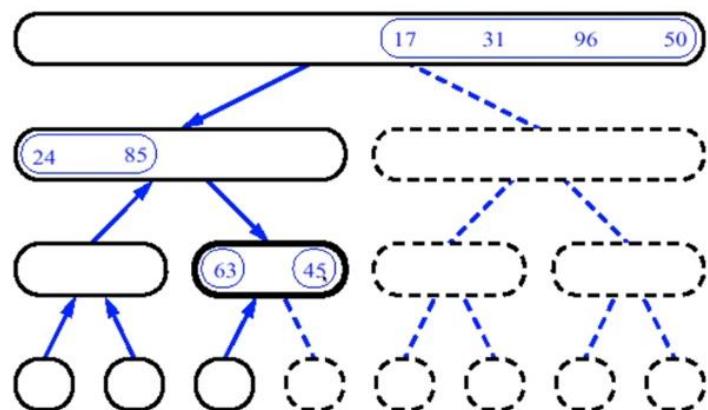
Vediamo graficamente un esempio:



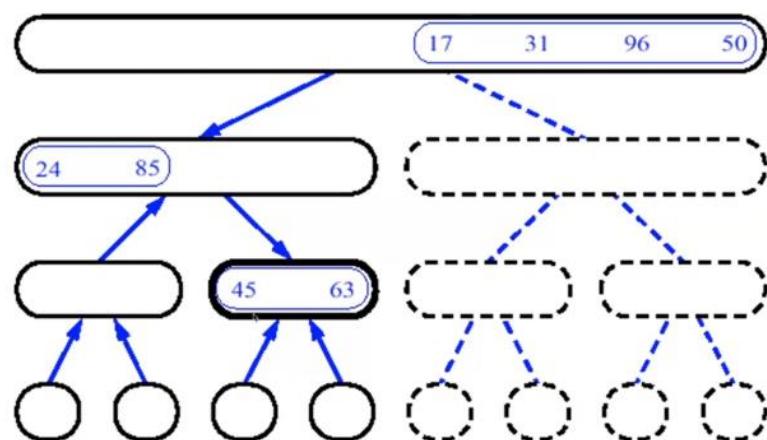


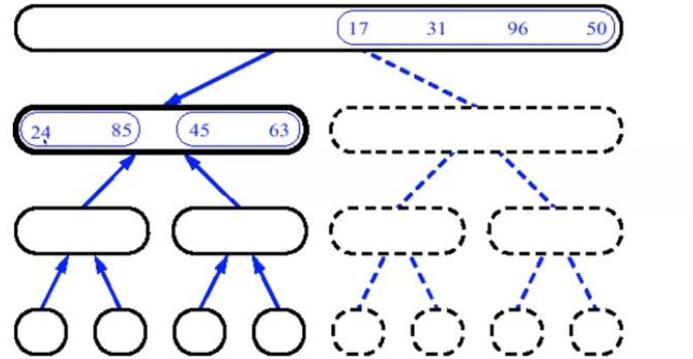
A questo punto viene invocata la funzione merge, che fonde in maniera corretta 85 e 24, elementi che presi singolarmente sono già ordinati ovviamente.



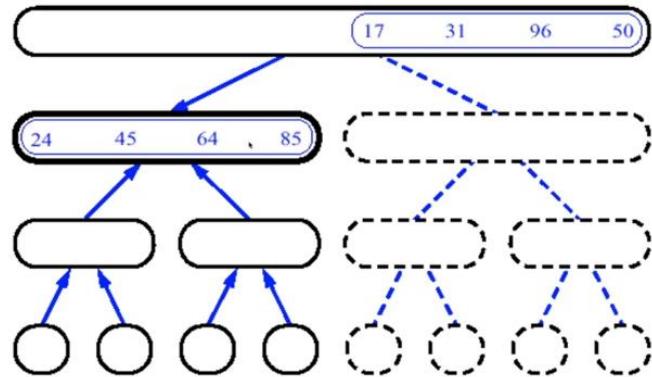


Viene invocata merge:

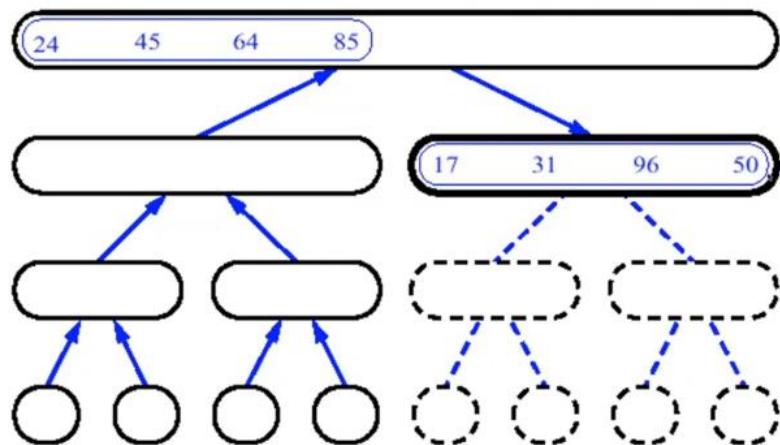




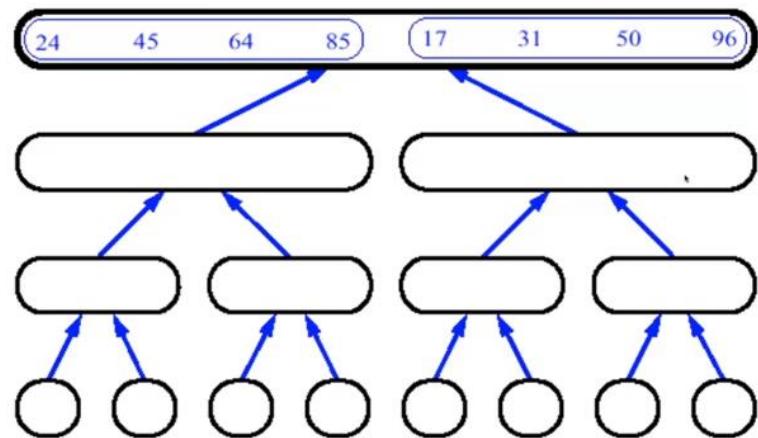
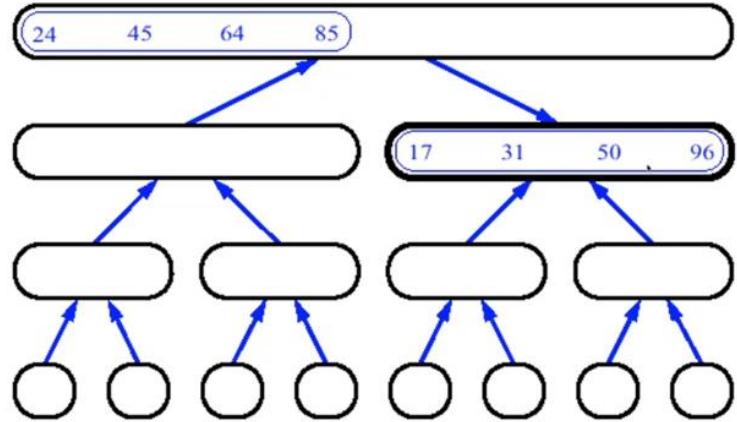
A questo punto viene invocata la funzione merge sulle 2 liste (24,85) e (45,63) per ottenere una lista ordinata:



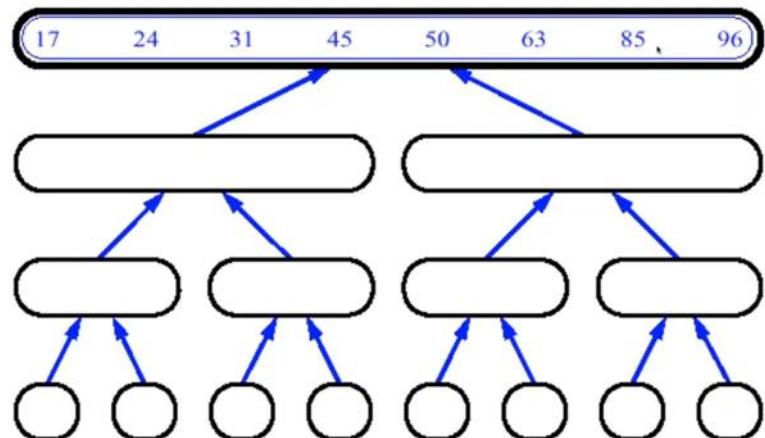
A questo punto il controllo viene restituito alla funzione di merge(0,7) e viene invocato mergesort sulla metà lista di destra.



Succede tutto quello che abbiamo visto per la metà lista di sinistra ed alla fine si giunge a:



Merge finale:



Parliamo adesso della complessità computazionale del merge sort.

Il merge sort è un algoritmo ricorsivo e quindi in realtà il suo tempo di esecuzione può essere descritto a sua volta da una relazione di ricorrenza.

Una **ricorrenza** è un'equazione o una diseguaglianza che descrive una funzione in termini del suo valore su input sempre più piccoli

La complessità del merge sort la possiamo vedere così:

```
MERGE-SORT ( $A, p, r$ )
1. if  $p < r$  then
2.    $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
3.   MERGE-SORT ( $A, p, q$ )
4.   MERGE-SORT ( $A, q+1, r$ )
5.   MERGE( $A, p, q, r$ )
```

$T(n/2) + T(n/2)$

```
MERGE-SORT ( $A, p, r$ )
1. if  $p < r$  then
2.    $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
3.   MERGE-SORT ( $A, p, q$ )
4.   MERGE-SORT ( $A, q+1, r$ )
5.   MERGE( $A, p, q, r$ )
```

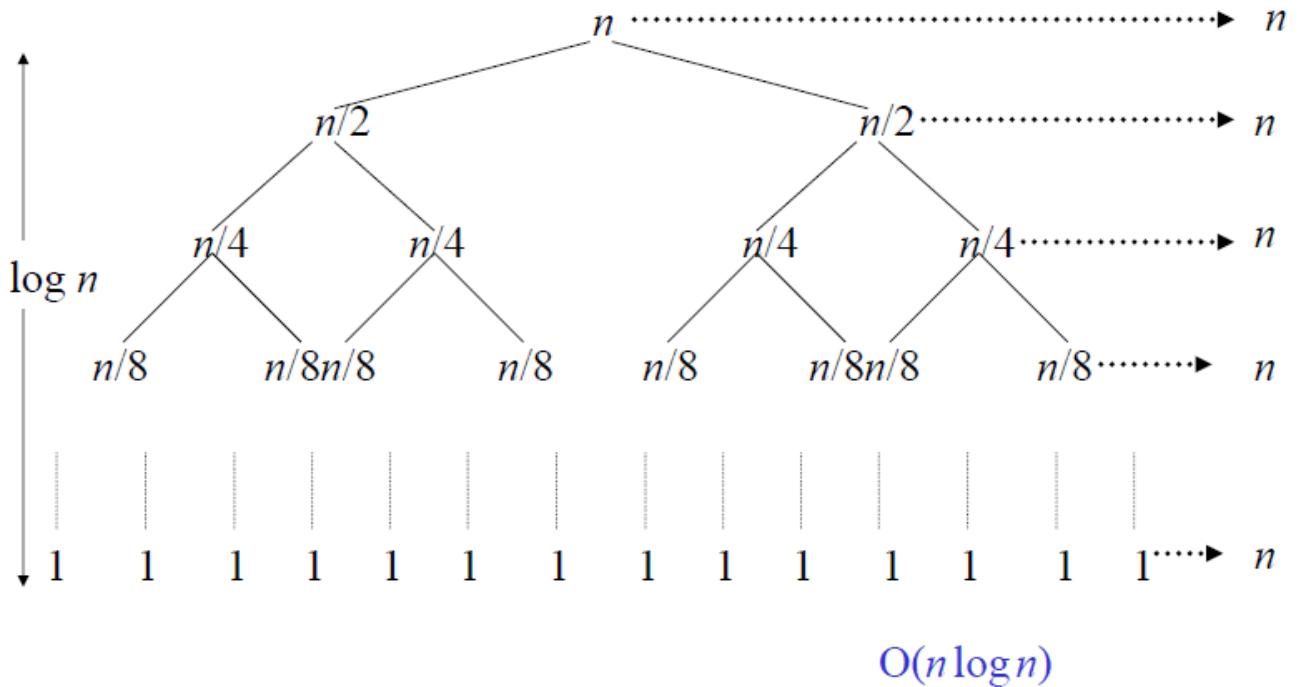
$\Theta(n)$

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Chiamata iniziale: MERGE-SORT ($A, 1, n$)

Questa è la relazione di ricorrenza. In maniera formale bisognerebbe risolvere l'equazione, ma noi cerchiamo di capire in maniera intuitiva quanto è $T(n)$ e lo facciamo tramite l'albero delle chiamate ricorsive.

ALBERO DELLE CHIAMATE RICORSIVE:



Il numero di volte in cui devo dividere per 2 il numero di elementi prima di arrivare ad una lista che so ordinare, ovvero una lista di 1 elemento, è pari esattamente a $\log(n)$. Su ogni livello che succede? Al primo livello devo fare un merge di n elementi $\rightarrow n$ operazioni. Scendendo di un livello il merge deve fare un numero di operazioni proporzionale ad $n/2 + n/2 \rightarrow n$ operazioni. E così via. Insomma il numero di operazioni è costante e pari ad n ad ogni livello. Di livelli ce ne sono $\log(n)$ \rightarrow in totale avremo:

$$T(n) = O(n * \log(n))$$

LEZIONE 11

Abbiamo visto l'algoritmo Merge Sort dove c'è una funzione *mergeSort* che divide la lista di partenza in liste di dimensione pari alla metà ed invoca la funzione *merge* che è quella che effettua la fusione delle liste ordinate, a partire da liste di 1 elemento ottenute tramite *mergeSort*. Qual è la differenza tra quest'algoritmo e quello della LCS? È che nel secondo caso i sottoproblemi sono sovrapposti, nel primo no.

BINARY SEARCH (Ricerca Binaria)

È il problema della ricerca di un elemento all'interno di un insieme ordinato di elementi. Quindi: abbiamo introdotto i vettori, abbiamo visto qualche metodo di ordinamento ed ora vediamo il problema della ricerca di un elemento all'interno di un vettore ordinato.

Sicuramente si può scorrere un vettore dall'inizio alla fine fino a trovare l'elemento di proprio interesse, se presente: questo procedimento si chiama *ricerca sequenziale*.

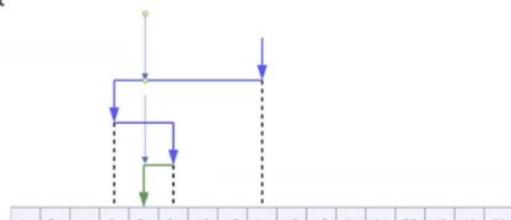
Tuttavia la ricerca binaria risulta essere la più efficiente e sfrutta il previo ordinamento del vettore per adottare l'approccio Divide et Impera. L'idea è simile a quando cerchiamo un nome all'interno di un'agenda senza indici. Ho una lista di n elementi ordinati ed il problema della ricerca viene ridotto a 3 opzioni possibili: potrebbe essere che l'elemento che sto cercando è quello centrale, e quindi come prima cosa confronto l'elemento da ricercare con quello centrale nella lista: se sono uguali allora sono fortunato e la ricerca si conclude; se non sono uguali confronto i due elementi e se quello che sto cercando è maggiore dell'elemento centrale allora la ricerca va effettuata solo dall'elemento centrale in poi, quindi sulla metà finale della lista; viceversa, se l'elemento da ricercare è minore dell'elemento centrale allora la ricerca va effettuata nella prima metà della lista. Queste 2 ultime operazioni si ripetono fino a quando non si ha che il valore ricercato corrisponde effettivamente al valore medio dell'intervallo che si è mano a mano ristretto.

Vediamo l'algoritmo nello specifico.

```
int binarySearch(vector<int> &A, int num) {
    int n = A.size();
    int left = 0;
    int right = n-1;

    while (left <= right){
        int mid = (right+left)/2;

        if(A[mid] == num)
            return(mid);
        if(A[mid] < num)
            left = mid+1;
        else
            right = mid-1;
    }
    return -1;
}
```



I numeri nella lista e nelle sottoliste che si vanno a creare sono sempre right-left+1. Se l'elemento ricercato non c'è nella lista allora si arriverà ad una lista di 1 elemento e al dimezzamento gli indici right e left si sovrapporranno e scambieranno; questo genera un'uscita dal while poiché left diventa maggiore di right e quindi viene restituito -1 come valore sentinella per indicare che il valore ricercato nella lista non è presente in essa.

Nel caso della *sequential search* il caso peggiore è quello in cui non è presente l'elemento nella lista e richiede che venga scorso tutto il vettore → almeno n confronti → $O(n)$.

Nel caso del *binary search* invece il caso peggiore, che è sempre quello in cui non è presente l'elemento nella lista, prevede una crescita logaritmica del numero di operazioni → $O(\log_2(n))$. Infatti al primo passo si passa da una lista di lunghezza n ad una di lunghezza $n/2$, al secondo passo da $n/2$ a $n/4$, e così via. Quindi dopo k confronti (k passi) la lista avrà $n/(2^k)$ elementi. La lista conterrà un unico elemento quando:

$$n/(2^k) = 1 \rightarrow n=2^k \rightarrow k = \log_2(n).$$

Allora si vede che la ricerca sequenziale ha una crescita lineare di complessità, quella binaria ha una crescita logaritmica: è decisamente più conveniente la ricerca binaria.

Come esercizio, implementare binarySearch tramite una funzione ricorsiva piuttosto che una iterativa.

QUICK SORT

È un altro algoritmo di ordinamento basato sulla ricorsione e che usa sempre l'approccio Divide et Impera. Usa un approccio ancora più furbo degli altri algoritmi ed è molto veloce: ha una complessità di $n\log(n)$ → $O(n\log(n))$.

Questa complessità è la minima per un algoritmo di ordinamento e lo si dimostra proprio (noi non lo facciamo ovviamente). È molto efficiente perché anche il numero di fattori costanti all'interno dell'algoritmo è molto basso. Tuttavia il quick sort presenta un caso medio ed un caso migliore favorevoli poiché di complessità $O(n\log(n))$, ma un caso peggiore sfavorevole poiché di complessità $O(n^2)$.

Come già detto, il quick sort è un algoritmo basato sul paradigma Divide et Impera:

- **Divide:** L'array $A[p, \dots, r]$ è partizionato in due sottoarray non vuoti $A[p, \dots, q]$ e $A[q+1, \dots, r]$ tali che
 - tutti gli elementi di $A[p, \dots, q]$ sono minori o uguali degli elementi di $A[q+1, \dots, r]$,
 - l'indice q è calcolato durante la procedura di partizionamento
- **Impera:** I due sottoarray $A[p, \dots, q]$ e $A[q+1, \dots, r]$ sono ordinati attraverso due chiamate ricorsive a quicksort
- **Combina:** non è necessaria alcuna combinazione

L'idea di base è quella di mettere tutti i valori piccoli a sinistra e tutti quelli grandi a destra in modo tale che tutti i valori a sinistra saranno più piccoli di quelli a destra. Quanto piccoli e quanto grandi devono essere questi valori lo vediamo poi. Dopo aver diviso la lista di partenza in queste 2 liste si ordinano queste ultime con 2 chiamate ricorsive alla stessa funzione *quickSort*. Il mergeSort divideva, poi faceva la parte “Impera” e poi la maggior parte del lavoro era fatto nella fase “combina” dalla funzione merge che faceva la fusione di sottoliste. Nel caso del quickSort invece gran parte del lavoro è fatto dal partizionamento, ovvero dal passo “Divide” nel quale si separano valori “piccoli” da valori “grandi”, mettendo a sinistra i valori che sono minori o uguali di quelli a destra. Per questo algoritmo non c’è bisogno di alcun passo di combinazione perché basta affiancare gli elementi di destra a quelli di sinistra. Vediamo la procedura del quickSort:

```
9 void quickSort(vector<int> &A, int p, int r){  
10    if(p < r){  
11        int q = partition(A, p, r);  
12        quickSort(A,p,q);  
13        quickSort(A,q+1,r);  
14    }  
15 }
```

Chiamata iniziale: $\text{quickSort}(A, 0, n-1)$

Abbiamo scelto come tipo quello intero, ma in realtà va bene qualsiasi tipo per cui siano definiti gli operatori $>$, $<$, \leq , \geq , $=$.

Il caso base è quello in cui la lista sia vuota o contenga un unico elemento. Il caso in cui ci sia un elemento è quello in cui $p=r$ e ovviamente non c'è bisogno di fare alcun ordinamento.

Se ci sono almeno 2 elementi da ordinare allora *quickSort* invoca un'altra funzione che si chiama *partition(A,p,r)* e che restituisce l'indice q tale che tutti gli elementi che stanno tra p e q sono minori o uguali di tutti gli elementi che stanno tra $q+1$ ed r . Fatta la divisione del vettore di partenza nei 2 gruppi di elementi, viene invocata la funzione *quickSort* ricorsivamente sia sul primo gruppo che sul secondo. Tutto il lavoro sporco lo fa la procedura *partition* per trovare l'indice q . Vediamo com'è fatta.

```
23 int partition(vector<int> &A, int p, int r) {  
24     int x = A[p];  
25     int i = p-1;  
26     int j = r+1;  
27     while(true) {  
28         do  
29             j = j - 1;  
30         while(A[j] > x);  
31         do  
32             i = i + 1;  
33         while (A[i] < x);  
34         if( i < j)  
35             swap(A,i,j);  
36         else  
37             return j;  
38     }  
39 }
```

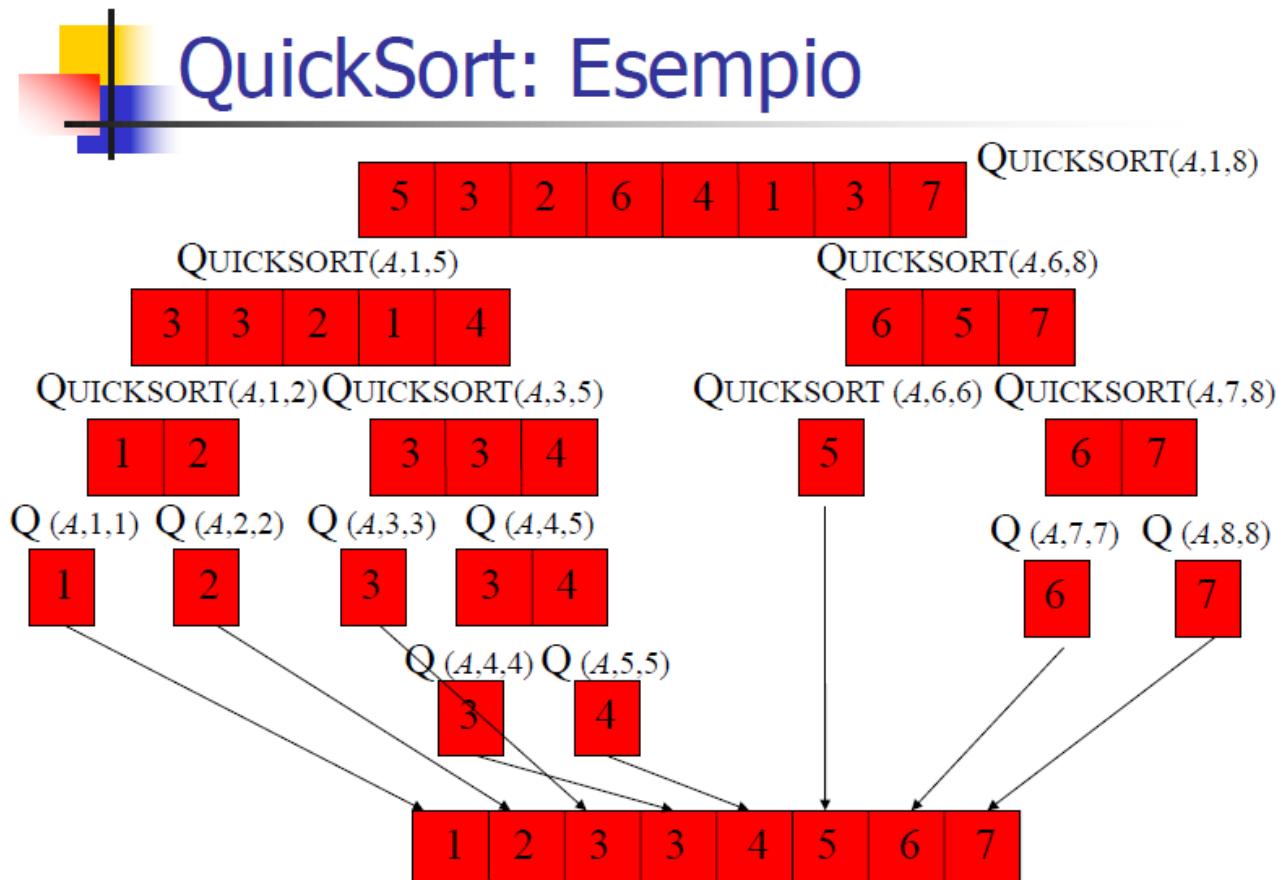
Si sceglie un elemento dell'array, in questo caso il primo (p), come elemento pivot x , ovvero quello in base al quale si effettua il partizionamento. Si utilizzano a questo punto 2 indici i e j , il primo che parte un po' a sinistra del primo elemento (che coincide con il pivot nel nostro caso) e il secondo che parte un po' più a destra dell'ultimo elemento dell'array; si sposta l'indice di sinistra verso destra e l'indice di destra verso sinistra fino a trovare da sinistra un elemento che sia maggiore o uguale del pivot e da destra uno che sia minore o uguale del pivot. In particolare si comincia a muovere j verso sx di una posizione in modo che punti proprio all'ultima casella e poi se e finché l'elemento a cui punta j è maggiore del pivot allora si continua a muovere j . Appena j punta ad un elemento minore del pivot finisce questo while e si sposta di una posizione l'indice i in modo che punti proprio alla prima posizione (che è il pivot in questo caso).

A questo punto si sposta i verso destra fin quando l'elemento a cui punta i è minore del pivot; appena diventa maggiore del pivot si finisce anche questo while e si entra in un if in cui si effettua uno scambio tra l'elemento puntato da i e quello puntato da j se $i < j$, ovvero se i due indici non si sovrappongono/scambiano. Se invece non si entra nell'if, ovvero se gli indici si sovrappongono, allora finisce la procedura *partition* e viene restituito j come elemento pivot poiché j sicuramente punterà al pivot. La procedura termina sicuramente e non c'è rischio di loop infinito poiché sicuramente ad un certo punto i due indici si "scontrano". *Swap* molto semplicemente fa lo swap tra 2 elementi del vettore, creando una variabile temporanea locale alla funzione.

Quanto ci costa la funzione *partition*? La funzione partition fa uno scorrimento verso destra ed uno verso sinistra, quindi di fatto al più 2 volte e quindi la complessità è proporzionale comunque al numero degli elementi:

- $O(n)$, con $n=r-p+1$

16



Oss: in questo esempio gli indici sono scalati tutti di uno verso destra, nel senso che gli elementi partono dalla posizione 1 e finiscono alla posizione 8.

Qual è il caso migliore? Ovvero, quando questa procedura è più veloce?

Se succedesse che a valle di *partition* si ha una divisione sempre uguale di elementi, ovvero se il pivot fosse sempre l'elemento centrale (a livello di grandezza), allora abbiamo praticamente la stessa complessità del merge sort. Infatti il tempo di esecuzione dell'algoritmo dipende dalla procedura *partition*, che abbiamo visto avere complessità $O(n)$, e dalle 2 chiamate ricorsive a *quickSort* che supponiamo in tal caso essere su liste di $n/2$ elementi. Quindi:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) = 2(2T(n/4) + \Theta(n/2)) + \Theta(n) = \\ &= 4T(n/4) + 2\Theta(n) = 4(2T(n/8) + \Theta(n/4)) + 2\Theta(n) = \\ &= 8T(n/8) + 3\Theta(n) = \dots = nT(1) + n \log n \Theta(1) \end{aligned}$$

la stessa relazione di ricorrenza del MergeSort, ovvero:

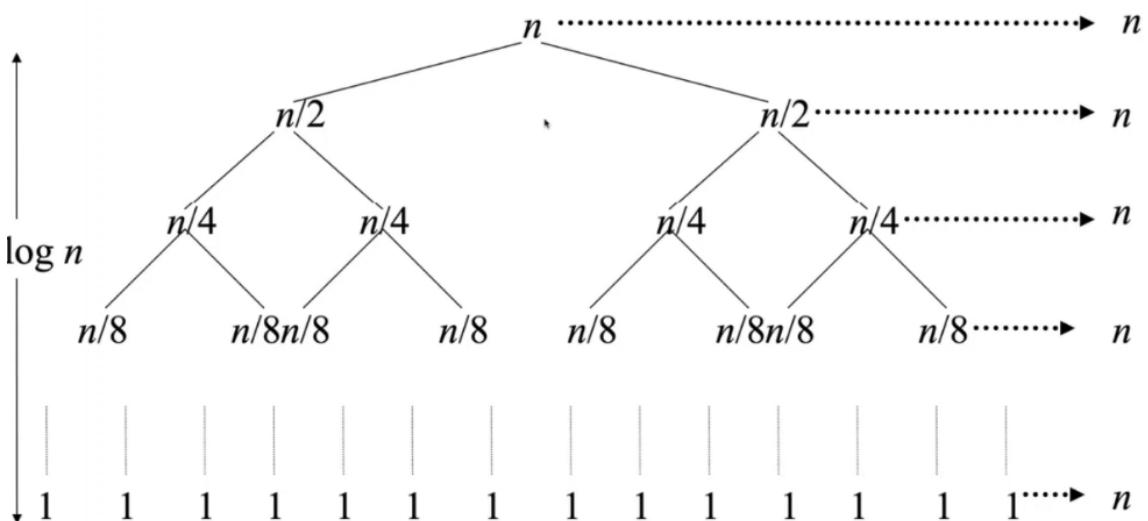
$$T(n) = 2T(n/2) + \Theta(n)$$

che avevamo già visto avere soluzione:

$$T(n) = \Theta(n \log n)$$

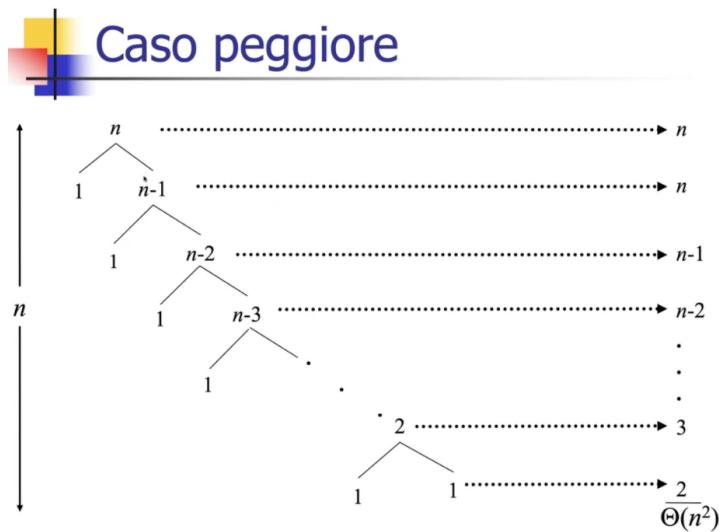
Possiamo sempre visualizzare la complessità tramite un albero:

Analisi: caso migliore (2)



C'è un numero di operazioni costante e pari ad n su ogni livello dell'albero, e l'albero è profondo $\log(n)$ → $O(n \log(n))$.

Il caso peggiore invece corrisponde a quello cui tutti gli elementi o vanno a sinistra o vanno a destra e quindi la complessità è praticamente quella dell'insertion sort. Infatti la prima chiamata di *partition* sulla lista di ampiezza n genera 2 sottoliste, una da $n-1$ elementi ed una da 1 elemento. Di queste due liste, quella da $n-1$ elementi viene divisa in 2 sottoliste, una di ampiezza 1 ed una di ampiezza $n-2$. E così via.



Il numero di livelli delle chiamate ricorsive è proporzionale ad n , poiché ogni volta n cresce di un fattore costante che è 1, ad ogni livello il numero di operazioni è pari ad n e quindi nel caso peggiore si vede che la complessità è:

$$T(n) = O(n^2)$$

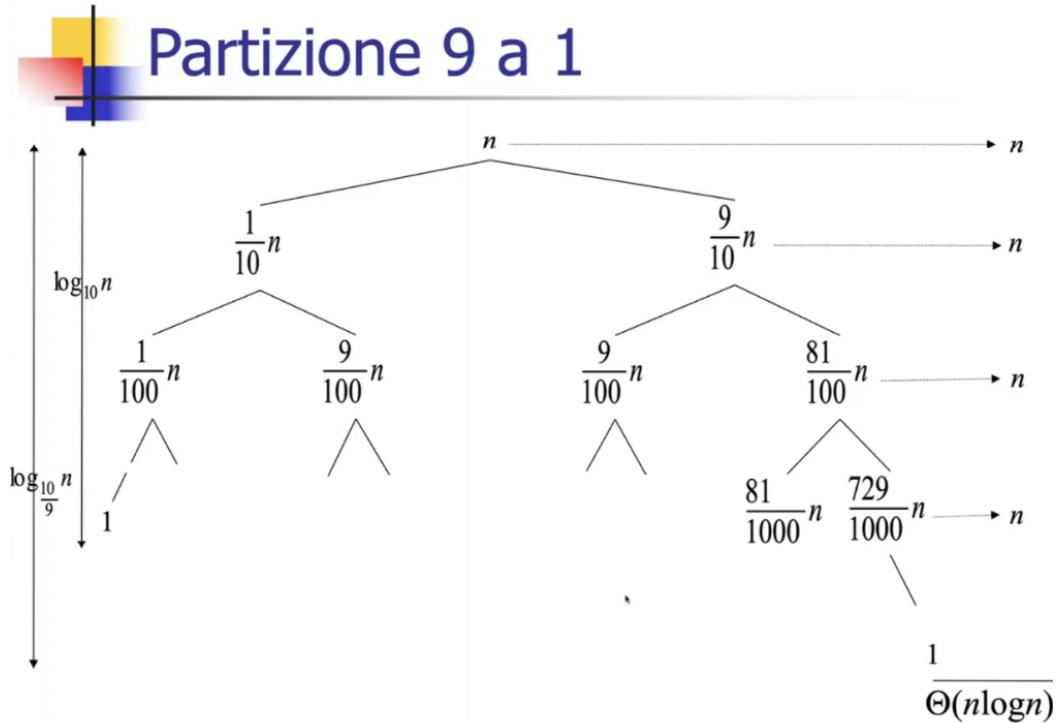
Questo caso peggiore lo si ha o quando il pivot è il più piccolo di tutti gli elementi, e quindi sta a sinistra di tutti, o quando è il più grande di tutti e quindi a destra c'è il pivot e a sinistra ci sono tutti gli altri elementi. Nel primo caso vuol dire che l'array è ordinato in ordine crescente, nel secondo caso in ordine decrescente.

Qual è il caso medio? Nell'insertion sort era quello con complessità $O(n^2)$; nel quick sort invece il caso medio resta con complessità:

$$T(n) = O(n \log(n))$$

Il caso medio è piuttosto complicato da studiare, noi lo vediamo solo intuitivamente. Anche se la lista è sbilanciata, e quindi magari il partizionamento produce sempre il 90% degli elementi a sinistra ed il 10% a destra, in realtà andiamo comunque a fare una suddivisione costante (non per ottenere $1/2$ e $1/2$ ma per ottenere $1/10$ e $9/10$ della

lista da suddividere) e quindi guardando l'albero delle chiamate ricorsive ci rendiamo conto che il numero delle divisioni da fare (dividere n per $10/9$ e $1/9$) per ottenere una lista di ampiezza 1 è proporzionale al logaritmo di n :



Potremmo osservare che c'è una bella differenza tra un logaritmo in base 10 ed uno in base $10/9$: il primo produce valori molto più grandi e sarebbe più conveniente. Tuttavia basta ricordare la relazione che sussiste tra i cambi di base: ovvero, \log_{10} e $\log_{10/9}$ sono legati da un fattore costante e nelle analisi asintotiche abbiamo detto che i fattori costanti li trascuriamo. Quindi che ad ogni passo la lista sia divisa in 2 liste con metà degli elementi o che sia divisa in 2 liste con, rispettivamente, $9/10$ degli elementi ed $1/10$, non ci interessa in termini di complessità asintotica. Per questo il caso medio si avvicina, nel caso del quick sort, molto di più al caso migliore che a quello peggiore.

Come facciamo ad impedire che un avversario malizioso ci dia delle liste ordinate in un senso o nell'altro e che quindi farebbero avere un caso peggiore?

Si effettua una randomizzazione. Quindi prima di chiamare partition si prende un elemento a caso nella lista, lo si sposta in prima posizione e si prende quello come pivot.

RANDOMIZED-PARTITION(A,p,r)

1. $i \leftarrow \text{RANDOM}(p,r)$
2. exchange $A[p] \leftrightarrow A[i]$
3. **return** PARTITION(A,p,r)

RANDOMIZED-QUICKSORT(A,p,r)

1. **if** $p < r$
2. **then** $q \leftarrow \text{RANDOMIZED-PARTITION}(A,p,r)$
3. RANDOMIZED-QUICKSORT(A,p,q)
4. RANDOMIZED-QUICKSORT($A,q+1,r$)

Il vantaggio rispetto al merge sort è quello che il merge sort richiedeva memoria ulteriore per l'allocazione del vettore c , mentre il quick sort non richiede alcuno spazio aggiuntivo in memoria e per questo si dice che effettua un ordinamento *in place*. Questo è un grosso vantaggio quando dobbiamo utilizzare questo algoritmo per ordinare grossi vettori.

Le uniche operazioni che fa il quick sort sono confronti e scambi, quindi possiamo ordinare non solo degli int ma qualsiasi lista su cui sono definite le operazioni di $>$ e $<$. Quindi si potrebbe, per esercizio, costruire un template per applicare questo algoritmo indipendentemente dal tipo di dato che contiene il vettore.

Fare poi quest'esercizio:

- Write a program that reads integers from the keyboard until EOF and stores the values in a vector
 - Before adding an element to the vector it check if the value is already in the vector, if it is present than the value is not added
 - At the end the program outputs the number of values stored in the vector and the values
 - If you have entered n numbers m of which are different, can you compute the worst case analysis?
 - What is the computational complexity if you mantain the vector ordered? (assume the `insert` operation to be constant, even it is not!)

Tenere a mente che la fine del file (EOF) coincide con ctrl-d o ctrl-z in base al sistema operativo.

STANDARD ALGORITHM LIBRARY

La libreria standard mette a disposizione delle funzioni di libreria che prendono in ingresso degli *iterators*, ovvero delle classi che permettono di scorrere in una struttura dati complessa come un array. In particolare in questa libreria c’è una funzione *sort*, il cui prototipo è definito nel namespace standard e la cui implementazione sta nell’header <algorithm>, che prende in ingresso un puntatore, ovvero un riferimento ad un iteratore che consiste in un riferimento al primo e all’ultimo elemento dell’array. Quindi alla funzione *sort* vanno passati i riferimenti al primo elemento ed all’ultimo elemento (*begin* and *end*).

E allora? Abbiamo perso tempo studiando tutti quegli algoritmi di ordinamento?

Di fatto sì.

ECCEZIONI

Le eccezioni (“exceptions”) corrispondono a dei casi eccezionali che possono coincidere anche con errori all’interno del nostro programma. Possiamo “catturare” le eccezioni in modo da tenere sotto controllo eventuali condizioni di errori fatali che possono avvenire durante l’esecuzione di un certo programma. Noi usiamo le eccezioni per indicare un errore fatale che avviene e che fa sì che il programma non possa continuare: in tal caso infatti il programma lancia un’eccezione ed un’altra parte del programma cerca di catturarla per poi risolvere il problema che è occorso. Il punto in cui avviene il problema che genera l’eccezione si chiama anche *throw point*.

Facciamo un esempio di eccezione:

```
1 #include <iostream>
2 #include <vector>
3 #include <stdexcept> // standard exception classes defined
4
5 using std::cout; using std::endl; using std::vector;
6
7 int main() {
8     vector<int> vec = {1, 2, 3};
9     try {
10         cout << vec.at(3) << endl;
11     } catch(const std::out_of_range& e) {
12         cout << "Exception: " << endl << e.what() << endl;
13     }
14     return 0;
15 }
```

```
$ g++ exception1.cpp -std=c++11 -o exception1 -Wall -pedantic
$ ./exception1
Exception:
vector::_M_range_check: __n (which is 3) >= this->size() (which is 3)
```

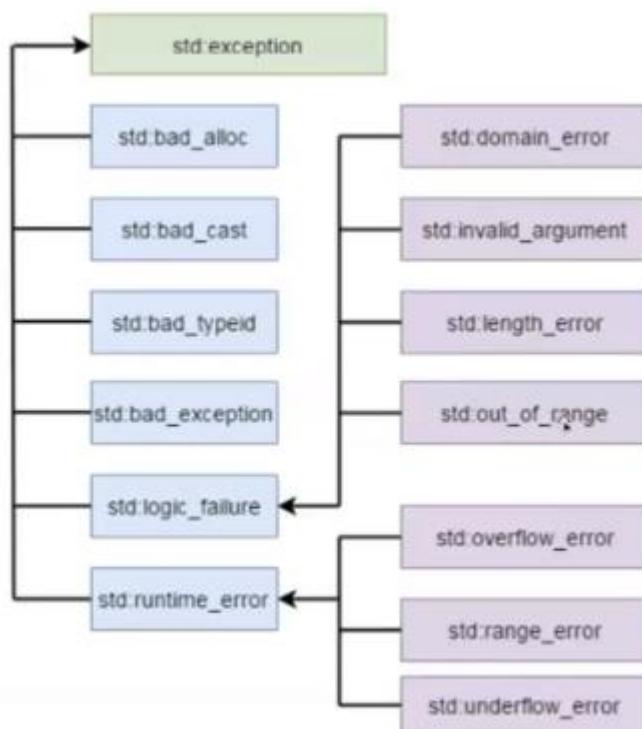
Usiamo `vec.at(3)` per accedere alla quarta casella del vettore `vec`. Tuttavia questo vettore ha 3 elementi in realtà e quindi questo genera un errore fatale.

Oss: accedere con l'operatore `[]` è meno sicuro di accedere con `.at`, nel senso che il primo non rileva possibili errori di questo tipo, come l'accesso ad un elemento che non esiste nell'array, mentre il secondo sì.

Tornando all'esempio, eventuali istruzioni che possono generare errori fatali possono essere inserite all'interno della clausola `try` che esegue delle istruzioni cercando però di catturare eventuali segnali lanciati dalle istruzioni che si trovano all'interno della clausola. In particolare quindi nel blocco `try` vengono eseguite istruzioni che potrebbero lanciare un'eccezione che verrebbe catturata dall'istruzione `catch` successiva al blocco `try`.

Per poter includere le eccezioni dobbiamo ovviamente includere la libreria `<stdexcept>`.

Esistono in realtà diversi tipi di eccezioni, motivo per cui si possono usare più “`catch`” per catturare specifiche eccezioni; una delle specifiche eccezioni è proprio `out of range` utilizzata nel programma. In generale dalla classe generica `std::exception` si possono derivare più classi che rappresentano le eccezioni particolari e che sono queste:



Tutte le eccezioni hanno una funzione membro che si chiama `what` e che restituisce una stringa che spiega cosa è successo.

Esempio più articolato:

```
1 // vectorDemo.cpp
2 // Demonstrating C++ Standard Library class template vector.
3 #include <iostream>
4 #include <iomanip>
5 #include <vector>
6 #include <stdexcept>
7 using std::cout;
8 using std::cin;
9 using std::cerr;
10 using std::endl;
11 using std::vector;
12 using std::out_of_range;
13 using std::vector<int>;
14 void outputVector(const vector<int>&); // display the vector
15 void inputVector(vector<int>&); // input values into the vector
16
17 int main() {
18     vector<int> integers1(7); // 7-element vector<int>
19     vector<int> integers2(10); // 10-element vector<int>
20
21     // print integers1 size and contents
22     cout << "Size of vector integers1 is " << integers1.size()
23     << "\nvector after initialization: ";
24     outputVector(integers1);
25
26     // print integers2 size and contents
27     cout << "\nSize of vector integers2 is " << integers2.size()
28     << "\nvector after initialization: ";
29     outputVector(integers2);
30
31     // input and print integers1 and integers2
32     cout << "\nEnter 17 integers: " << endl;
33     inputVector(integers1);
34     inputVector(integers2);
35
36     cout << "\nAfter input, the vectors contain:\n"
37     << "integers1: ";
38     outputVector(integers1);
39     cout << "integers2: ";
40     outputVector(integers2);
41
42     // use inequality (!=) operator with vector objects
43     cout << "\nEvaluating: integers1 != integers2" << endl;
44
45     if (integers1 != integers2) {
46         cout << "integers1 and integers2 are not equal" << endl;
47     }
48
49     // create vector integers3 using integers1 as an
50     // initializer; print size and contents
51     vector<int> integers3(integers1); // copy constructor
52
53     cout << "\nSize of vector integers3 is " << integers3.size()
54     << "\nvector after initialization: ";
55     outputVector(integers3);
56
57     // use overloaded assignment (=) operator
58     cout << "\nAssigning integers2 to integers1" << endl;
59     integers1 = integers2; // assign integers2 to integers1
60
61     cout << "integers1: ";
62     outputVector(integers1);
63     cout << "integers2: ";
64     outputVector(integers2);
65
66     // use equality (==) operator with vector objects
67     cout << "\nEvaluating: integers1 == integers2" << endl;
68
69     if (integers1 == integers2) {
```

```
72     // use square brackets to use the value at location 5 as an rvalue
73     cout << "\nintegers1[5] is " << integers1[5];
74
75     // use square brackets to create lvalue
76     cout << "\n\nAssigning 1000 to integers1[5]" << endl;
77     integers1[5] = 1000;
78     cout << "integers1: ";
79     outputVector(integers1);
80
81     // attempt to use out-of-range subscript
82     try {
83         cout << "\nAttempt to display integers1.at(15)" << endl;
84         cout << integers1.at(15) << endl; // ERROR: out of range
85     }
86     catch (out_of_range& ex) {
87         cerr << "An exception occurred: " << ex.what() << endl;
88     }
89
90
91     // changing the size of a vector
92     cout << "\nCurrent integers3 size is: " << integers3.size() << endl;
93     integers3.push_back(1000); // add 1000 to the end of the vector
94     cout << "New integers3 size is: " << integers3.size() << endl;
95     cout << "integers3 now contains: ";
96     outputVector(integers3);
97 }
98
99 // output vector contents
100 void outputVector(const vector<int>& items) {
101     for (int item : items) {
102         cout << item << " ";
103     }
104     cout << endl;
105 }
106
107
108 // input vector contents
109 void inputVector(vector<int>& items) {
110     for (int& item : items) {
111         cin >> item;
112     }
113 }
```

In questo esempio si definiscono 2 vettori di lunghezza rispettivamente 7 e 10, si visualizza il contenuto e si cerca di accedere ad un elemento al di fuori del vettore; essendo quest'ultima cosa fatta all'interno di un blocco try ed essendo messa la funzione catch allora l'eccezione che si genera viene catturata. Il fatto è però che, ovviamente, non sempre siamo in grado di prevedere la specifica eccezione lanciata da un particolare pezzo del nostro codice.

Es:

```
if(a == b) {
    try {
        while(c < 10) {
            try {
                if(d % 3 == 1) {
                    throw std::runtime_error("!");
                }
            } catch(const bad_alloc &e) {
                ...
            }
        }
    } catch(const runtime_error &e) {
        // after throw, control moves here
        ...
    }
}
```

Nell'esempio c'è un try contenuto all'interno di un altro try contenuto all'interno di un altro try e così via. La parola chiave *throw* genera un nuovo oggetto di tipo *standard exception* (in tal caso `runtime_error` con il messaggio di errore “!”) e l'eccezione che viene lanciata non viene catturata necessariamente dal catch più esterno e quindi quello che si cerca di fare è un “unwinding”: si torna indietro fino alla prima clausola catch in grado di raccogliere l'eccezione generata all'interno di un try. Se non ci fosse nessuna clausola catch in grado di gestire una particolare eccezione, il programma non ha altra scelta che terminare.

LEZIONE 12

PUNTATORI

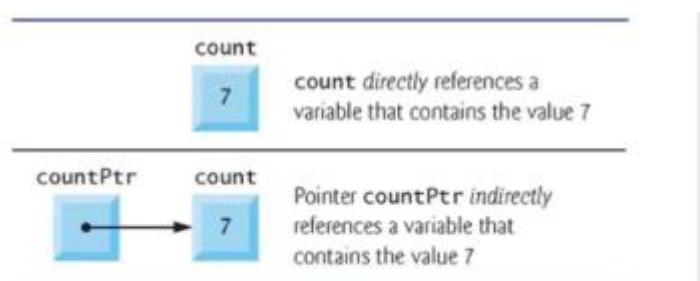
I puntatori sono uno degli aspetti più potenti ma anche più impegnativi del C++.

Lo si è visto anche nell'esame di Calcolatori con l'indirizzamento indiretto che ci sono dei registri che contengono non dei valori ma degli indirizzi di locazioni di memoria. I puntatori li possiamo vedere come variabili che contengono non dei valori ma degli indirizzi, ovvero dei riferimenti ad altre variabili.

I puntatori ci permettono di effettuare passaggi per riferimento ma possono essere utilizzati anche per manipolare strutture dati dinamiche, ovvero che crescono o decrescono durante l'esecuzione del programma. C'è una strettissima relazione tra built-in-arrays e puntatori. In generale gli oggetti array e i vettori sono un po' più sicuri perché effettuano checking dell'indicizzazione e quindi conviene spesso usarli.

Un puntatore in definitiva è una variabile che contiene l'indirizzo di memoria di una variabile che contiene a sua volta un valore specifico.

Es:



```
#include <iostream>

int main(){
    int *countPtr=nullptr;
    int count;
    count = 7;
    countPtr = &count;
    *countPtr = 10;
    std::cout << "Value of count: " << count << "\n";
}
```

countPtr è un puntatore a int; nel definire un puntatore usiamo il simbolo * per indicare che la variabile in questione è un puntatore. Nel nostro caso abbiamo definito il puntatore *countPtr* inizializzandolo all'indirizzo di memoria nullo, fittizio.

Con l'operatore di referenziazione, **&**, assegniamo l'indirizzo al puntatore; con quello di dereferenziazione, *****, assegniamo un valore alla variabile puntata dal puntatore. Ci sono però vari pericoli nascosti dietro l'uso dei puntatori, come ad esempio il fatto che il puntatore può essere istanziato dinamicamente in modo di puntare dinamicamente a varie aree di memoria accedendo direttamente a queste aree di memoria.

Una variabile puntatore la si rappresenta tipicamente con una freccia che va dalla variabile puntatore alla variabile di cui il puntatore è riferimento.

La dichiarazione va quindi fatta così: (supp. tipo intero)

`int* countPtr;`

ovviamente i puntatori possono puntare a qualsiasi tipo di oggetto, sia base che definito da noi.

Bisogna sempre inizializzare un puntatore perché potrebbe puntare altrimenti a qualsiasi area di memoria e nel momento in cui lo si dereferenzia potrebbe venirsi a generare un errore. Nelle prime versioni del C++ un puntatore che non punta ancora a niente veniva inizializzato a NULL od a 0. Nelle nuove versioni del C++ si usa invece la costante *nullptr*. L'inizializzazione del puntatore è importantissima perché altrimenti si rischia di puntare ad aree di memoria non accessibili dall'utente.

& = referenziazione, restituisce/assegna l'indirizzo del puntatore;

***** = dereferenziazione, restituisce il valore della variabile puntata dal puntatore. Questo simbolo viene usato, come sappiamo, anche per la moltiplicazione: in generale ha precedenza il simbolo di dereferenziazione.

PASSAGGIO DEI PARAMETRI AD UNA FUNZIONE

Ci sono 3 modi per passare dei parametri ad una funzione in C++:

- Passaggio per valore;
- Passaggio per riferimento con parametri indirizzi;
- Passaggio per riferimento con parametri puntatori.

Nel primo caso non si passa una variabile ma si passa una copia di essa alla funzione. Nel secondo caso si utilizza “**&**” per passare l'indirizzo del parametro alla funzione, cosicché essa possa operare direttamente sul parametro e non su di una copia. Nel terzo caso si utilizza “*****” per passare alla funzione un puntatore al parametro. Questo fa sì che venga passato l'indirizzo del parametro e quindi il parametro può essere modificato dalla funzione; tuttavia in realtà viene effettuata una copia locale

della variabile nella funzione. In tal caso però la variabile copiata è l'indirizzo del parametro, e non il parametro!

Vediamo la differenza tra passaggio per valore e per riferimento con puntatore.

Esempio:

```
1 // Fig. 8.6: fig08_06.cpp
2 // Pass-by-value used to cube a variable's value.
3 #include <iostream>
4 using namespace std;
5
6 int cubeByValue(int); // prototype
7
8 int main() {
9     int number(5);
10
11    cout << "The original value of number is " << number;
12    number = cubeByValue(number); // pass number by value to cubeByValue
13    cout << "\nThe new value of number is " << number << endl;
14 }
15
16 // calculate and return cube of integer argument
17 int cubeByValue(int n) {
18     return n * n * n; // cube local variable n and return result
19 }
```

Pass by reference

\leftarrow Errore:
“Pass by value”


```
2 // Pass-by-reference with a pointer argument used to cube a
3 // variable's value.
4 #include <iostream>
5 using namespace std;
6
7 void cubeByReference(int*); // prototype
8
9 int main() {
10     int number(5);
11
12    cout << "The original value of number is " << number;
13    cubeByReference(&number); // pass number address to cubeByReference
14    cout << "\nThe new value of number is " << number << endl;
15 }
16
17 // calculate cube of *nPtr; modifies variable number in main
18 void cubeByReference(int* nPtr) {
19     *nPtr *= *nPtr * *nPtr; // cube *nPtr
20 }
```

Pass by reference
with pointers

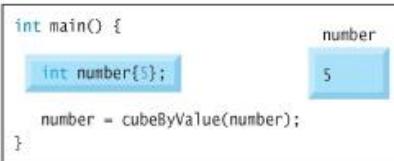
The original value of number is 5
The new value of number is 125

Come avviene il passaggio per riferimento con puntatori? Quello per valore porta ad una copia dei parametri in ingresso; quando usiamo i puntatori, invece, passiamo per riferimento ma vengono comunque effettuate delle copie locali, esattamente come nel passaggio per valore; si tratta però delle copie degli indirizzi delle variabili passate che costituiscono poi i contenuti dei puntatori che vengono definiti nella funzione localmente. Quindi di fatto il passaggio per riferimento con i puntatori è molto più simile al passaggio per valore rispetto al passaggio per riferimento con indirizzo! Questo, appunto, perché vengono create delle variabili locali per contenere gli indirizzi delle variabili passate con i puntatori.

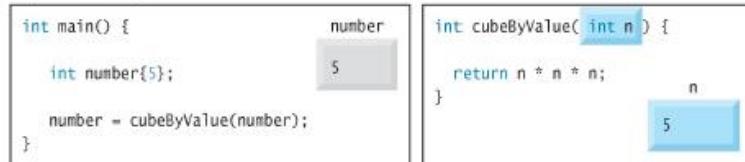
Però, da un altro punto di vista, i passaggi per riferimento sono simili poiché entrambi permettono di modificare direttamente i parametri in ingresso (se non antecediamo *const*).

PASSAGGIO PER VALORE

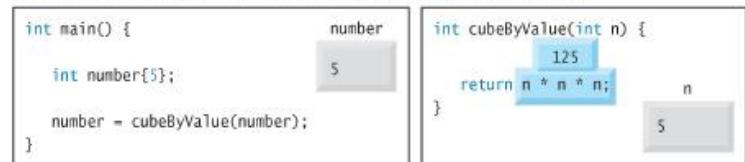
Step 1: Before main calls cubeByValue:



Step 2: After cubeByValue receives the call:



Step 3: After cubeByValue cubes parameter n and before cubeByValue returns to main:

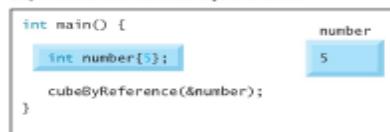


Step 4: After cubeByValue returns to main and before assigning the result to number. Step 5: After main completes the assignment to number:

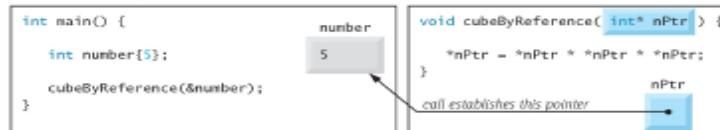


PASSAGGIO PER RIFERIMENTO CON PUNTATORE

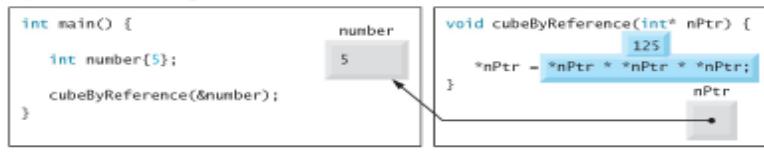
Step 1: Before main calls cubeByReference:



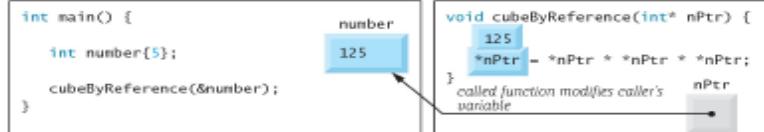
Step 2: After cubeByReference receives the call and before *nPtr is cubed:



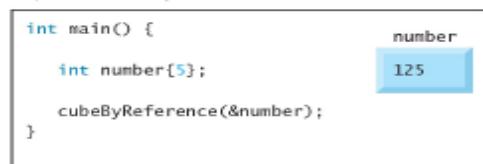
Step 3: Before *nPtr is assigned the result of the calculation $5 * 5 * 5$:



Step 4: After *nPtr is assigned 125 and before program control returns to main:



Step 5: After cubeByReference returns to main:



BUILT-IN ARRAYS

C'è una relazione molto stretta tra i built-in-arrays del C o C++ (che non sono le classi array che abbiamo visto). I built-in arrays sono strutture dati *fixed size*. La dichiarazione di un built-in array la si fa così:

type arrayName[arraySize]

Il compilatore riserva lo spazio appropriato di memoria all'atto della dichiarazione.

La dimensione deve essere un intero, mentre il tipo può essere uno qualunque tra i tipi fondamentali, ma anche non; quindi ad esempio può essere anche un oggetto di una certa classe.

Per accedere ad un elemento dell'array si può usare:

arrayName[i],

dove *i* è l'indice che individua la posizione dell'elemento a cui si vuole accedere.

Possiamo inizializzare gli elementi di un built-in array con una lista di inizializzatori; ad esempio:

```
int n[5]{50,20,30,10,40};
```

Se invece scrivessimo:

```
int n[5]{50,20};
```

allora staremmo inizializzando a 0 gli ultimi 3 elementi. Allo stesso modo i bool vengono istanziati a false, gli array di puntatori a nullptr e gli oggetti con i costruttori di default.

Ovviamente se inizializzassimo 6 elementi in un array che ne contiene 5 otterremmo un errore di compilazione, poiché il compilatore se ne accorgerebbe.

Quando inizializziamo con una lista di inizializzatori possiamo anche evitare di specificare la lunghezza e lasciare [] vuota.

Il legame stretto tra puntatori e built-in arrays sta nel fatto che una variabile built-in array non è altro che un puntatore al primo elemento dell'array e quindi di fatto:

arrayName ↔ &arrayName[0].

Allora quando passiamo un array built-in ad una funzione non c'è bisogno di prenderne l'indirizzo, poiché il nome dell'array è già un puntatore al primo elemento dell'array, e quindi se passiamo un built-in array la funzione ha automaticamente il privilegio di cambiare gli elementi dell'array, a meno che non si faccia precedere al parametro built-in array la parola *const*.

Ad esempio se vogliamo passare un built-in array per effettuare una somma di elementi allora o si può usare:

```
int sumElements(const int values[], const size_t  
    numberOfElements)
```

o, equivalentemente:

```
int sumElements(const int* values, const size_t  
    numberOfElements)
```

Notiamo che, a differenza delle classi array e vector, con i built-in arrays c'è bisogno di specificare la dimensione dell'array, poiché non c'è alcuna funzione del tipo size().

Però la cosa interessante è che per il compilatore non fa alcuna differenza che si passi ad una funzione un puntatore o un built-in array.

In realtà noi abbiamo già visto i puntatori in alcune occasioni, ad esempio quando abbiamo visto la funzione sort(A.begin(), A.end()); della libreria standard. Infatti A.begin() e A.end() sono puntatori al primo e all'ultimo elemento e nello specifico sono dei puntatori a degli iterators, che poi vedremo.

I built-in arrays hanno diverse limitazioni:

- 2 built-in arrays non possono essere confrontati direttamente ma bisogna usare un loop per confrontare ogni elemento;
- Non possono essere assegnati l'uno all'altro; ovvero se A e B sono array non posso scrivere A=B;
- Non conoscono la loro dimensione;
- Non fanno un checking automatico degli indici e questo può portare ad errori fatali nel programma.

Tuttavia a volte è necessario usare i built-in arrays.

ARGOMENTI DEL MAIN

Ad esempio, noi fino ad ora abbiamo sempre considerato la funzione main con 0 argomenti. In realtà può avere degli argomenti che sono gli argomenti sulla linea di comando con cui abbiamo invocato il nostro programma. Gli argomenti sulla linea di comando sono individuati da un intero **argc**, che rappresenta il numero di argomenti, ed un array di puntatori a caratteri **argv** che rappresenta quali sono gli argomenti. Il nome del programma sarà sempre argv[0], ad esempio.

Es:

```
1 // demostrante command line arguments
2
3 #include <iostream>
4
5 int main(int argc, char** argv) {
6     std::cout << "This is the program " << argv[0] << "\n";
7     std::cout << "You specified " << argc << " arguments:" << std::endl;
8     std::cout << "Hello ";
9     for (int i = 1; i < argc; ++i) {
10         std::cout << argv[i] << " ";
11     }
12     std::cout << "\n";
13
14 }
```

Se io vado a eseguire il programma (`./commandLineArguments.exe`) da linea di comando specificando ulteriori argomenti, come ad esempio:

```
./commandLineArguments Roberto Torpedine
```

Allora il programma mostrerà:

```
This is program ./commandLineArguments
You specified 3 arguments:
Hello Roberto Torpedine
```

Questa cosa è interessante perché ad esempio posso passare come argomento il nome di un file, e poi lo vedremo meglio; in generale una cosa del genere serve a non interagire necessariamente con l'utente da tastiera e quindi ad esempio se voglio fare un programma che stampa i primi n numeri anziché chiedere all'utente quanto vale n l'utente affianca il parametro n (ad esempio 10) al comando di esecuzione del programma.

In generale ci sono 3 modi per “leggere” un file:

- 1) Definire una variabile e porla uguale a nomeFile;
- 2) Leggere il nome del file da tastiera chiedendo all'utente di immetterlo;
- 3) Come argomento di un programma.

PUNTATORI PT.2

Molto spesso quando passiamo un puntatore ad una funzione diamo alla funzione il privilegio di cambiarne il contenuto. Tuttavia nel passaggio dei parametri dobbiamo sempre utilizzare il *principio del minor privilegio*: bisogna dare ad una funzione sempre il minor privilegio possibile, o meglio quello necessario alla funzione ma niente

di più. Ci sono 4 modi per passare un puntatore ad una funzione (elenco dal privilegio massimo a quello minimo)

1) Puntatore non costante ad una variabile non costante.

La funzione può cambiare il contenuto della variabile puntata dal puntatore ed il puntatore stesso.

Es:

int* countPtr;

2) Puntatore non costante ad un dato costante.

La funzione può cambiare il puntatore ma non il dato a cui punta poiché definito come costante.

Es:

const int* xPtr;

Vediamo un codice di esempio:

```
1 // -----
2 // Attempting to modify data through a
3 // nonconstant pointer to constant data.
4
5 void f(const int*); // prototype
6
7 int main() {
8     int y{0};
9
10    f(&y); // f will attempt an illegal modification
11 }
12
13 // constant variable cannot be modified through xPtr
14 void f(const int* xPtr) {
15     *xPtr = 100; // error: cannot modify a const object
16 }
```

GNU C++ compiler error message:

```
fig08_10.cpp: In function 'void f(const int*)':
fig08_10.cpp:17:12: error: assignment of read-only location '* xPtr'
```

Quando passo un puntatore ad una funzione in questa maniera posso dereferenziare a destra ma non a sinistra, nel senso che posso assegnare il valore della variabile puntata dal puntatore ad un'altra variabile ma non posso cambiarlo. Passare un puntatore in questa maniera serve a passare una struttura dati pesante senza copiarla ogni volta ma non dando il privilegio alla funzione di cambiarne il contenuto. Quello che si può fare con questo tipo di passaggio è cambiare la posizione in memoria della variabile puntata dal puntatore.

3) Puntatore costante a un dato non costante.

La funzione ha il privilegio di cambiare il valore della variabile puntata dal puntatore ma il puntatore deve puntare sempre alla stessa locazione di memoria.

Questo vuol dire che la dereferenziazione del puntatore può avvenire a sinistra di un'assegnazione stavolta.

Es:

int* const ptr{&x};

in questo modo ptr è un puntatore costante ad intero.

Es:

```
2 // Attempting to modify a constant pointer to nonconstant data.
3
4 int main() {
5     int x, y;
6
7     // ptr is a constant pointer to an integer that can be modified
8     // through ptr, but ptr always points to the same memory location.
9     int* const ptr{&x}; // const pointer must be initialized
10
11    *ptr = 7; // allowed: *ptr is not const
12    ptr = &y; // error: ptr is const; cannot assign to it a new address
13 }
```

Microsoft Visual C++ compiler error message:

```
'ptr': you cannot assign to a variable that is const
```

- Puntatore costante a dato costante.

la situazione in cui la funzione non ha alcun privilegio: non può modificare l'area di memoria puntata dal puntatore né dereferenziare il puntatore a sinistra.

Es:

const int* const ptr{&x};

in questa maniera diciamo che ptr è un puntatore costante ad un intero costante.

Programma di esempio:

```
2 // Attempting to modify a constant pointer to constant data.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int x{5}, y;
8
9     // ptr is a constant pointer to a constant integer.
10    // ptr always points to the same location; the integer
11    // at that location cannot be modified.
12    const int* const ptr{&x};
13
14    cout << *ptr << endl;
15
16    *ptr = 7; // error: *ptr is const; cannot assign new value
17    ptr = &y; // error: ptr is const; cannot assign to it a new address
18 }
```

Xcode LLVM compiler error message:

```
Read-only variable is not assignable
Read-only variable is not assignable
```

Operatore **sizeof**: determina la dimensione in byte del built-in array o di qualsiasi variabile.

Es:

```
1 // Fig. 8.13: fig08_13.cpp
2 // Sizeof operator when used on a built-in array's name
3 // returns the number of bytes in the built-in array.
4 #include <iostream>
5 using namespace std;
6
7 size_t getSize(double*); // prototype
8
9 int main() {
10    double numbers[20]; // 20 doubles; occupies 160 bytes on our system
11
12    cout << "The number of bytes in the array is " << sizeof(numbers);
13
14    cout << "\nThe number of bytes returned by getSize is "
15    << getSize(numbers) << endl;
16 }
17
18 // return size of ptr
19 size_t getSize(double* ptr) {
20    return sizeof(ptr);
21 }
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

`sizeof` è una funzione molto utile quando dobbiamo allocare dinamicamente strutture dati. Ovviamente quando la invochiamo sul puntatore ci restituisce il numero di byte necessari per rappresentare il puntatore (e quindi 4 byte nel nostro esempio perché è un puntatore che punta a un `double`), quando la invochiamo su un array ci restituisce la dimensione in byte di quell'array. In particolare per ottenere la lunghezza dell'array basta fare:

lunghezza di array = `sizeof(array) / sizeof(array[0])`.

Allora ad esempio potremmo scoprire quanti byte vengono utilizzati per rappresentare i tipi sul nostro sistema:

```
1 #include <iostream>
2 int main(){
3    int array[20];
4    int* ptr{array};
5    std::cout << "char :           " << sizeof(char) << "\n";
6    std::cout << "short int:      " << sizeof(short int) << "\n";
7    std::cout << "int:            " << sizeof(int) << "\n";
8    std::cout << "long int:       " << sizeof(long int) << "\n";
9    std::cout << "float:          " << sizeof(float) << "\n";
10   std::cout << "double:         " << sizeof(double) << "\n";
11   std::cout << "long double:    " << sizeof(long double) << "\n";
12   std::cout << "array:          " << sizeof(array) << "\n";
13   std::cout << "ptr:            " << sizeof(ptr) << "\n";
14
15
16 }
```

File eseguibile del prof:

```
(base) Micheles-iMac-2:misc ceccarelli$ ./size
char :           1
short int:      2 *
int          4
long int       8
float:         4
double:        8
long double   16
array          80
ptr            8
```

Sui puntatori possono essere fatte operazioni aritmetiche come incremento e decremento. Attenzione con queste operazioni: se io faccio un incremento di 1 in un array in realtà punto non 1 dopo ma 1*numero di byte usato per rappresentare il tipo dell'array. Ad esempio, supponiamo di avere un array di interi e che il nostro sistema usi 4 byte per rappresentare un intero.

- int v[5] è il nostro array ;
- v[0] ha indirizzo di memoria 3000;
- int* vPtr{v} è la dichiarazione del puntatore vPtr che punta all'array v;
(int* vPtr{&v[0]})

Se incrementiamo di 2 vPtr ($vPtr+=2$) quello che succede è che il puntatore avanza di 8 in realtà, ovvero va all'indirizzo 3008 e non 3002 poiché automaticamente viene moltiplicato il numero di byte necessari per rappresentare i dati dell'array. Questo è molto comodo perché rende anche più facile scorrere un array. Basta fare un ciclo e fare $ptr++$ ad ogni iterazione. In realtà scorrere un array con incremento e decremento richiede comunque delle operazioni di moltiplicazione interne che su strutture pesanti possono incidere non poco, e questo non ci piace.

Nell'aritmetica dei puntatori in realtà rientrano anche altre operazioni come quelle dirette tra puntatori. Ad esempio possiamo fare la differenza tra 2 puntatori.

Es:

$vPtr=3000;$

$v2Ptr=3008;$

$x=v2Ptr-vPtr=2.$

Questo può servire per sapere quanto sono distanti 2 elementi in un array ad esempio.

Ancora una volta, questa operazione tiene conto dei byte necessari per la rappresentazione dei valori.

Quest'aritmetica tra puntatori ha senso solo se si sta facendo una differenza tra 2 puntatori che puntano allo stesso built-in array. Sottrarre 2 puntatori che puntano a 2 array differenti è un errore logico, nel senso che è una cosa priva di senso ma il compilatore non se ne accorge e non dà errore.

Nel convertire il valore del puntatore a destra dell'assegnazione al tipo a sinistra dell'assegnazione viene usato un operatore di cast. L'unico caso in cui non è richiesto è quando si definisce un puntatore a void, e questo è utile quando non si conosce ancora il tipo che si userà per quel puntatore.

I puntatori, poi, possono essere confrontati tra di loro, per uguaglianza ad esempio, ed in questo caso ancora una volta questa operazione ha senso solo se effettuata sullo stesso built-in array. E valgono poi anche gli operatori di relazione di > e <.

In questi esempi ora vediamo i legami stretti che ci sono tra puntatori e built-in array e i modi diversi per assegnare variabili o indirizzi.

- Pointers can be used to do any operation involving array subscripting.

```
// create 5-element int array b; b is a const pointer
int b[5];
// create int pointer bPtr, which isn't a const pointer
int* bPtr;
• We can set bPtr to the address of the first element in the built-in
array b with the statement
// assign address of built-in array b to bPtr
bPtr = b;
• This is equivalent to assigning the address of the first element as
follows:
// also assigns address of built-in array b to bPtr
bPtr = &b[0];
```

Quindi le ultime 2 istruzioni dell'esempio sono equivalenti e questo perché il nome dell'array coincide con il nome della prima locazione di memoria.

Anche nell'accesso ad un array puntatori e built-in array sono molto legati.

- Built-in array element b[3] can alternatively be referenced with the pointer expression *(bPtr + 3)
 - The 3 in the preceding expression is the offset to the pointer.
 - This notation is referred to as pointer/offset notation.
 - The parentheses are necessary, because the precedence of * is higher than that of +.
 - Just as the built-in array element can be referenced with a pointer expression, the address &b[3]
 - can be written with the pointer expression
bPtr + 3
 - For example, the expression
*(b + 3)
 - also refers to the element b[3].
 - In general, all subscripted built-in array expressions can be written with a pointer and an offset.
 - Pointers can be subscripted exactly as built-in arrays can.
 - For example, the expression
bPtr[1]
 - refers to b[1];

Per accedere al terzo elemento dell'array b o uso b[3] o uso *(bPtr+3), sfruttando l'aritmetica dei puntatori e la dereferenziazione che serve ad accedere effettivamente all'elemento calcolato tra parentesi tonde.

Se invece mi interessa accedere all'indirizzo del terzo elemento uso l'operatore di referenziazione → &b[3], oppure mi riferisco al puntatore e sommo l'offset rispetto all'indirizzo del primo elemento → bPtr+3.

Importante

Dobbiamo capire una cosa importante: se io definisco un array b[5] di interi, ho che la variabile b è l'indirizzo del primo elemento b[0] e quindi è anch'essa un puntatore. Solo che se definisco un puntatore bPtr=b ho che bPtr lo posso muovere e spostare mentre b no; b è un puntatore costante. Quindi nel programma potrò fare puntare bPtr ad un altro array c, ma non potrò fare lo stesso con b. Questa è l'unica differenza che c'è tra il nome del built-in array ed un semplice puntatore, e ovvero che un puntatore può stare a sinistra di un'assegnazione, il nome dell'array no.

Programma esempio per l'uso delle varie notazioni:

```
1 // subscript.cpp
2 // Using subscripting and pointer notations with built-in arrays.
3 #include <iostream>
4 using std::cout;
5
6 int main() {
7     int b[]{10, 20, 30, 40}; // create 4-element built-in array b
8     int* bPtr{b}; // set bPtr to point to built-in array b
9
10    // output built-in array b using array subscript notation
11    cout << "Array b displayed with:\n\nArray subscript notation\n";
12
13    for (size_t i{0}; i < 4; ++i) {
14        cout << "b[" << i << "] = " << b[i] << '\n';
15    }
16
17    // output built-in array b using array name and pointer/offset notation
18    cout << "\nPointer/offset notation where "
19        << "the pointer is the array name\n";
20
21    for (size_t offset1{0}; offset1 < 4; ++offset1) {
22        cout << "*(" << b << " + " << offset1 << ") = " << *(b + offset1) << '\n';
23    }
24
25    // output built-in array b using bPtr and array subscript notation
26    cout << "\nPointer subscript notation\n";
27
28    for (size_t j{0}; j < 4; ++j) {
29        cout << "bPtr[" << j << "] = " << bPtr[j] << '\n';
30    }
31
32    cout << "\nPointer/offset notation\n";
33
34    // output built-in array b using bPtr and pointer/offset notation
35    for (size_t offset2{0}; offset2 < 4; ++offset2) {
36        cout << "*(" << bPtr << " + " << offset2 << ") = "
37            << *(bPtr + offset2) << '\n';
38    }
39
```

```
./subscript
Array b displayed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Anche le stringhe possono essere ottenute attraverso dei built-in array e quindi attraverso puntatori a char. La cosa particolare è che una stringa è un array di caratteri che termina con un carattere speciale di fine stringa, ovvero '\0'.

LEZIONE 13

Da ora in poi vedremo più nello specifico le caratteristiche della programmazione orientata agli oggetti.

CLASSI & OGGETTI

Dovremo cercare di separare il *cosa* dal *come*: il cosa è composto da quali servizi un certo oggetto mette a disposizione, e questo lo mettiamo all'interno dell'header file (.h); il come è dato da come questi servizi sono realizzati. Dobbiamo quindi separare l'interfaccia (che mettiamo nell'header file) dall'implementazione. Allora ad esempio se voglio mettere a disposizione una classe fatta da me a diversi utenti allora sicuramente dovrò mettere a loro disposizione l'header file in cui sono spiegate tutte le funzionalità della classe, e quello che potrei voler fare è mascherare l'implementazione delle funzionalità.

Es:

Time class

```
1 // Time.h
2 // Time class definition.
3 // Member functions are defined in Time.cpp
4 #include <iostream>
5
6 // prevent multiple inclusions of header
7 ifndef TIME_H
8 define TIME_H
9
10 // Time class definition
11 class Time {
12 public:
13     void setTime(int, int, int); // set hour, minute and second
14     std::string toUniversalString() const; // 24-hour time format string
15     std::string toStandardString() const; // 12-hour time format string
16 private:
17     unsigned int hour{0}; // 0 - 23 (24-hour clock format)
18     unsigned int minute{0}; // 0 - 59
19     unsigned int second{0}; // 0 - 59
20 };
21
22 endif
23
24 // Program to test class Time.
25 // NOTE: This file must be compiled with Time.cpp.
26 #include <iostream>
27 #include <stdexcept> // for invalid_argument exception class
28 #include "Time.h" // include definition of class Time from Time.h
29 using namespace std;
30
31 // displays a Time in 24-hour and 12-hour formats
32 void displayTime(const string& message, const Time& time) {
33     cout << message << "\nUniversal time: " << time.toUniversalString()
34     << "\nStandard time: " << time.toStandardString() << "\n\n";
35 }
36
37 int main() {
38     Time t; // instantiate object t of class Time
39
40     displayTime("Initial time:", t); // display t's initial value
41     t.setTime(13, 27, 61); // change time
42     displayTime("After setTime:", t); // display t's new value
43
44     // attempt to set the time with invalid values
45     try {
46         t.setTime(99, 99, 99); // all values out of range
47     } catch (invalid_argument& e) {
48         cout << "Exception: " << e.what() << "\n\n";
49     }
50
51     // display t's value after attempting to set an invalid time
52     displayTime("After attempting to set an invalid time:", t);
53 }
```

```
1 // Time.cpp
2 // Time class member-function definitions.
3 #include <iomanip> // for setw and setfill stream manipulators
4 #include <stdexcept> // for invalid_argument exception class
5 #include <iostream> // for ostreamstream class
6 #include <iostream>
7 #include "Time.h" // include definition of class Time from Time.h
8
9 using namespace std;
10
11 // set new Time value using universal time
12 void Time::setTime(int h, int m, int s) {
13     // validate hour, minute and second
14     if ((h >= 0 && h < 24) && (m >= 0 && m < 60) && (s >= 0 && s < 60)) {
15         hour = h;
16         minute = m;
17         second = s;
18     } else {
19         throw invalid_argument(
20             "hour, minute and/or second was out of range");
21     }
22 }
23
24
25 // return Time as a string in universal-time format (HH:MM:SS)
26 string Time::toUniversalString() const {
27     ostringstream output;
28     output << setw(2) << setw(2) << hour << ":"
29     << setw(2) << minute << ":" << setw(2) << second;
30     return output.str(); // returns the formatted string
31 }
32
33 // return Time as string in standard-time format (HH:MM:SS AM or PM)
34 string Time::toStandardString() const {
35     ostringstream output;
36     output << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
37     << setw(2) << minute << ":" << setw(2)
38     << second << (hour < 12 ? " AM" : " PM");
39     return output.str(); // returns the formatted string
40 }
41
42
43 #Makefile for the testTime program
44 testTime: testTime.o Time.o
45     g++ -o testTime testTime.o Time.o
46 testTime.o: Time.h testTime.cpp
47     g++ -c testTime.cpp -std=c++11 -Wall -pedantic
48 Time.o: Time.h Time.cpp
49     g++ -c Time.cpp -std=c++11 -Wall -pedantic
```

Ho un file Time.h in cui c'è la definizione dell'interfaccia (cosa) ed un file Time.cpp c'è l'implementazione delle funzionalità che il nostro oggetto mette a disposizione

(come). C'è poi una funzione client che utilizza le funzioni messe a disposizione dall'oggetto time.

La classe **Time.h** ha 3 variabili membro di tipo privato che rappresentano l'orario splittato in ore, minuti e secondi. Questa classe mette poi a disposizione 3 servizi pubblici, ovvero 3 funzioni membro di tipo *public*, una per istanziare le variabili membro e le altre due per stampare l'orario rispettivamente su 24 h e su 12 h con a.m. e p.m.

Nel file .h c'è un *include guard*, ovvero una guarda da inclusione. Infatti se il nostro progetto include diversi file sorgenti, più di un file sorgente potrebbe includere il file Time.h e quindi correremmo il rischio di importare o diversi file Time.h che si trovano in diverse parti o includere più volte lo stesso file .h . Per evitare questa evenienza, prima della definizione della classe si mette un if-else particolare.

```
1 // Time.h
2 // Time class definition.
3 // Member functions are defined in Time.cpp
4 #include <iostream>
5
6 // prevent multiple inclusions of header
7 #ifndef TIME_H
8 #define TIME_H
9
10 // Time class definition
11 class Time {
12 public:
13     void setTime(int, int, int); // set hour, minute and second
14     std::string toUniversalString() const; // 24-hour time format string
15     std::string toStandardString() const; // 12-hour time format string
16 private:
17     unsigned int hour(0); // 0 - 23 (24-hour clock format)
18     unsigned int minute(0); // 0 - 59
19     unsigned int second(0); // 0 - 59
20 };
21
22 #endif
```

Cosa vogliono dire le righe 7,8 e 22? Racchiudono un if-else. Alle righe 7-8 si sta dicendo “se non è stato definito TIME_H (#ifndef) allora definisci TIME_H” (#define), e dopo c'è la definizione vera e propria della classe; alla riga 22 finisce questo if con #endif. Questo vuol dire che se è già definita questa classe allora non viene ridefinita e non si entra proprio nella riga 11 alla definizione della classe; si salta direttamente ad #endif.

Grazie a questo controllo, quindi, se in un progetto un file sorgente include la classe, e quindi l'identificativo (TIME_H) risulta già definito, e poi un altro file sorgente la include di fatto l'identificativo della classe già risulterà definito e quindi non viene ridefinito: il preprocessore ignora il codice tra #ifndef e #endif.

Ora passiamo al file **Time.cpp**. il file in cui implementiamo le funzioni definite in Time.h e quindi questo file deve essere necessariamente compilato ed include l'header file.h.

```

1 // Time.cpp
2 // Time class member-function definitions.
3 #include <iomanip> // for setw and setfill stream manipulators
4 #include <stdexcept> // for invalid_argument exception class
5 #include <sstream> // for ostringstream class
6 #include <string>
7 #include "Time.h" // include definition of class Time from Time.h
8
9 using namespace std;
10
11 // set new Time value using universal time
12 void Time::setTime(int h, int m, int s) {
13     // validate hour, minute and second
14     if ((h >= 0 && h < 24) && (m >= 0 && m < 60) && (s >= 0 && s < 60)) {
15         hour = h;
16         minute = m;
17         second = s;
18     }
19     else {
20         throw invalid_argument(
21             "hour, minute and/or second was out of range");
22     }
23 }
24
25 // return Time as string in universal-time format (HH:MM:SS)
26 string Time::toUniversalString() const {
27     ostringstream output;
28     output << setfill('0') << setw(2) << hour << ":"
29         << setw(2) << minute << ":" << setw(2) << second;
30     return output.str(); // returns the formatted string
31 }
32
33 // return Time as string in standard-time format (HH:MM:SS AM or PM)
34 string Time::toStandardString() const {
35     ostringstream output;
36     output << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
37         << setfill('0') << setw(2) << minute << ":" << setw(2)
38         << second << (hour < 12 ? " AM" : " PM");
39     return output.str(); // returns the formatted string
40 }

```

Sono implementate le funzioni *setTime*, *toUniversalString* e *toStandardString* pubbliche e messe a disposizione dalla classe Time.

setTime restituisce un void, *toUniversalString* e *toStandardString* restituiscono una stringa.

Notiamo la presenza dell'operatore:

::

Quest'operatore si chiama *scope resolution operator*, ovvero operatore di risoluzione di scope, e serve a dire che ciò che segue fa parte di ciò che lo precede ed è quello il suo scope. Quindi nel nostro caso *Time::toStandardString* sta a dire che la funzione *toStandardString* ha come scope la classe Time, ovvero è visibile alla classe Time. Questo vale anche per le altre 2 funzioni. Senza la specifica di scope queste funzioni sarebbero funzioni qualunque e non potrebbero accedere ai dati membro privati o alle funzioni membro della classe Time senza fare riferimento ad uno specifico oggetto della classe.

La funzione *setTime* istanzia le 3 variabili membro facendo anche un check affinché l'istanziamento sia corretto: le ore devono essere comprese tra 0 e 24, i minuti tra 0 e 60 e i secondi tra 0 e 60. Se non avviene ciò allora viene lanciata un'eccezione il cui argomento è il testo da visualizzare che spiega l'errore.

La funzione `toUniversalString` non ha il privilegio di cambiare i dati membro dell'oggetto e quindi viene definita come “const”. Questa funzione restituisce una stringa ed utilizza un tipo di dato che si chiama `ostringstream` (output string stream) che serve per stampare non a video ma in una stringa la variabile dichiarata di questo tipo. Quindi l'orario (ore-minuti-secondi) non viene stampato sullo schermo ma sulla variabile `output` dichiarata come `ostringstream`. Questa variabile al suo interno contiene una funzione membro che si chiama `str` e che è la stringa che è stata istanziata. Quindi la funzione `toUniversalString` ritorna `output.str()`, ovvero la stringa contenuta nella variabile `output`. Per potere usufruire di questo tipo (`ostringstream`) c'è bisogno di includere l'header `<sstream>`. La funzione `setw(i)` serve per dire che si vogliono stampare *i* caratteri, nel nostro caso 2 per le ore, 2 per i minuti e 2 per i secondi. C'è poi un'altra funzione `setfill('i')` è un operatore di formattazione e serve a fare il filling a 0, ovvero mettere degli zeri a sinistra se il nostro valore può essere stampato con meno caratteri di quelli indicati. Grazie a queste 2 funzioni la stringa in `output` sarà del tipo: HH:MM:SS.

La stessa cosa la fa la funzione `toStandardString` ma sul formato dell'orario di 12 ore. Viene utilizzato l'operatore condizionale, una sorta di if-else che abbiamo già incontrato e che serve per stampare a video “12” se è vera la condizione che precede 12, ovvero se l'ora è uguale a 0 o 12; altrimenti sarà dato dalle ore modulo (%) 12.

Abbiamo visto il “cosa”, il “come” ed ora vediamo la funzione **main(.cpp)** che è la funzione client che utilizza la nostra classe; nel nostro caso l'abbiamo chiamata `testTime.cpp`.

```
1 // PROGRAM TO TEST CLASS TIME.
2 // NOTE: This file must be compiled with Time.cpp.
3 #include <iostream>
4 #include <stdexcept> // for invalid_argument exception class
5 #include "Time.h" // include definition of class Time from Time.h
6 using namespace std;
7
8 // displays a Time in 24-hour and 12-hour formats
9 void displayTime(const string& message, const Time& time) {
10     cout << message << "\nUniversal time: " << time.toUniversalString()
11     << "\nStandard time: " << time.toStandardString() << "\n\n";
12 }
13
14 int main() {
15     Time t; // instantiate object t of class Time
16
17     displayTime("Initial time:", t); // display t's initial value
18     t.setTime(13, 27, 6); // change time
19     displayTime("After setTime:", t); // display t's new value
20
21     // attempt to set the time with invalid values
22     try {
23         t.setTime(99, 99, 99); // all values out of range
24     }
25     catch (invalid_argument& e) {
26         cout << "Exception: " << e.what() << "\n\n";
27     }
28
29     // display t's value after attempting to set an invalid time
30     displayTime("After attempting to set an invalid time:", t);
31 }
```

La funzione *displayTime* non fa altro che visualizzare un messaggio e quindi prende in ingresso una stringa ed un oggetto di tipo *Time* e stampa a video il messaggio e invoca poi *toUniversalString* e *toStandardString* sull'oggetto ricevuto.

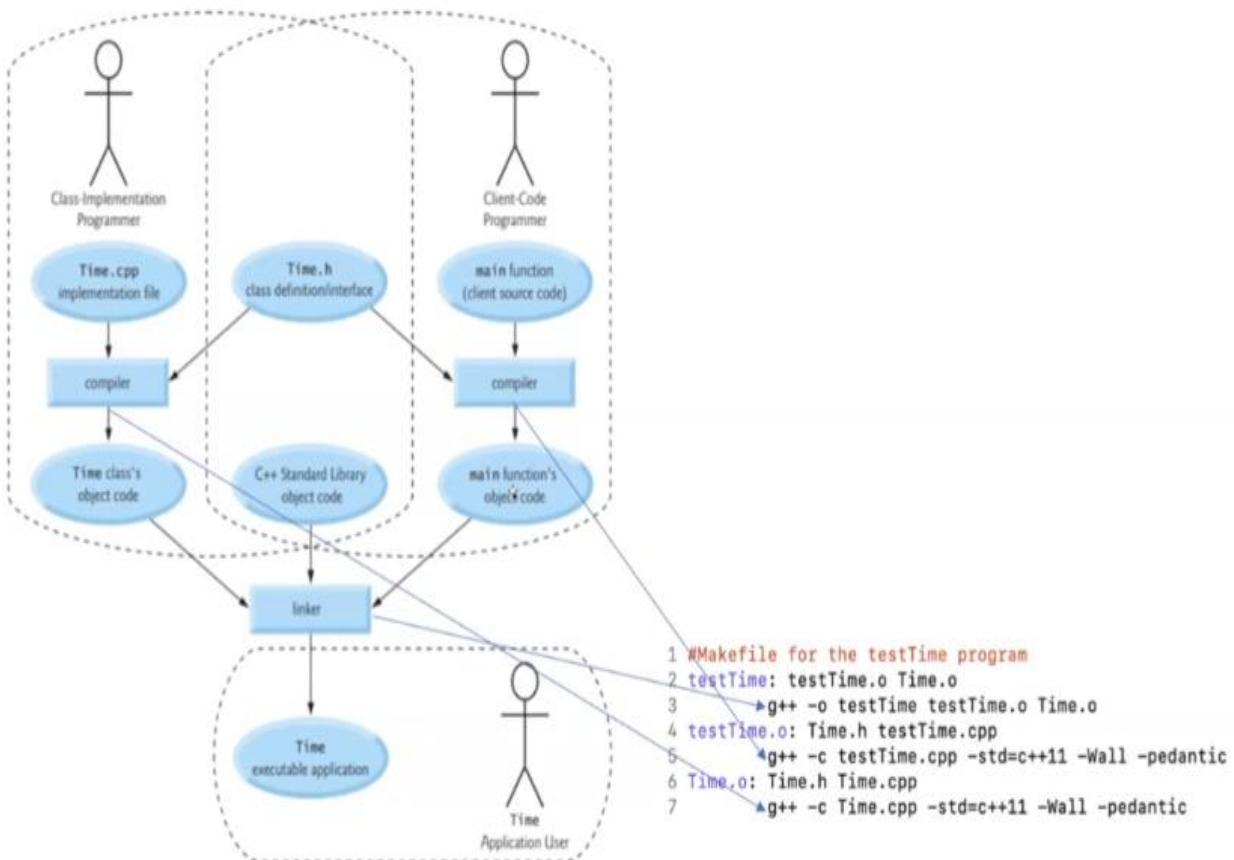
Poi invochiamo la funzione *setTime* per impostare l'ora e richiamiamo la funzione *displayTime* per mostrare a video l'ora aggiornata.

Poi cerchiamo di settare il tempo con quantità non consentite e quindi facciamo un *setTime* all'interno di un *try*, che serve per generare un'eccezione. La clausola *catch* del *try* cattura l'eccezione e visualizza il messaggio dell'eccezione. A seguito di questo tentativo il valore della variabile *Time t* continua ad essere l'ultimo corretto istanziato.

Una volta creato il tipo *Time* sarà consentito fare tutte le operazioni che si fanno con i tipi base. Quindi si possono costruire array di *Time*, istanziare oggetti di tipo *Time* ovviamente, fare referenze (&) a variabili di tipo *Time* e usare puntatori (*) a variabili di tipo *Time*.

Ora concentriamoci sul **makefile**. Il client ha bisogno di conoscere come utilizzare le funzionalità della classe *Time* ma non deve sapere come queste funzionalità vengono implementate. Quindi è solo il linker che costruisce l'applicazione con il file oggetto del main e il file di libreria che implementa le funzioni definite nella libreria *Time.h*.

Le dipendenze sono:



- per generare il codice di testTime (testTime.o), testTime ha bisogno del sorgente di testTime (testTime.cpp) e dell’interfaccia Time.h;
 - per generare il codice di Time.o c’è bisogno del sorgente Time.cpp e dell’interfaccia Time.h;
 - per generare l’eseguibile si ha bisogno dei codici oggetto di Time e di testTime.
-

Ovviamente, lo abbiamo già detto, per accedere ai dati membro e alle funzioni membro di una certa classe utilizziamo l’operatore “.”. Vediamo un esempio su cosa si può fare con una variabile (oggetto) di tipo Account, dove Account è una classe.

```
Account account; // an Account object
// accountRef refers to an Account object
Account& accountRef{account};
// accountPtr points to an Account object
Account* accountPtr{&account}
• To reference an object's members via a pointer to an object, follow the pointer name by the arrow member-selection operator (->) and the member name, as
in pointerName->memberName.
// call setBalance via the Account object
account.setBalance(123.45);
// call setBalance via a reference to the Account object
accountRef.setBalance(123.45);
// call setBalance via a pointer to the Account object
accountPtr->setBalance(123.45);
```

Possiamo definire un riferimento alla variabile di tipo Account ed un puntatore ad una variabile di tipo Account. Per accedere ai membri della variabile puntata dal puntatore accountPtr si usa l’operatore “ \rightarrow ” che dereferenzia un puntatore ad una struttura per accedere alle funzioni membro ed ai dati membro di un certo oggetto.

COSTRUTTORE PER LA CLASSE TIME

Time.h

```
//Time.h
//Time class containing a constructor with default arguments
#include<string>

#ifndef TIME_H
#define TIME_H

class Time {
public:
    explicit Time(int = 0, int = 0, int = 0); //costruttore di default

    //set functions
    void setTime (int,int,int); //setta ore, minuti e secondi
    void setHour (int); //setta l'ora (dopo la validazione)
    void setMinute (int); //setta i minuti (dopo la validazione)
    void setSecond (int); //setta i secondi (dopo la validazione)

    //get functions
    std::string toUniversalString() const; //24h
    std::string toStandardString() const; //12h

    unsigned int getHour() const;
    unsigned int getMinute() const;
    unsigned int getSecond() const;

private:
    unsigned int hour{0}; // 0 - 23
    unsigned int minute{0}; // 0 - 59
    unsigned int second{0}; // 0 - 59
};

#endif
```

Ogni volta che istanziamo un oggetto della classe *Time* verrà invocato il costruttore di default che nel nostro caso inizializza a 0 le variabili membro dell'oggetto.

Notiamo che abbiamo usato *explicit* anche se abbiamo più argomenti; questo perché c'è anche l'oggetto che si definisce potrebbe avere anche un unico argomento. Questo poi lo vedremo meglio quando vedremo i costruttori di copia.

Time.cpp

```
//Time.cpp
//Member function definitions for class Time

#include<iomanip>
#include<stdexcept>
#include<sstream>
#include<string>
#include "Time.h"

using std::ostringstream;
using std::invalid_argument;
using std::string;
using std::setw;
using std::setfill;

//Costruttore di Time inizializza ogni dato membro
Time::Time(int hour, int minute, int second) {
    setTime(hour, minute, second);
}

//set new Time using universal time
void Time::setTime(int h, int m, int s) {
    setHour(h);
    setMinute(m);
    setSecond(s);
}

//set hour value
void Time::setHour(int h) {
    if (h>=0 && h<24) {
        hour=h;
    }
    else {
        throw invalid_argument("Hour must be 0-23");
    }
}

//set minute value
void Time::setMinute(int m) {
    if (m>=0 && m<59) {
        minute=m;
    }
    else {
        throw invalid_argument("Minute must be 0-59");
    }
}

//set second value
void Time::setSecond(int s) {
    if (s>=0 && s<59) {
        second=s;
    }
    else {
        throw invalid_argument("Second must be 0-59");
    }
}
```

```
        }
    }

//return hour value
unsigned int Time::getHour() const {
    return hour;
}

//return minute value
unsigned int Time::getMinute() const {
    return minute;
}

//return second value
unsigned int Time::getSecond() const {
    return second;
}

//return Time as a string in universal-time format (24h)
string Time::toUniversalString() const {
    ostringstream output;
    output<<setfill('0')<<setw(2)<<getHour()<<":"<<setw(2)<<getMinute()<<":"<<setw(2)<<getSecond();
    return output.str();
}

//return Time as a string in standard-time format (12h)
string Time::toStandardString() const {
    ostringstream output;
    output<<((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12)<<":"<<setfill('0')<<setw(2)<<getMinute()<<":"<<setw(2)<<getSecond()<<(hour<12 ? " AM" : " PM");
    return output.str();
}
```

testTime.cpp

```
//testTime.cpp
//constructor with default arguments
#include<iostream>
#include<stdexcept>
#include"Time.h"
using std::cout;
using std::invalid_argument;
using std::endl;
using std::cerr;
using std::string;

void displayTime(const string& message, const Time& time) {
    cout<<message<<"\nUniversal time: "<<time.toUniversalString()<<"\nStandard time
: "<<time.toStandardString()<<"\n\n";
}

int main () {
    Time t1;
    Time t2{2};
    Time t3{21,34};
    Time t4{12,25,42};

    cout<<"Constructed with:\n\n";
    displayTime("t1: all argument defaulted", t1);
    displayTime("t2: hour specified; minute and second defaulted",t2);
    displayTime("t3: hour and minute specified; second defaulted", t3);
    displayTime("t4: hour, minute and second specified", t4);

//tentativo di inizializzare t5 con valori non consentiti
    try{
        Time t5{27, 74, 99};
    }
    catch (invalid_argument& e){
        cerr<< "Exception while initializing t5: "<<e.what()<<endl;
    }
}
```

Makefile

```
testTime: testTime.o Time.o
    g++ -o testTime testTime.o Time.o
testTime.o: testTime.cpp Time.h
    g++ -c testTime.cpp -std=c++11 -Wall -pedantic
Time.o: Time.h Time.cpp
    g++ -c Time.cpp -std=c++11 -Wall -pedantic
```

Così come esistono i costruttori esistono anche i:

DISTRUTTORI

In generale quando definiamo una classe dobbiamo sempre scrivere un costruttore ed un distruttore. Così come i costruttori hanno lo stesso nome della classe a cui fanno riferimento, i distruttori hanno anch'essi lo stesso nome ma sono preceduti da una tilde “ ~ ”. Un distruttore può non specificare i parametri ed il tipo di ritorno, esattamente come il costruttore. Quando un oggetto esce dal suo scopo viene implicitamente chiamato un distruttore di quell'oggetto. Se noi non abbiamo definito il distruttore allora il compilatore definisce un distruttore vuoto del mio oggetto, ma se il mio oggetto ha allocato memoria devo necessariamente scrivere un distruttore per deallocare la memoria allocata all'interno di quell'oggetto. Quindi il costruttore viene chiamato implicitamente quando un oggetto è creato, un distruttore quando un oggetto è distrutto, ovvero rimosso dalla memoria. Le chiamate ai distruttori sono fatte in ordine inverso rispetto alle corrispondenti chiamate ai costruttori; tuttavia gli oggetti static e globali possono alterare l'ordine di chiamata dei distruttori.

Es: vediamo un esempio in cui si mostra l'ordine in cui vengono invocati costruttori e distruttori.

definizione della classe

```
1 // CreateAndDestroy.h
2 // CreateAndDestroy class definition.
3 // Member functions defined in CreateAndDestroy.cpp.
4 #include <string>
5
6 #ifndef CREATE_H
7 #define CREATE_H
8
9 class CreateAndDestroy {
10 public:
11     CreateAndDestroy(int, std::string); // constructor
12     ~CreateAndDestroy(); // destructor
13 private:
14     int objectID; // ID number for object
15     std::string message; // message describing object
16 };
17
18 #endif
```

```

1 // CreateAndDestroy.cpp
2 // CreateAndDestroy class member-function definitions.
3 #include <iostream>
4 #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
5 using std::cout; using std::endl; using std::string;
6
7 // constructor sets object's ID number and descriptive message
8 CreateAndDestroy::CreateAndDestroy(int ID, string messageString)
9   : objectID{ID}, message{messageString} {
10  cout << "Object " << objectID << " constructor runs "
11  << message << endl;
12 }
13 // destructor
14 CreateAndDestroy::~CreateAndDestroy() {
15  // output newline for certain objects; helps readability
16
17  cout << "Object " << objectID << " destructor runs "
18  << message << endl;
19 }
20 // testDestructors.cpp
21 // Order in which constructors and
22 // destructors are called.
23 #include <iostream>
24 #include "CreateAndDestroy.h" // include CreateAndDestroy class def
25 using std::cout; using std::endl;
26
27 void create(); // prototype
28
29 CreateAndDestroy first{1, "(global before main)" }; // global object
30
31 int main() {
32  cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
33  CreateAndDestroy second{2, "(local in main)" };
34  static CreateAndDestroy third{3, "(local static in main)" };
35
36  create(); // call function to create objects
37
38  cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
39  CreateAndDestroy fourth{4, "(local in main)" };
40  cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
41 }
42
43 // function to create objects
44 void create() {
45  cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
46  CreateAndDestroy fifth{5, "(local in create)" };
47  static CreateAndDestroy sixth{6, "(local static in create)" };
48  CreateAndDestroy seventh{7, "(local in create)" };
49  cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
50 }

```

```

1 // CreateAndDestroy.h
2 // CreateAndDestroy class definition.
3 // Member functions defined in CreateAndDestroy.cpp.
4 #include <string>
5
6 #ifndef CREATE_H
7 #define CREATE_H
8
9 class CreateAndDestroy {
10 public:
11  CreateAndDestroy(int, std::string); // constructor
12  ~CreateAndDestroy(); // destructor
13 private:
14  int objectID; // ID number for object
15  std::string message; // message describing object
16 };
17
18 #endif

```

```

1 # Makefile for testDestructors
2 testDestructors: testDestructors.o CreateAndDestroy.o
3  g++ -o testDestructors testDestructors.o CreateAndDestroy.o
4 testDestructors.o: testDestructors.cpp CreateAndDestroy.h
5  g++ -c testDestructors.cpp -std=c++11 -Wall -pedantic
6 CreateAndDestroy.o: CreateAndDestroy.h CreateAndDestroy.cpp
7  g++ -c CreateAndDestroy.cpp -std=c++11 -Wall -pedantic

```

Guardiamo un attimo testDestructors.cpp.

Il costruttore dell'oggetto *first* è il primo ad essere invocato poiché l'oggetto in questione è una variabile globale.

Il main contiene un messaggio sullo schermo e poi istanzia un nuovo oggetto con ID=2 e locale allo scopo del main. Quindi questo oggetto sarà distrutto quando terminerà il main. L'oggetto 3 è sempre definito nel main ma è di tipo statico; quindi questo oggetto sarà distrutto per ultimo prima del termine della funzione main. C'è poi l'invocazione ad una funzione *create()* che istanzia un nuovo oggetto con ID=5, locale alla funzione *create*. Poi viene istanziato un oggetto statico locale alla funzione *create*, e poi infine un settimo oggetto locale alla funzione *create*. Poi la funzione termina ed il controllo viene restituito alla funzione main. Si istanzia un nuovo oggetto con ID=4, locale al main.

Vediamo allora che il primo oggetto che viene creato è quello globale, prima che venga iniziata l'esecuzione della funzione main. Quando viene eseguita la funzione main viene invocato il costruttore dell'oggetto 2 e poi quello dell'oggetto 3. A questo punto

viene invocata la funzione *create* che istanzia l'oggetto 5, poi quello 6 ed infine quello 7. Terminata la funzione *create*, vengono invocati i distruttori degli oggetti qui definiti; tutti ad eccezione di quello dell'oggetto 6 che è di tipo statico e quindi continuerà ad esistere fino a quando non sarà terminato il main. Il controllo passa nuovamente al main e viene quindi invocato il costruttore dell'oggetto 4 locale al main. Termina il main e quindi vengono invocati i distruttori degli oggetti 4 e 2, nell'ordine inverso rispetto a quello dei costruttori. A questo punto vengono invocati, sempre in ordine inverso a quello dei costruttori, i distruttori degli oggetti nello scopo globale e quindi anche quelli statici: l'oggetto 6 per primo perché l'ultimo di cui è stato invocato il costruttore, l'oggetto 3 poi ed infine l'oggetto 1, il primo di cui è stato invocato il costruttore e quindi ultimo ad essere distrutto.

I distruttori servono quando il nostro oggetto alloca memoria dinamicamente; tramite essi infatti restituiamo al SO la memoria richiesta per la costruzione di un certo oggetto. Se non alloco dinamicamente allora, nel momento in cui il distruttore di default viene invocato, la memoria per rappresentare i dati membro dell'oggetto viene restituita al sistema operativo. Se però creiamo un oggetto che contiene un puntatore a float e lo istanziamo in modo tale che il puntatore punti ad una certa area di memoria, magari grande, quando viene invocato il distruttore di default di quell'oggetto viene rilasciata al SO soltanto la memoria necessaria per rappresentare il puntatore, ma non quella puntata dal puntatore. Quindi quando allochiamo dinamicamente in memoria dobbiamo SEMPRE scrivere un distruttore.

COSTRUTTORI DI COPIA

Vediamo subito un esempio:

```

1 // Date.cpp
2 // Date class member-function definitions.
3 #include <iostream>
4 #include <string>
5 #include "Date.h" // include definition of class Date from Date.h
6 using namespace std;
7
8 // Date constructor (should do range checking)
9 Date::Date(unsigned int m, unsigned int d, unsigned int y)
10 : month(m), day(d), year(y) {}
11
12 // print Date in the format mm/dd/yyyy
13 string Date::toString() const {
14     ostringstream output;
15     output << day << '/' << month << '/' << year;
16     return output.str();
17 }
18 // Date class declaration. Member functions are defined in Date.cpp.
19 #include <string>
20
21 // prevent multiple inclusions of header
22 #ifndef DATE_H
23 #define DATE_H
24
25 // class Date definition
26 class Date {
27 public:
28     explicit Date(unsigned int = 1, unsigned int = 1, unsigned int = 2000);
29     std::string toString() const;
30 private:
31     unsigned int month;
32     unsigned int day;
33     unsigned int year;
34 };
35 #endif
36 
```

```

1 // testData.cpp
2 // Demonstrating that class objects can be assigned
3 // to each other using default memberwise assignment.
4 #include <iostream>
5 #include "Date.h" // include definition of class Date from Date.h
6 using std::cout; using std::endl;
7
8 int main() {
9     Date date1{6, 4, 2021};
10    Date date2; // date2 defaults to 1/1/2000
11
12    cout << "date1 = " << date1.toString()
13    << "\ndate2 = " << date2.toString() << "\n\n";
14
15    date2 = date1; // default memberwise assignment
16
17    cout << "After default memberwise assignment, date2 = "
18    << date2.toString() << endl;
19 }
20 
```

```

$ make
g++ -c testData.cpp -std=c++11 -Wall -pedantic
g++ -c Date.cpp -std=c++11 -Wall -pedantic
g++ -o testData testData.o Date.o
$ ./testData
date1 = 4/6/2021
date2 = 1/1/2000

After default memberwise assignment, date2 = 4/6/2021

```

Il messaggio di questo esempio sta tutto nella riga 15 di *testDate.cpp*. La riga 15 copia il contenuto di un oggetto in un altro oggetto con l'istruzione:

```
date2 = date1 ;
```

dove *date1* e *date2* sono 2 oggetti della stessa classe *Date*.

Gli oggetti di tipo *Date* contengono 2 variabili membro private: mese, giorno ed anno; e poi hanno un costruttore con dei valori di default che riceve in ingresso 3 interi (m, d, y) ed istanzia mese, giorno ed anno con questi parametri che riceve in ingresso.

Alla riga 15 di *testDate.cpp* succede che, con l'assegnazione fatta, i dati membro di *date1* vengono copiati nei dati membro di *date2*. Quindi:

- *date1.day()* viene copiato in *date2.day()*;
- *date1.month()* viene copiato in *date2.month()*;
- *date1.year()* viene copiato in *date2.year()*.

Per default questa assegnazione quindi è fatta per ogni variabile mebro.

Questa cosa però non va fatta assolutamente con un puntatore. Infatti in tal modo copiamo non il valore puntato dal puntatore ma il contenuto del puntatore e quindi facciamo in maniera tale che il puntatore contenuto nell'oggetto a sinistra dell'assegnazione punti alla stessa area di memoria dell'oggetto a destra dell'assegnazione. Quindi non generiamo nuova memoria ma facciamo puntare alla stessa area di memoria 2 puntatori.

Non lo abbiamo ancora visto, ma di fatto quando ad una funzione passiamo un oggetto come parametro per valore, quello che succede è che il C++ usa un costruttore di copia di default per fare una copia *memberwise*, esattamente come quella che abbiamo visto nell'esempio precedente. Per ciascuna classe il compilatore fornisce un *default copy constructor*, cioè un costruttore copia di default che copia ciascun membro dell'oggetto originale nel corrispondente membro del nuovo oggetto.

La cosa importante è che con i puntatori non si alloca nuova memoria perché si copia il puntatore e non il valore a cui punta. Quindi di fatto dovremo scrivere noi un copy constructor per risolvere questa cosa, e per farlo ci serve il concetto di *overloading* degli operatori (lo vedremo la prossima lezione).

COMPOSIZIONE DI OGGETTI

Un oggetto può contenere al suo interno altri oggetti; ad esempio può contenere variabili membro che sono oggetti di un'altra classe. Vediamo subito un esempio.

```

1 // Date.h
2 // Date class definition; Member functions defined in Date.cpp
3 #include <iostream>
4
5 #ifndef DATE_H
6 #define DATE_H
7
8 class Date {
9 public:
10     static const unsigned int monthsPerYear{12}; // months in a year
11     explicit Date(unsigned int m = 1, unsigned int d = 1, unsigned int y = 1900);
12     std::string toString() const; // date string in month/day/year format
13     ~Date(); // provided to confirm destruction order
14 private:
15     unsigned int month; // 1-12 (January-December)
16     unsigned int day; // 1-31 based on month
17     unsigned int year; // any year
18
19     // utility function to check if day is proper for month and year
20     unsigned int checkDay(int) const;
21 };
22

```

```

1 // Date.cpp
2 // Date class member-function definitions.
3 #include <array>
4 #include <iostream>
5 #include <sstream>
6 #include <stdexcept>
7 #include "Date.h" // include Date class definition
8 using std::string; using std::ostringstream;
9 using std::endl; using std::array; using std::cout;
10 using std::invalid_argument;
11
12 // constructor confirms proper value for month; calls
13 // utility function checkDay to confirm proper value for day
14 Date::Date(unsigned int dy, unsigned int mn, unsigned int yr)
15 : day{checkDay(dy)}, month{mn}, year{yr} {
16     if (mn < 1 || mn > monthsPerYear) { // validate the month
17         throw invalid_argument("month must be 1-12");
18     }
19
20     // output Date object to show when its constructor is called
21     cout << "Date object constructor for date " << toString() << endl;
22 }
23
24 // print Date object in form month/day/year
25 string Date::toString() const {
26     ostringstream output;
27     output << day << '/' << month << '/' << year;
28     return output.str();
29 }
30 // output Date object to show when its destructor is called
31 Date::~Date() {
32     cout << "Date object destructor for date " << toString() << endl;
33 }
34
35
36 // utility function to confirm proper day value based on
37 // month and year; handles leap years, too
38 unsigned int Date::checkDay(int testDay) const {
39     static const array<int, monthsPerYear + 1> daysPerMonth{
40         0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
41
42     // determine whether testDay is valid for specified month
43     if (testDay > 0 && testDay <= daysPerMonth[month]) {
44         return testDay;
45     }
46
47     // February 29 check for leap year
48     if (month == 2 && testDay == 29 && (year % 400 == 0 ||
49         (year % 4 == 0 && year % 100 != 0))) {
50         return testDay;
51     }
52
53     throw invalid_argument("Invalid day for current month and year");
54 }
55

```

```

1 // Employee.h
2 // Employee class definition showing composition.
3 // Member functions defined in Employee.cpp.
4 #include <string>
5
6 #ifndef EMPLOYEE_H
7 #define EMPLOYEE_H
8
9 #include <string>
10 #include "Date.h" // include Date class definition
11
12 class Employee {
13 public:
14     Employee(const std::string&, const std::string&,
15             const Date&, const Date&);
16     std::string toString() const;
17     ~Employee(); // provided to confirm destruction order
18 private:
19     std::string firstName; // composition: member object
20     std::string lastName; // composition: member object
21     const Date birthDate; // composition: member object
22     const Date hireDate; // composition: member object
23 },
24
25 #endif

```

```

1 // Employee.cpp
2 // Employee class member-function definitions.
3 #include <iostream>
4 #include <sstream>
5 #include "Employee.h" // Employee class definition
6 #include "Date.h" // Date class definition
7 using std::string; using std::endl;
8 using std::ostringstream; using std::cout;
9
10 // constructor uses member initializer list to pass initializer
11 // values to constructors of member objects
12 Employee::Employee(const string& first, const string& last,
13                     const Date &dateOfBirth, const Date &dateOfHire)
14 : firstName{first}, // initialize firstName
15   lastName{last}, // initialize lastName
16   birthDate{dateOfBirth}, // initialize birthDate
17   hireDate{dateOfHire} { // initialize hireDate
18     // output Employee object to show when constructor is called
19     cout << "Employee object constructor: "
20     << firstName << ' ' << lastName << endl;
21 }
22
23 // print Employee object
24 string Employee::toString() const {
25     ostringstream output;
26     output << lastName << ", " << firstName << " Hired: "
27     << hireDate.toString() << " Birthday: " << birthDate.toString();
28     return output.str();
29 }
30
31 // output Employee object to show when its destructor is called
32 Employee::~Employee() {
33     cout << "Employee object destructor: "
34     << lastName << ", " << firstName << endl;
35 }
36
1 #Makefile for testEmployee example
2 testEmployee: testEmployee.o Date.o Employee.o
3         g++ -o testEmployee testEmployee.o Date.o      Employee.o
4 testEmployee.o: testEmployee.cpp Date.h Employee.h
5         g++ -c testEmployee.cpp -Wall -pedantic -std=c++11
6 Employee.o: Date.h Employee.h Employee.cpp
7         g++ -c Employee.cpp -Wall -pedantic -std=c++11
8 Date.o: Date.h Date.cpp
9         g++ -c date.cpp -Wall -pedantic -std=c++11

```

```

1 // testEmployee.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <iostream>
4 #include "Date.h" // Date class definition
5 #include "Employee.h" // Employee class definition
6 using std::cout;
7 using std::endl;
8 int main() {
9     Date birth{26, 5, 1926};
10    Date hire{ 17, 8, 1958}; //Date of hire (A Kind of Blue)
11    Employee manager{"Miles", "Davis", birth, hire};
12
13    cout << "\n" << manager.toString() << endl;
14 }

```

Abbiamo due classi (*Date* e *Employee*) di cui forniamo l’interfaccia e l’implementazione, e poi c’è un programma per testare le classi. L’obiettivo di questo progetto è quello di mostrare la composizione di oggetti. In particolare la classe *Employee* fa uso della classe *Date*.

Sul file *Date.h* notiamo alla riga 10 viene definita una variabile costante di tipo *static*; quindi possiamo avere dei servizi statici all’interno di un oggetto. Poi vedremo meglio l’utilità di ciò, però il senso è che questi servizi sono servizi della classe e non di uno specifico oggetto.

In *Employee.h* (righe 21-22) definiamo 2 variabili membro che sono a loro volta oggetti di tipo *Date*; questa è un esempio di composizione di oggetti.

Notiamo poi che in *Date.h* è definita la funzione *checkDay* come privata (e costante); perché è definita come privata? Perché è usata solo dalle funzioni membro della classe *Date* e quindi non c’è nessuna necessità che un client utilizzi questa funzione. Quindi vediamo che i membri privati non devono essere necessariamente delle variabili ma possono essere anche delle funzioni.

Notiamo infine che in *Employee.h* alle righe 16 e 17 l’inizializzazione delle variabili membro, che sono degli oggetti, avviene tramite l’invocazione del costruttore copia di default.

LEZIONE 14

Finiamo con gli ultimi 2 argomenti di cui dobbiamo discutere prima di parlare dell’overloading, e cioè con le “friend class and functions” e con i membri statici di una classe.

FUNZIONI E CLASSI “FRIEND”

Le funzioni e le classi *friend* scavalcano il concetto di encapsulamento a cui facciamo riferimento nell’ambito della programmazione orientata agli oggetti; infatti permettono ad una funzione o ad una classe di accedere ai membri privati ed a quelli protetti (ovvero quelli ereditati da delle classi derivate da una classe base, poi vedremo) di un’altra classe in cui è dichiarata come “amica”.

Es:

```
1 class Node {  
2     private:  
3         int key;  
4         Node* next;  
5     /* Other members of Node Class */  
6  
7     // Now class LinkedList can  
8     // access private members of Node  
9     friend class LinkedList;  
10 };  
11  
  
5 class Box {  
6     double width;  
7  
8     public:  
9         friend void printWidth( Box box );  
10        void setWidth( double wid );  
11 };  
  
--  
14 // Note: printWidth() is not a member function of any class.  
15 void printWidth( Box box ) {  
16     /* Because printWidth() is a friend of Box, it can  
17     directly access any member of this class */  
18     cout << "Width of box : " << box.width << endl;  
19 }  
--
```

Ho una classe *Node* che concede ad un’altra classe, *LinkedList*, il privilegio di accedere ai propri dati privati.

Questo scavala/elimina il vantaggio dell’incapsulamento proprio dei linguaggi *object oriented*, però in alcuni casi può essere utile.

Classi e funzioni amiche vanno usate con cautela e per scopi specifici poiché diminuisce il valore dell’incapsulamento e della separazione tra le classi: se tutti sono amici di tutti allora tutti possono accedere a tutto e quindi perde di senso la separazione tra le classi.

Nell’esempio poi abbiamo dichiarato una funzione, *printWidth*, come *friend* della classe *Box* e quindi *printWidth* ha il privilegio, anche se non è una funzione membro dell’oggetto *Box*, di accedere ai membri privati di quest’oggetto. Ed infatti *printWidth* prende in ingresso un oggetto di tipo *Box*, tramite cui accede al membro privato *width*. Notiamo che in realtà non abbiamo definito *width* come privato; questo perché quando non specifichiamo la modalità di accesso ad un certo membro, questo di default è privato.

Cosa interessante è che la “friendship” non è simmetrica: se A è amica di B non necessariamente B è amica di A. Nel nostro esempio la classe *LinkedList* può accedere ai dati membro della classe *Node*, ma non è detto che valga il contrario.

PUNTATORE “THIS”

Ogni oggetto ha definito al suo interno implicitamente un puntatore a se stesso, detto puntatore “this”. Questo puntatore è un parametro隐式的 a tutte le funzioni membro e non è altro che un puntatore all’oggetto su cui una funzione viene invocata.

Es: (alla riga 1 c’è “int”, non “in” ovviamente)

```
1 void Time::setHour(int hour) {
2     if (hour >= 0 && hour < 24) {
3         this->hour = hour; // use this-> to access data member
4     }
5     else {
6         throw invalid_argument("hour must be 0-23");
7     }
8 }
```

La funzione *setHour* è la funzione membro di un certo oggetto di classe *Time* e viene quindi invocata su tale oggetto. Quindi *this* non è altro che un puntatore a un oggetto su cui è stata invocata la funzione membro. Nell’esempio l’utilizzo di *this* serve per capire *hour* a quale variabile si riferisce. Infatti *hour* è sia la variabile di tipo intero che riceve la funzione *setHour* che il nome di un dato membro. Quindi per evitare conflitti *this->hour* si riferisce alla variabile membro, *hour* alla variabile *int*.

“This” è molto utile nelle funzioni membro, come quelle di tipo *set*, che vanno a modificare dati membro restituiscano il puntatore all’oggetto che è stato modificato. Questo perché molto spesso un oggetto può far parte di un’espressione.

Vediamo un esempio sfruttando la classe *Time*.

Example: cascade function calls with `this`

```

1 // Time.h
2 // Time class modified to enable cascaded member-function calls.
3 #include <string>
4
5 // Time class definition.
6 // Member functions defined in Time.cpp.
7 #ifndef TIME_H
8 #define TIME_H
9
10 class Time {
11 public:
12     explicit Time(int = 0, int = 0, int = 0); // default constructor
13
14     // set functions (the Time& return types enable cascading)
15     Time& setTime(int, int, int); // set hour, minute, second
16     Time& setHour(int); // set hour
17     Time& setMinute(int); // set minute
18     Time& setSecond(int); // set second
19
20     unsigned int getHour() const; // return hour
21     unsigned int getMinute() const; // return minute
22     unsigned int getSecond() const; // return second
23     std::string toUniversalString() const; // 24-hour time format string
24     std::string toStandardString() const; // 12-hour time format string
25 private:
26     unsigned int hour{0}; // 0 - 23 (24-hour clock format)
27     unsigned int minute{0}; // 0 - 59
28     unsigned int second{0}; // 0 - 59
29 };
30
31 // Cascade.cpp
32 // Cascading member-function calls with the this pointer.
33 #include <iostream>
34 #include "Time.h" // Time class definition
35 using std::cout; using std::endl;
36
37 int main() {
38     Time t; // create Time object
39
40     t.setHour(18).setMinute(30).setSecond(22); // cascaded function calls
41
42     // output time in universal and standard formats
43     cout << "Universal time: " << t.toUniversalString()
44     << "\nStandard time: " << t.toStandardString();
45
46     // cascaded function calls
47     cout << "\n\nNew standard time: "
48     << t.setTime(20, 20, 20).toStandardString() << endl;
49 }

```

```

12 Time::Time(int hr, int min, int sec) {
13     setTime(hr, min, sec);
14 }
15 // set values of hour, minute, and second
16 Time& Time::setTime(int h, int m, int s) { // note Time& return
17     setHour(h);
18     setMinute(m);
19     setSecond(s);
20     return *this; // enables cascading
21 }
22
23 // set hour value
24 Time& Time::setHour(int h) { // note Time& return
25     if (h >= 0 && h < 24) {
26         hour = h;
27     }
28     else {
29         throw invalid_argument("hour must be 0-23");
30     }
31     return *this; // enables cascading
32 }
33
34 // set minute value
35 Time& Time::setMinute(int m) { // note Time& return
36     if (m >= 0 && m < 60) {
37         minute = m;
38     }
39     else {
40         throw invalid_argument("minute must be 0-59");
41     }
42     return *this; // enables cascading
43 }
44
45 // set second value
46 Time& Time::setSecond(int s) { // note Time& return
47     if (s >= 0 && s < 60) {
48         second = s;
49     }
50     else {
51         throw invalid_argument("second must be 0-59");
52     }
53     return *this; // enables cascading
54 }
55
56 // get hour value
57 unsigned int Time::getHour() const { return hour; }
58
59 // get minute value
60 unsigned int Time::getMinute() const { return minute; }
61
62 // get second value
63 unsigned int Time::getSecond() const { return second; }
64
65 // return Time as a string in universal-time format (HH:MM:SS)
66 string Time::toUniversalString() const {
67     ostringstream output;
68     output << setw(2) << getHour() << ":"
69     << setw(2) << getMinute() << ":" << setw(2) << getSecond();
70     return output.str();
71 }
72
73 // return Time as string in standard-time format (HH:MM:SS AM or PM)
74 string Time::toStandardString() const {
75     ostringstream output;
76     output << ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12)
77     << ":" << setw(2) << getMinute() << ":" << setw(2)
78     << getSecond() << (hour < 12 ? " AM" : " PM");
79     return output.str();
80 }

```

```

1 #Makefile for cascade
2 cascade: cascade.o Time.o
3 g++ -o cascade cascade.o Time.o
4 cascade.o: cascade.cpp Time.h
5 g++ -c cascade.cpp -Wall -pedantic -std=c++11
6 Time.o: Time.cpp Time.h
7 g++ -c Time.cpp -Wall -pedantic -std=c++11

```

- The dot operator (.) associates from left to right, so line 10
`t.setHour(18).setMinute(30).setSecond(22);`
- first evaluates `t.setHour(18)`, then returns a reference to (the updated) object `t` as the value of this function call. The remaining expression is then interpreted as
`t.setMinute(30).setSecond(22);`
- The `t.setMinute(30)` call executes and returns a reference to the (further updated) object `t`. The remaining expression is interpreted as
`t.setSecond(22);`

L’oggetto modificato viene restituito da ciascuna delle funzioni *set* tramite il puntatore *this*. Nel momento in cui definisco una variabile *t* di tipo *Time* viene invocato il costruttore di default con tutti i valori delle variabili membro posti a 0.

La riga 10 è importante perché ci dice che possiamo invocare in cascata diverse funzioni sullo stesso oggetto (*t*). Gli operatori “.” che stanno allo stesso livello vengono eseguiti da sinistra a destra. Quindi *t.setHour(18)* assegna alla variabile membro *hour* dell’oggetto *t* il valore “18”; a questo punto *setHour* restituisce l’oggetto che ha appena modificato. Quindi l’oggetto modificato diventa l’oggetto che si trova all’interno dell’espressione. *t.setHour* è a sua volta un oggetto, quindi posso utilizzare l’operatore “.” ed invocare uno dei servizi messi a disposizione dalla classe *Time*, come *setMinute*. Allora: *t.setHour* è un oggetto in cui ho scritto 18; sullo stesso oggetto viene invocato *setMinute* per cambiare i minuti ed in particolare per metterli a 30; anche *setMinute* restituisce **this*, poiché restituisce l’oggetto su cui è stato invocato. Allora si vede quanto sia utile restituire il puntatore *this* in espressioni in cui compaiono funzioni membro in cascata su un certo oggetto. Senza “return **this*” avremmo dovuto invocare le funzioni membro in diverse istruzioni, separate quindi da un punto e virgola. Il puntatore *this* invece ci permette di usare l’oggetto modificato all’interno di

un'espressione. Restituendo l'oggetto infatti avremo che `t.setHour()` è un oggetto, così come `t.setMinute()` etc. e quindi possiamo usare l'operatore “.”. La cosa importante da capire quindi è che `setHour()`, `setMinute()` etc. sono TUTTI oggetti grazie al puntatore `this`.

MEMBRI STATICI DI UNA CLASSE

Abbiamo visto le variabili statiche di una funzione ed abbiamo detto che queste continuano ad esistere dopo che la funzione è terminata. Le variabili statiche di un oggetto per analogia dovrebbero continuare ad esistere anche dopo che quell'oggetto termina. In realtà queste variabili condivise tra tutti gli oggetti di una classe che servono quando vogliamo rendere delle informazioni globali a tutti gli oggetti di quella classe. Le variabili di tipo statico sono usate a livello di classe e non di uno specifico oggetto.

Es: orchestra

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 enum saxtype {SOPRANO, ALTO, TENOR, BARITONE};
6
7 class Saxophonist {
8 public:
9     static int saxCount;
10    // Constructor definition
11    Saxophonist(std::string n, saxtype p): name(n), plays(p){
12        cout << "Constructor called." << endl;
13        // Increase every time object is created
14        saxCount++;
15    }
16    ~Saxophonist(){
17        cout << "~Saxophonist() called" << endl;
18        --saxCount;
19    }
20    int getCount () { return saxCount;}
21
22 private:
23    std::string name;      // Musician name
24    saxtype plays;        // type os sax played
25 };
26
27 int Saxophonist::saxCount=0;
28 int main(void) {
29
30     cout << "Number of available saxophonists before definition " <<
31     Saxophonist::saxCount << endl;
32     {
33         Saxophonist m1("John Coltrane", TENOR);    // Declare musician1
34         Saxophonist m2("Cannonball Haderley", ALTO); // Declare musician2
35         Saxophonist m3("Steve Lacy", SOPRANO); //Declare musician3
36
37
38     // Print total number of objects.
39     cout << "Total objects in scope: " << Saxophonist::saxCount << endl;
40 }
41 cout << "Total objects outside the scope: " << Saxophonist::saxCount << endl;
42 return 0;
43 }
```

Number of available saxophonists before definition 0
Constructor called.
Constructor called.
Constructor called.

Total objects in scope: 3
~Saxophonist() called
~Saxophonist() called
~Saxophonist() called

Total objects outside the scope: 0

In questo caso ho la sezione flauti e voglio sapere quante persone ci sono nella sezione flauti; idem per violoncelli etc. Per fare questo usiamo una variabile membro di tipo statico.

Nella classe *Saxophonist* uso ad esempio una variabile *saxCount* di tipo statico comune a tutti gli oggetti della classe *Saxophonist*. Nel costruttore ci sono nome del musicista e tipo di sassofono che suona. Ad ogni chiamata del costruttore di un oggetto di questo tipo incremento la variabile *saxCount* e ad ogni invocazione di un distruttore di un certo oggetto di classe *Saxophonist* decremento *saxCount*. C'è poi una funzione di tipo public che restituisce il valore contenuto di questa variabile di tipo statico. La variabile di tipo statico come va usata? Guardiamo la riga 27, ha di strano una cosa. Ha di strano il fatto che viene utilizzata *saxCount* anche se non è stato creato alcun oggetto di tipo *Saxophonist*! Quindi le variabili statiche vengono create con la classe e non con l'oggetto. Solitamente invece quando accediamo ad un dato membro dobbiamo accedere ad un dato membro di un certo oggetto e quindi va prima creato un oggetto, come abbiamo fatto nel penultimo esempio, quello sulle cascate di funzioni membro. Lì abbiamo creato un oggetto *t* di tipo *Time* e poi abbiamo invocato le funzioni. I membri statici invece sono servizi di classe.

Tornando al nostro esempio, alla riga 27 quindi viene inizializzato il membro statico. Vengono poi creati 3 oggetti e viene stampata a video *saxCount*, comune a tutta la classe.

Oss: Notiamo che sempre alla riga 27 abbiamo anteposto il tipo int alla variabile *saxCount* prima di inizializzarla nonostante già l'avessimo definita nella classe; questo poiché di fatto alla riga 27 è una dichiarazione. Non c'è bisogno di static ma solo del tipo perché è con questa istruzione che viene allocata la variabile in memoria e quindi c'è bisogno di sapere lo spazio da riservare.

Altro esempio:

COMPOSIZIONE DI OGGETTI

classe RETTANGOLO.

In questo esempio vediamo la creazione di un rettangolo con varie funzioni che calcolano perimetro, area, altezza, base e c'è anche una funzione che disegna il rettangolo. Vengono sfruttate 2 classi: una è la classe *Point*, l'altra è la classe *Rectangle* che fa uso della prima; è un esempio di composizione di oggetti. In questo esempio vediamo anche un makefile particolare.

Es:

```
1 // Point.h
2 #ifndef POINT_H
3 #define POINT_H
4
5 class Point {
6 public:
7     explicit Point(double x = 0.0, double y = 0.0); // default constructor
8
9     // set and get functions
10    void setX(double);
11    void setY(double);
12    double getX() const;
13    double getY() const;
14 private:
15    double x; // 0.0 <= x <= 20.0
16    double y; // 0.0 <= y <= 20.0
17 };
18

1 // Point.cpp
2 // Member-function definitions for class Point.
3 #include <stdexcept>
4 #include "Point.h" // include definition of class Point
5 using namespace std;
6
7 Point::Point(double xCoord, double yCoord) {
8     setX(xCoord); // invoke function setX
9     setY(yCoord); // invoke function setY
10 }
11
12 // set x coordinate
13 void Point::setX(double xCoord) {
14     if (xCoord < 0.0 || xCoord > 20.0) {
15         throw invalid_argument("x must be >= 0.0 and <= 20.0");
16     }
17
18     x = xCoord;
19 }
20
21 // set y coordinate
22 void Point::setY(double yCoord) {
23     if (yCoord < 0.0 || yCoord > 20.0) {
24         throw invalid_argument("y must be >= 0.0 and <= 20.0");
25     }
26
27     y = yCoord;
28 }
29
30 // return x coordinate
31 double Point::getX() const {
32     return x;
33 }
34
35 // return y coordinate
36 double Point::getY() const {
37     return y;
38 }
```

```
1 // Rectangle.h
2 #ifndef RECTANGLE_H
3 #define RECTANGLE_H
4
5 #include "Point.h" // include definition of class Point
6
7 class Rectangle {
8 public:
9     // default constructor
10    explicit Rectangle(Point p1 = Point(0.0, 1.0), Point p2 = Point(1.0, 1.0),
11        Point p3 = Point(1.0, 0.0), Point p4 = Point(0.0, 0.0),
12        char fillChar = '*', char perimeterChar = '*');
13
14    // sets x, y, x2, y2 coordinates
15    // sets x, y, x2, y2 coordinates
16    void setCoord(Point p1, Point p2, Point p3, Point p4);
17    double height() const; // length
18    double width() const; // width
19    double perimeter() const; // perimeter
20    double area() const; // area
21    bool square() const; // square
22    void draw() const; // draw rectangle
23    void setPerimeterCharacter(char); // set perimeter character
24    void setFillCharacter(char); // set fill character
25 private:
26    Point point1;
27    Point point2;
28    Point point3;
29    Point point4;
30    char fillCharacter;
31    char perimeterCharacter;
32 };
33
34 #endif
35

1 // Rectangle.cpp
2 // Member-function definitions for class Rectangle.
3 #include <iostream>
4 #include <iomanip>
5 #include <cmath>
6 #include <stdexcept>
7 #include "Rectangle.h" // include definition of class Rectangle
8 using std::invalid_argument;
9 using std::cout;
10
11 Rectangle::Rectangle(Point p1, Point p2, Point p3, Point p4) {
12     setCoord(p1, p2, p3, p4); // invoke function setCoord
13     setFillCharacter(fillCharacter); // set fill character
14     setPerimeterCharacter(perimeterCharacter); // set perimeter character
15 }
16
17 void Rectangle::setCoord(Point p1, Point p2, Point p3, Point p4) {
18     // Arrangement of points
19     // p4.....p3
20     //   .
21     //   .
22     //   .
23     // p1.....p2
24
25     // verify that points form a rectangle
26     if (((p1.getY() == p2.getY() && p1.getX() == p4.getX())
27          && p2.getX() == p3.getX() && p3.getY() == p4.getY())) {
28         throw invalid_argument("Coordinates do not form a rectangle!\n");
29     }
30
31     point1 = p1;
32     point2 = p2;
33     point3 = p3;
34     point4 = p4;
35 }
36
37 double Rectangle::height() const {
38     return fabs(point4.getY() - point1.getY());
39 }
40
41 double Rectangle::width() const {
42     return fabs(point2.getX() - point1.getX());
43 }
44
45
46 double Rectangle::perimeter() const {
47     return 2 * (height() + width());
48 }
49
50 double Rectangle::area() const {
51     return height() * width();
52 }
53
54 bool Rectangle::square() const {
55     return height() == width();
56 }
```

(Rectangle.cpp è a metà; la parte finale è alla pagina successiva.)

```

53
54 bool Rectangle::square() const {
55     return height() == width();
56 }
57
58 // draw rectangle
59 void Rectangle::draw() const {
60     for (double y{25.0}; y >= 0.0; --y) {
61         for (double x{0.0}; x <= 25.0; ++x) {
62             if ((point1.getX() == x && point1.getY() == y) ||
63                 (point4.getX() == x && point4.getY() == y)) {
64                 // print horizontal perimeter of rectangle
65                 while (x <= point2.getX()) {
66                     cout << perimeterCharacter;
67                     ++x;
68                 }
69
70                 cout << '.'; // print remainder of quadrant
71             }
72             else if (((x <= point4.getX() && x >= point1.getX()) &&
73 point4.getY() >= y && point1.getY() <= y) {
74                 cout << perimeterCharacter;
75
76                 // fill inside of rectangle
77                 for (++x; x < point2.getX();)
78                     cout << fillCharacter;
79                     ++x;
80                 }
81
82                 cout << perimeterCharacter;
83             }
84             else {
85                 cout << '.'; // print quadrant background
86             }
87         }
88
89         cout << '\n';
90     }
91 }
92
93 // set fill character
94 void Rectangle::setFillCharacter(char fillChar) {
95     fillCharacter = fillChar;
96 }
97
98 // set perimeter character
99 void Rectangle::setPerimeterCharacter(char perimeterChar) {
100    perimeterCharacter = perimeterChar;
101 }
```

```

1 // driver.cpp
2 #include <iostream>
3 using std::cout;
4 #include "Rectangle.h" // include definition of class Rectangle
5
6 int main() {
7     Point point1{6.0, 10.0};
8     Point point2{18.0, 10.0};
9     Point point3{18.0, 20.0};
10    Point point4{6.0, 20.0};
11
12    Rectangle rectangle{point1, point2, point3, point4, '-', '*'};
13
14    cout << "Rectangle height : " << rectangle.height() << "\n";
15    cout << "Rectangle width : " << rectangle.width() << "\n";
16    cout << "Rectangle perimeter : " << rectangle.perimeter() << "\n";
17    cout << "Rectangle area : " << rectangle.area() << "\n";
18
19    rectangle.draw(); // invokes function draw
20 }
```

- This is a better **Makefile** for our program see (<https://makefiletutorial.com>) for a tutorial
- The rule in line 8 says that it applies to all files ending in the .o suffix
- Uses automatic variables in the rules
 - **\$@** and **\$^**, the left and right sides of the : respectively, and the **\$<** is the first item in the dependencies list

make

```

g++ -c -o Point.o Point.cpp -std=c++11
g++ -c -o Rectangle.o Rectangle.cpp -std=c++11
g++ -c -o driver.o driver.cpp -std=c++11
g++ -o driver Point.o Rectangle.o driver.o -std=c++11
```

```

1 # Makefile for the Rectangle example
2
3 CC=g++
4 CFLAGS=-std=c++11
5 DEPS = Point.h Rectangle.h
6 OBJ = Point.o Rectangle.o driver.o
7
8 %.o: %.cpp $(DEPS)
9         $(CC) -c -o $@ $< $(CFLAGS)
10
11 driver: $(OBJ)
12         $(CC) -o $@ $^ $(CFLAGS)
13 ■
```

RISCRITTURA AUTOMATICA DEL MAKEFILE

In basso a destra nell'ultima immagine è mostrato il makefile che è diverso dal solito. Si sono innanzitutto create delle variabili per il comando g++, per le opzioni, per racchiudere i file.h e per racchiudere quelli .o, ovvero gli oggetti da generare. Poi si sono usate le *variabili automatiche* che servono per una riscrittura automatica del makefile; questo conviene quando si hanno molti file da includere in un progetto e quando si hanno molte classi; noi alla fin fine ne useremo massimo 2 in un programma. In questo tipo di makefile si utilizzano:

- **\$@** = quello che c'è a sinistra di “ : ” ;
- **\$^** = quello che c'è a destra di “ : ” ;
- **\$<** = primo elemento nella lista di dipendenze (destra dei :).

La forma del makefile resta la stessa. Solo che possiamo usare dei simboli generici che vanno ad essere riscritti e a generare un insieme di regole.

```
1 # Makefile for the Rectangle example
2
3 CC=g++
4 CFLAGS=-std=c++11
5 DEPS = Point.h Rectangle.h
6 OBJ = Point.o Rectangle.o driver.o
7
8 %.o: %.cpp $(DEPS)
9     $(CC) -c -o $@ $< $(CFLAGS)
10
11 driver: $(OBJ)
12     $(CC) -o $@ $^ $(CFLAGS)
13 ■
```

Alla riga 11 vediamo che il target *driver*, ovvero l'applicazione finale, dipende solo dai file oggetto in quanto è la fase di linking. Le dipendenze allora sono i file oggetto contenuti nella variabile **OBJ**, a cui si accede con **\$(OBJ)** e poi per compilare usiamo il comando g++ contenuto nella variabile **CC** e usiamo 2 delle variabili definite prima; nello specifico usiamo **\$@** per riferirci al target della regola mentre **\$^** per riferirci alla parte a destra dei 2 punti e quindi ai prerequisiti. Quando il makefile incontra queste variabili automatiche sostituisce ad esse la parte sinistra e la parte destra della dipendenza. Quindi di fatto la riga 12 diventerà:

g++ -o driver Point.o Rectangle.o driver. o –std=c++11.

Passiamo alla regola alla riga 8. Questa va a generare una regola per tutti i file .cpp, ovvero una regola di riscrittura che dice, tramite **%.o** e **%.cpp** per riferirci ad un generico file rispettivamente .o e .cpp, che il corrispondente file .o dipende dal file .cpp e dal contenuto della variabile **DEPS**, ovvero dalle interfacce *Point.h* e *Rectangle.h*. Alla riga 9 c'è la ricetta che corrisponde, come prima cosa, a:

g++ -c –o Point.o Point.cpp (primo elemento a destra dei :) –std=c++11

Lo stesso lo si fa con Rectangle.cpp e con driver.cpp e quindi vengono generati in tutto 3 file oggetto.

C'è poi un altro esercizio sfizioso, ovvero quello del gioco del TRIS.

Exercise: TicTacToe

- Create a class TicTacToe that will enable you to write a complete program to play the game.
- Data should be organized as 3x3 array
- The players in turn make the choice
- After the choice the program checks if there is win or a draw
- The program outputs the board at every choice
- Example of the game on the right

```
s ./driver
 0 1 2
 0
 1
 2
```

```
?player X enter move: 0 0
```

```
 0 1 2
  X |
  - -
  L
  - -
  Z
```

```
?player O enter move: 1 1
```

```
 0 1 2
  X |
  - -
  L | O
  - -
  Z
```

```
?player X enter move: 0 1
```

```
 0 1 2
  X | X |
  - -
  L | O
  - -
  Z
```

```
?player O enter move: 1 2
```

```
 0 1 2
  X | X |
  - -
  L | O | O
  - -
  Z
```

```
?player X enter move: 0 2
```

```
 0 1 2
  X | X | X
  - -
  L | O | O
  - -
  Z
```

```
1 // TicTacToe.h
2 #ifndef TICTACTOE_H
3 #define TICTACTOE_H
4 #include <array>
5
6 class TicTacToe {
7 private:
8     enum Status {WIN, DRAW, CONTINUE}; // enumeration constants
9     std::array<std::array<char, 3>, 3> board;
10 public:
11     TicTacToe(); // default constructor
12     void makeMove(); // make move
13     void printBoard() const; // print board
14     bool validMove(int, int) const; // validate move
15     bool xoMove(char); // x o move
16     Status gameStatus() const; // game status
17 };
18
19 #endif
```

```

1 // TicTacToe.cpp
2 // Member-function definitions for class TicTacToe.
3 #include <iostream>
4 #include <iomanip>
5 #include "TicTacToe.h" // include definition of class TicTacToe
6 using std::cout;
7 using std::cin;
8 using std::setw;
9 TicTacToe::TicTacToe() {
10    for (int j{0}; j < 3; ++j) { // initialize board
11        for (int k{0}; k < 3; ++k) {
12            board[j][k] = ' ';
13        }
14    }
15 }
16
17 bool TicTacToe::validMove(int r, int c) const {
18    return r >= 0 && r < 3 && c >= 0 && c < 3 && board[r][c] == ' ';
19 }
20
21 // must specify that type Status is part of the TicTacToe class.
22 TicTacToe::Status TicTacToe::gameStatus() const {
23    // check for a win on diagonals
24    if (board[0][0] != ' ' && board[0][0] == board[1][1] && board[0][0] == board[2][2]) {
25        return WIN;
26    }
27    else if (board[2][0] != ' ' && board[2][0] == board[1][1] && board[2][0] == board[0][2]) {
28        return WIN;
29    }
30
31    // check for win in rows
32    for (int a{0}; a < 3; ++a) {
33        if (board[a][0] != ' ' && board[a][0] == board[a][1] && board[a][0] == board[a][2]) {
34            return WIN;
35        }
36    }
37
38    // check for win in columns
39    for (int a{0}; a < 3; ++a) {
40        if (board[0][a] != ' ' && board[0][a] == board[1][a] && board[0][a] == board[2][a]) {
41            return WIN;
42        }
43    }
44
45    // check for a completed game
46    for (int r{0}; r < 3; ++r) {
47        for (int c{0}; c < 3; ++c) {
48            if (board[r][c] == ' ') {
49                return CONTINUE; // game is not finished
50            }
51        }
52    }
53
54    return DRAW; // game is a draw
55 }

-- void TicTacToe::printBoard() const {
56     cout << "   0   1   2\n\n";
57
58     for (int r{0}; r < 3; ++r) {
59         cout << r;
60
61         for (int c = 0; c < 3; ++c) {
62             cout << setw(3) << board[r][c];
63
64             if (c != 2) {
65                 cout << " |";
66             }
67         }
68
69         if (r != 2) {
70             cout << "\n ____|____|____\n      |      \n";
71         }
72     }
73
74     cout << "\n\n";
75 }
76
77 }

```

```

79 void TicTacToe::makeMove() {
80     printBoard();
81
82     while (true) {
83         if (xoMove('X')) {
84             break;
85         }
86         else if (xoMove('O')) {
87             break;
88         }
89     }
90 }
91
92 bool TicTacToe::xoMove(char Player) {
93     int x;
94     int y;
95
96     do {
97         cout << "Player " << Player << " enter move: ";
98         cin >> x >> y;
99         cout << '\n';
100    } while (!validMove(x, y));
101
102    board[x][y] = Player;
103    printBoard();
104    Status xoStatus = gameStatus();
105
106    if (xoStatus == WIN) {
107        cout << "Player " << Player << " wins!\n";
108        return true;
109    }
110    else if (xoStatus == DRAW) {
111        cout << "Game is a draw.\n";
112        return true;
113    }
114    else { // CONTINUE
115        return false;
116    }
117 }

```

```

1 // driver.cpp: use of TicTacToe class
2 #include "TicTacToe.h" // include definition of class TicTacToe
3
4 int main() {
5     TicTacToe g; // creates object g of class TicTacToe
6     g.makeMove(); // invokes function makeMove
7 }

```

```

1 # Makefile for the Rectangle example
2
3 CC=g++
4 CFLAGS=-std=c++11
5 DEPS = TicTacToe.h
6 OBJ = TicTacToe.o driver.o
7
8 %.o: %.cpp $(DEPS)
9         $(CC) -c -o $@ $< $(CFLAGS)
10
11 driver: $(OBJ)
12         $(CC) -o $@ $^ $(CFLAGS)
13

```

LEZIONE 15

In questa lezione vediamo l'overloading degli operatori e la gestione dinamica della memoria.

OVERLOADING

Abbiamo cominciato a vedere la scorsa lezione l'utilizzo di un oggetto all'interno di un'espressione. In generale abbiamo vari operatori che operano normalmente sui dati fondamentali, come somma, differenza, operatori relazionali, noi potremmo avere la possibilità di utilizzare oggetti *custom* che utilizziamo all'interno di espressioni. Questo lo facciamo cercando di riscrivere e ridefinire gli operatori (ovvero farne l'*overloading*) in base agli oggetti su cui questi operatori sono applicati. Un esempio già lo abbiano visto quando abbiamo parlato delle stringhe. Quando sommiamo 2 stringhe di fatto facciamo non la somma aritmetica ma la concatenazione; quindi l'operatore somma viene riscritto (overloaded) in base al fatto che è applicato ad oggetti di tipo stringa. Allo stesso modo potremmo fare una classe polinomio e fare l'overloading di uno specifico operatore e idem per una classe matrice. Ma potremmo anche effettuare un confronto tra delle date se abbiamo una classe di tipo Date.

Classe Date

```
43 bool Date::operator<(const Date& second) const {
44     if (year < second.year)
45         return true;
46     if (year == second.year && month < second.month)
47         return true;
48     if (year == second.year && month == second.month
49                     && day < second.day)
50         return true;
51     return false;
52 }
```

“operator” è una parola chiave del C++.

Definito l'operatore < a questo punto se d1, d2 sono due oggetti della classe Date allora d1<d2 verrà interpretato come d1.operator < (d2). Quindi viene interpretato come invocazione della funzione membro operatore <, che ha come argomento d2, sull'oggetto d1.

year, month e day sono le variabili membro dell'oggetto su cui viene invocato l'operatore <.

Ci sono però delle complicazioni.

Es:

Spesso alcuni operatori conviene scriverli come funzioni *friend* piuttosto che come funzioni membro. Come mai? Beh, supponiamo di voler fare la somma tra una data ed un intero, per aggiungere un certo numero di giorni ad una data. Definire la funzione come friend ci permette di poter fare sia $d+7$ che $7+d$, dove 7 è un intero a caso e d è una data; ovvero ci permette di avere la libertà di porre l'oggetto sia a destra che a sinistra di un operatore binario. In realtà l'utilità di *friend* sta soprattutto nel poter sfruttare la funzione privata *helpIncrement()*.

```
class Date {
    friend std::ostream& operator<<(std::ostream&, const Date&);
    friend Date operator+(int ,const Date& );
    friend Date operator+(const Date&,int );

54 // add a specific number of days
55 Date operator+(int numdays,const Date& dd)
56 {
57     Date newdate = dd;
58     for (int i = 1 ; i <= numdays ; i++)
59         newdate.helpIncrement();
60     return newdate;
61 }
62
63 // add the invoking object and a specific number of days
64 Date operator+(const Date& dd,int numdays)
65 {
66     Date newdate = dd;
67     for (int i = 1 ; i <= numdays ; i++)
68         newdate.helpIncrement();
69     return newdate;
70 }

133 // overloaded output operator
134 ostream& operator<<(ostream& output, const Date& d) {
135     static string monthName[13]("", "January", "February",
136         "March", "April", "May", "June", "July", "August",
137         "September", "October", "November", "December");
138     output << d.day << ' ' << monthName[d.month] << ' ' << d.year;
139     return output; // enables cascading
140 }
```

In particolare la funzione friend che descrive l'operatore '+' la scrivo con 2 signature diverse: una volta con (int, Date) e l'altra volta con (Date, int). Con l'operatore < questo problema naturalmente non c'era.

Le funzioni friend, lo abbiamo visto ieri, hanno come particolarità quella di poter accedere ai dati membro della classe (in questo caso Date).

Nella definizione dell'operatore + abbiamo definito una variabile, locale alla funzione, di nome *newdate* e che non è altro che un oggetto della classe Date. Questa variabile l'abbiamo inizializzata con i valori di *dd* tramite quindi una copia memberwise e la funzione ritorna proprio questa variabile, o meglio quest'oggetto di tipo Date definito localmente nella funzione. La funzione chiamante avrà allocato spazio nello stack

frame per contenere un valore di ritorno e questo valore di ritorno sarà appunto l'oggetto definito nella funzione e di tipo Date.

Cosa sarebbe successo se avessimo passato un riferimento ad un oggetto Date e non un oggetto Date? Questo avrebbe generato un errore poiché lo scopo di *newdate* è la funzione operator + dentro cui è definita. Alla fine dell'esecuzione della funzione operator +, *newdate* non esisterà più e quindi il riferimento alla variabile non avrà più senso. Quindi non avremmo potuto passare *newdate* per riferimento, ovvero non avremmo potuto scrivere:

```
friend Date& operator+(...)
```

sarebbe stato un errore perché il valore di ritorno viene copiato nel pop dello stack frame e quindi utilizzato nella funzione che ha invocato questo operatore.

Analogamente all'operatore +, posso ridefinire l'operatore di trasferimento che prende in ingresso un output stream ed una data e restituisce un output stream.

Vediamo tutto il programma il cui scopo è quello di definire la classe Date e di testare le sue funzionalità.

```
1 // Date.h
2 // Date class definition with overloaded increment operators.
3 #ifndef DATE_H
4 #define DATE_H
5
6 #include <array>
7 #include <iostream>
8
9 class Date {
10     friend std::ostream& operator<<(std::ostream&, const Date&);
11     friend Date operator+(int ,const Date& );
12     friend Date operator+(const Date&,int );
13 public:
14     Date(int d = 1, int m = 1, int y = 1900); // default constructor
15     void setDate(int, int, int); // set month, day, year
16     Date& operator++(); // prefix increment operator
17     Date operator++(int); // postfix increment operator
18     Date& operator+=(unsigned int); // add days, modify object
19     static bool leapYear(int); // is year a leap year?
20     bool endOfMonth(int) const; // is day at the end of month?
21     bool operator<(const Date&) const;
22 private:
23     unsigned int day;
24     unsigned int month;
25     unsigned int year;
26     static const std::array<unsigned int, 13> days; // days per month
27     void helpIncrement(); // utility function for incrementing date
28 };
29
30 #endif
```

```

1 // Date.cpp
2 // Date class member- and friend-function definitions.
3 #include <iostream>
4 #include <string>
5 #include "Date.h"
6 using namespace std;
7
8 // initialize static member; one classwide copy
9 const array<unsigned int, 13> Date::days{
10     0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
11
12 // Date constructor
13 Date::Date(int day, int month, int year) {
14     setDate(day, month, year);
15 }
16
17 // set month, day and year
18 void Date::setDate(int dd, int mm, int yy) {
19     if (mm >= 1 && mm <= 12) {
20         month = mm;
21     }
22     else {
23         throw invalid_argument("Month must be 1-12");
24     }
25
26     if (yy >= 1900 && yy <= 2100) {
27         year = yy;
28     }
29     else {
30         throw invalid_argument("Year must be >= 1900 and <= 2100");
31     }
32
33 // test for a leap year
34 if ((month == 2 && leapYear(year) && dd >= 1 && dd <= 29) ||
35     (dd >= 1 && dd <= days[month])) {
36     day = dd;
37 }
38 else {
39     throw invalid_argument(
40         "Day is out of range for current month and year");
41 }
42 }
43 bool Date::operator<(const Date& second) const {
44     if (year < second.year)
45         return true;
46     if (year == second.year && month < second.month)
47         return true;
48     if (year == second.year && month == second.month
49             && day < second.day)
50         return true;
51     return false;
52 }
53
54 // add a specific number of days
55 // add a specific number of days
56 Date operator+(int numdays, const Date& dd)
57 {
58     Date newdate = dd;
59     for (int i = 1; i <= numdays; i++)
60         newdate.helpIncrement();
61     return newdate;
62 }
63 // add the invoking object and a specific number of days
64 Date operator+(const Date& dd, int numdays)
65 {
66     Date newdate = dd;
67     for (int i = 1; i <= numdays; i++)
68         newdate.helpIncrement();
69     return newdate;
70 }
71
72 // overloaded prefix increment operator
73 Date& Date::operator++() {
74     helpIncrement(); // increment date
75     return *this; // reference return to create an lvalue
76 }
77
78 // overloaded postfix increment operator; note that the
79 // dummy integer parameter does not have a parameter name
80 // Date Date::operator++(int) {
81 Date Date::operator++(int) {
82     Date temp{*this}; // hold current state of object
83     helpIncrement();
84
85     // return unincremented, saved, temporary object
86     return temp; // value return; not a reference return
87 }
88
89 // add specified number of days to date
90 Date& Date::operator+=(unsigned int additionalDays) {
91     for (unsigned int i = 0; i < additionalDays; ++i) {
92         helpIncrement();
93     }
94
95     return *this; // enables cascading
96 }
97
98 // if the year is a leap year, return true; otherwise, return false
99 bool Date::leapYear(int testYear) {
100     return (testYear % 400 == 0 ||
101            (testYear % 100 != 0 && testYear % 4 == 0));
102 }
103
104 // determine whether the day is the last day of the month
105 bool Date::endOfMonth(int testDay) const {
106     if (month == 2 && leapYear(year)) {
107         return testDay == 29; // last day of Feb. in leap year
108     }
109     else {
110         return testDay == days[month];
111     }
112 }

```

```

1 // testdate.cpp
2 // Date class test program.
3 #include <iostream>
4 #include "Date.h" // Date class definition
5 using namespace std;
6
7 int main() {
8     Date d1{25, 12, 2020}; // December 25, 2010
9     Date d2; // defaults to January 1, 1900
10
11    cout << "d1 is " << d1 << "\nd2 is " << d2;
12    cout << "\n\n d1 += 7 is " << (d1 += 7);
13
14    d2.setDate(28, 2, 2008);
15    cout << "\n\n d2 is " << d2;
16    cout << "\n++d2 is " << ++d2 << " (leap year allows 29th)";
17
18    Date d3{13, 1, 2010};
19
20    cout << "\n\nTesting the prefix increment operator:\n"
21        << " d3 is " << d3 << endl;
22    cout << "++d3 is " << ++d3 << endl;
23    cout << " d3 is " << d3;
24
25    cout << "\n\nTesting the postfix increment operator:\n"
26        << " d3 is " << d3 << endl;
27    cout << "d3++ is " << d3++ << endl;
28    cout << " d3 is " << d3 << endl;
29
30    std::string check = (d2 < d3)? " is ":"is not ";
31    cout << "\n\nTestings relational operators:\n"
32        << d2 << check << "less than " << d3 << "\n";
33
34    Date d4 = d3+10;
35    cout << "\n\nTesting summing numberofdays\n"
36        << d4 << " = "<< d3 << " + 10 days \n";
37 }

```

```

1 # Makefile for the Date example
2
3 CC=g++
4 CFLAGS=-std=c++11
5 DEPS = Date.h
6 OBJ = Date.cpp testdate.o
7
8 %.o: %.cpp $(DEPS)
9     $(CC) -c -o $@ $< $(CFLAGS)
10
11 testdate: $(OBJ)
12     $(CC) -o $@ $^ $(CFLAGS)

```

Le funzioni friend ovviamente non fanno parte della classe e di conseguenza non possono usufruire del ritorno di *this* pointer poiché non vedono proprio l'oggetto in questione. Da ciò segue che quando definiamo funzioni friend prevediamo che non ci possa essere cascata di queste funzioni.

L'operatore `+=` definito è leggermente diverso da quello `+`: il primo è invocato sull'oggetto attuale su cui si invoca la funzione `helpincrement()` ed a cui si ritorna tramite il puntatore `this`; il secondo invece utilizza un nuovo oggetto `newdate` definito localmente alla funzione su cui viene invocato `helpincrement()` e che sarà proprio ciò che la funzione restituisce al programma client chiamante.

Il file eseguibile sarà questo:

```
8 $ ./testdate
9 d1 is 25 December 2020
10 d2 is 1 January 1900
11
12 d1 += 7 is 1 January 2021
13
14   d2 is 28 February 2008
15 ++d2 is 29 February 2008 (leap year allows 29th)
16
17 Testing the prefix increment operator:
18   d3 is 13 January 2010
19 ++d3 is 14 January 2010
20   d3 is 14 January 2010
21
22 Testing the postfix increment operator:
23   d3 is 14 January 2010
24 d3++ is 14 January 2010
25   d3 is 15 January 2010
26
27
28 Testings relational operators:
29 29 February 2008 is less than 15 January 2010
30
31
32 Testing summing numberofdays
33 25 January 2010 = 15 January 2010 + 10 days
34 $
```

Come esercizio fare ciò:

Exercise

- Extend the Date example
- Implement operators `>`, `>=`, `<=`, `!=`
- Try to implement the decrement operator as well

GESTIONE DINAMICA DELLA MEMORIA

La gestione dinamica della memoria serve per controllare l'allocazione e la deallocazione di memoria per gli oggetti e per i tipi custom che definiamo. La gestione dinamica della memoria avviene tramite gli operatori **new** e **delete**.

“new” serve per allocare dinamicamente una quantità specifica di memoria, “delete” per rimuovere questa porzione di memoria.

Gli oggetti sono allocati in un’area di memoria detta *heap* (mucchio), che è una struttura dati ad albero da cui si può estrarre in tempi logaritmici gli oggetti; è una struttura dati alternativa allo stack, alla coda, alla lista, etc.

Quando creiamo un oggetto richiediamo implicitamente con la dichiarazione dell’oggetto una o più “caselle” di memoria al sistema operativo; quando utilizziamo *new* lo facciamo esplicitamente.

Esempio di dichiarazione dinamica:

```
Date* datePtr{new Date};
```

In questo modo stiamo dichiarando un puntatore ad una variabile di tipo *Date*, ovvero *datePtr*, in modo tale che questo puntatore punti ad un’area di memoria che contiene la quantità di memoria libera necessaria per rappresentare una variabile di tipo *Date*. Quindi *new* alloca spazio per un oggetto di tipo *Date*, non inizializza a niente l’oggetto ma chiama il costruttore di default della classe *Date* per inizializzarlo e restituisce un puntatore all’oggetto specificato.

Potrebbe capitare che siamo ingordi di memoria ed il nostro programma continua ad effettuare delle richieste al SO con dei new e il SO rifiuta di allocare spazio in memoria; in tal caso viene lanciata un’eccezione del tipo *bad_alloc*.

Quando abbiamo terminato l’utilizzo dell’oggetto che ha allocato memoria dobbiamo rilasciare questo spazio al SO tramite *delete*.

Supponiamo allora di voler allocare un array di coefficienti di 10 double. Lo facciamo così:

```
double *coefs{new double[10]{};}
```

coefs è una variabile di tipo *double** che viene istanziata con il puntatore restituito dall’operatore *new*; *new* alloca in memoria uno spazio per contenere 10 double ({} serve per inizializzare gli elementi a 0). Per deallocare facciamo così:

```
delete[] coefs;
```

Se il puntatore punta ad un array di oggetti allora la delezione chiama prima il distruttore di default di ciascun oggetto e poi viene deallocata la memoria.

Oss: usare sempre [] in modo da deallocate non il puntatore ma tutti gli elementi dell'array.

Polynomial class

Es: vogliamo rappresentare dei polinomi. Un polinomio ha bisogno di un grado e dei coefficienti. Un polinomio di grado 2 ha 3 coefficienti, uno di grado 3 ha 4 coefficienti e così via (c'è il termine noto). Allora usiamo 2 variabili private per rappresentare il polinomio, di cui una variabile per il numero di coefficienti (che sarà =grado+1).

File.h:

```
1 #ifndef POLYNOMIAL_H
2 #define POLYNOMIAL_H
3
4 #include <iostream>
5 using namespace std;
6
7 class Polynomial {
8
9     friend bool operator==(const Polynomial & lhs, const Polynomial & rhs);
10
11 public:
12
13     // constructors
14     Polynomial();
15     Polynomial(double coefficients[], int number);
16     Polynomial(const Polynomial & rhs);
17     explicit Polynomial(double const_term);
18
19     // destructor
20     ~Polynomial();
21
22     // named member functions
23     int degree() const { return size - 1; }
24     void print(ostream & out = cout) const;
25
26     // assignment operators
27     Polynomial & operator=(const Polynomial & rhs);
28     Polynomial & operator+=(const Polynomial & rhs);
29     Polynomial & operator*=(double rhs);
30
31     private:
32         int size;      // size of the coefs array (= degree + 1)
33         double * coefs; // coefs will be an array
34     };
35
36 Polynomial operator+(const Polynomial & lhs, const Polynomial & rhs);
37
38 Polynomial operator*(const Polynomial & lhs, double rhs);
39 Polynomial operator*(double lhs, const Polynomial & rhs);
40
41 ostream & operator<<(ostream & out, const Polynomial & p);
42
43 #endif
```

L'allocazione dinamica la usiamo perché vogliamo che il grado del polinomio sia eventualmente qualsiasi. Quindi i coefficienti del polinomio li rappresentiamo come un built-in array. Definita la classe polinomio vogliamo poter essere in grado di scrivere espressioni che coinvolgano i polinomi, sfruttando ad esempio la somma tra polinomi, il confronto, il prodotto di un polinomio per un double (o viceversa). Tutte queste funzioni sono definite esternamente (non come friends perché in realtà sfruttano le funzioni definite all'interno della classe come += e *=).

In questo caso dobbiamo anche scrivere dei costruttori copia perché quello di default non va più bene: se costruiamo un polinomio a partire da uno esistente il costruttore copia di default copia gli elementi membro per membro. Quindi nel momento in cui invoco il costruttore con un costruttore copia di default per un polinomio, il nuovo polinomio viene creato con i dati del precedente; e quindi nella variabile *size* ci va lo stesso numero di coefficienti e nella variabile *coefs*, che è un puntatore a double, ci va il valore della variabile *coefs* del polinomio che sto copiando. Quindi succede che la variabile *coefs* dei due polinomi punta alla stessa area di memoria. Per evitare ciò dobbiamo scrivere esplicitamente il costruttore copia (riga 16 del file.h). Un costruttore copia è fatto allo stesso modo di un costruttore: avrà lo stesso nome della classe ma ci saranno poi delle parentesi tonde che contengono un riferimento alla classe). Allo stesso modo non possiamo fidarci del memberwise assignment nemmeno per l'operatore di assegnazione ma dobbiamo implementare una funzione.

Nel file .cpp individuiamo più “fasi”:

Constructors

```

1 #include <iostream>
2 #include "Polynomial.h"
3
4 // Constructor
5 // Creates a default polynomial p of the form p(x) = 0.0
6 Polynomial::Polynomial() : size{1}, coefs{new double[1]} {
7     coefs[0] = 0.0;
8 }
9
10 // Constructor
11 // Given an array of coefficients C (and it's size N) creates a polynomial
12 // p(x) = C[N-1]x^(N-1) + ... + C[2]x^2 + C[1]x + C[0]
13 Polynomial::Polynomial(double coefficients[], int number) :
14     size{number}, coefs{new double[number]} {
15     for (int i = 0; i < size; i++) {
16         coefs[i] = coefficients[i];
17     }
18 }
19
20 // Constructor
21 // Given a constant term A, creates the polynomial p(x) = A
22 Polynomial::Polynomial(double const_term) : size{1}, coefs{new double[1]} {
23     coefs[0] = const_term;
24 }
25

```

Copy constructor, Destructor, and =

```
26 // Copy constructor
27 // Creates a polynomial from the given polynomial
28 Polynomial::Polynomial(const Polynomial & rhs) :
29     size{rhs.size}, coefs{new double[rhs.size]} {
30     for (int i = 0; i < size; i++) {
31         coefs[i] = rhs.coefs[i];
32     }
33 }
34
35 // Destructor
36 Polynomial::~Polynomial() {
37     delete [] coefs;
38 }
39
40
41 // Overload assignment =
42 Polynomial & Polynomial::operator=(const Polynomial & rhs) {
43     if (this == &rhs) {
44         return *this;
45     }
46
47     else {
48         delete [] coefs;
49         coefs = new double[rhs.size];
50         size = rhs.size;
51         for (int i = 0; i < size; i++) {
52             coefs[i] = rhs.coefs[i];
53         }
54     }
55     return *this;
56 }
```

Overloaded operators (Member functions)

```
58 // Overload assignment +=
59 Polynomial & Polynomial::operator+=(const Polynomial & rhs) {
60     int newSize = (rhs.size > size) ? rhs.size : size;
61     double *newCoef = new double [newSize];
62
63     for (int i = 0; i < newSize; i++) {
64         newCoef[i] = 0;
65     }
66     for (int i = 0; i < rhs.size; i++) {
67         newCoef[i] += rhs.coefs[i];
68     }
69     for (int i = 0; i < size; i++) {
70         newCoef[i] += coefs[i];
71     }
72     delete [] coefs;
73     coefs = newCoef;
74     size = newSize;
75     return *this;
76 }
77
78 // Overload assignment *= so it supports scalar multiplication
79 Polynomial & Polynomial::operator*=(double rhs) {
80     for (int i = 0; i < size; i++) {
81         coefs[i] *= rhs;
82     }
83     return *this;
84 }
```

Relational operator ==

```
86 // Overload ==
87 bool operator==(const Polynomial & lhs, const Polynomial & rhs) {
88     if (lhs.size != rhs.size) {
89         return false;
90     }
91
92     for (int i = 0; i < lhs.size; i++) {
93         if (lhs.coefs[i] != rhs.coefs[i]) {
94             return false;
95         }
96     }
97     return true;
98 }
```

Overloaded operators (non-member functions)

```
100 // Overload +
101 Polynomial operator+(const Polynomial & lhs, const Polynomial & rhs) {
102     Polynomial answer{lhs};
103     answer += rhs;
104     return answer;
105 }
106
107 // Overload * so it supports scalar multiplication. Note that
108 // we overload it twice so we can do either:
109 //     polynomial * scalar
110 // or
111 //     scalar * polynomial
112
113 Polynomial operator*(const Polynomial & lhs, double rhs) {
114     Polynomial answer{lhs};
115     answer *= rhs;
116     return answer;
117 }
118
119 Polynomial operator*(double lhs, const Polynomial & rhs) {
120     Polynomial answer{rhs};
121     answer *= lhs;
122     return answer;
123 }
124
125 // Prints the polynomial to the given ostream.
126 void Polynomial::print(std::ostream & out) const {
127     if (size == 0) {
128         return;
129     }
130
131     for (int i = size - 1; i > 0; i--)
132         out << coefs[i] << "x^" << i << " + ";
133     out << coefs[0];
134 }
135
136 // Overload << for output
137 std::ostream & operator<<(std::ostream & out, const Polynomial & p) {
138     p.print(out);
139     return out;
140 }
```

C'è poi il programma che testa la classe:

Test polynomial

```
1 # Makefile for the Polynomial example
2
3 CC=g++
4 CFLAGS=-std=c++11
5 DEPS = Polynomial.h
6 OBJ = Polynomial.o testPolynomial.o
7
8 %.o: %.cpp $(DEPS)
9         $(CC) -c -o $@ $< $(CFLAGS)
10
11 testPolynomial: $(OBJ)
12         $(CC) -o $@ $^ $(CFLAGS)

./testPolynomial
p1 is 4.4x^3 + 3.3x^2 + 2.2x^1 + 1.1
p2 is 4.4x^3 + 3.3x^2 + 2.2x^1 + 1.1
p1 is 13.2x^3 + 9.9x^2 + 6.6x^1 + 3.3
p2 is 4.4x^3 + 3.3x^2 + 2.2x^1 + 1.1
p3 is 8.8x^3 + 6.6x^2 + 4.4x^1 + 2.2
3x^4 + 3x^3 + 3x^2 + 3x^1 + 3
3x^4 + 3x^3 + 3x^2 + 3x^1 + 11
1.5x^4 + 1.5x^3 + 1.5x^2 + 1.5x^1 + 5.5
6x^4 + 6x^3 + 6x^2 + 6x^1 + 22
p4 == p4
```

Questo programma è il classico esempio di programma che dobbiamo saper fare.

LEZIONE 16

Continuiamo con l'esempio della classe polinomio. Gli screen del programma stanno alla lezione precedente.

Abbiamo definito 3 possibili costruttori: uno senza argomenti che inizializza size a 0 ed il primo coefficiente a 0, uno con 2 argomenti che corrispondono ad un vettore di coefficienti e ad un numero che sarà assegnato a size, un costruttore con un unico argomento che è un double (polinomio costante di grado 0) ed infine un costruttore di copia; riconosciamo che è un costruttore di copia perché prende in ingresso un riferimento ad un oggetto della stessa classe.

```
13 // constructors
14 Polynomial();
15 Polynomial(double coefficients[], int number);
16 Polynomial(const Polynomial & rhs);
17 explicit Polynomial(double const_term);
```

Abbiamo poi bisogno ovviamente di un distruttore. Il compilatore ci mette poi a disposizione un costruttore di copia che copia “bit a bit” il contenuto di un oggetto in un altro oggetto. Questo però può creare problemi se stiamo allocando memoria poiché nel nostro caso ad esempio il puntatore al nuovo oggetto viene istanziato con lo stesso puntatore all’array dei coefficienti in ingresso, ovvero a *rhs*. Se scriviamo un costruttore copia dovremo ovviamente effettuare l’overloading per l’operatore di assegnazione “=”.

Questa è la cosiddetta *regola del 3 (THE BIG THREE)*: se creo un costruttore copia allora sicuramente creerò un distruttore e riscriverò l’operatore di assegnazione. Valgono anche le altre implicazioni.

Abbiamo definito alcune le funzioni operatori +, * etc. fuori dallo scopo della classe e nemmeno friends; come mai? Perché queste funzioni non hanno bisogno di accedere alle variabili membro della classe *Polynomial* in quanto sfruttano funzioni membro che accedono a queste variabili e sfruttano il costruttore copia da noi definito. Ad esempio la funzione + che somma 2 polinomi sfrutta la funzione membro += definita precedentemente. Notiamo che stesso nella funzione + non restituiamo un riferimento; come mai? Beh, *answer* è una variabile locale. Non avrebbe senso restituire un riferimento a qualcosa che muore subito dopo che la funzione termina. Per questo faccio *return answer* invocando implicitamente il copy constructor; infatti *answer* viene copiata alla fine della funzione nella variabile definita nella funzione client di questo operatore, dopodiché muore. Quindi il copy constructor viene invocato sia alla riga 102 che alla riga 104.

```
100 // Overload +
101 Polynomial operator+(const Polynomial & lhs, const Polynomial & rhs) {
102     Polynomial answer{lhs};
103     answer += rhs;
104     return answer;
105 }
```

Stesse cose succedono per gli operatori * (prodotto per scalare).

L'operatore == invece per come è scritto ha bisogno di accedere a delle variabili membro e quindi è dichiarata come friend.

Cosa interessantissima: l'uso di **explicit**. Se nel driver provassimo a scrivere:

```
31 // Uncommenting these lines and compiling will result in a .
32 // compilation error. Why?
33 // int x = 2;
34 // p5 += x;
35 // cout << p5 << endl << x << endl;
```

Le istruzioni dalla riga 33 alla 35 genererebbero un errore. Perché? Proprio perché abbiamo utilizzato *explicit* per il costruttore:

```
explicit Polynomial(double const_term);
```

L'errore sarebbe dovuto al fatto che poniamo $p5=p5+2$, dove 2 è un intero. Quindi vuol dire che andrebbe fatta una conversione implicita da *int* a *double*, ovvero una promozione. Questo tipicamente avviene, ma quando utilizziamo *explicit* viene vietata la promozione implicita. Se levassimo quindi *explicit* il programma funzionerebbe perché ci sarebbe una promozione implicita da *int* a *double* e quindi da *double* a polinomio. Infatti so convertire da *double* a polinomio, ovvero tramite quel costruttore di sopra preceduto da *explicit*. Senza *explicit* automaticamente qualsiasi *double* sarebbe promosso ad essere un polinomio; e quindi è questo il motivo per cui mettiamo *explicit!* Avere un *double* interpretato come polinomio non è quello che vogliamo. In generale l'uso di *explicit* evita situazioni “dangerous”.

In generale il costruttore di copia è chiamato quando:

- Inizializziamo;
- Passiamo un oggetto per valore;
- Ritorniamo un oggetto per valore.

Nella composizione di oggetti quando non specifichiamo il copy constructor o l'operatore =, il compilatore invoca implicitamente un copy constructor che semplicemente fa la copia di ogni campo (*shallow copying*); il puntatore all'area di memoria heap viene semplicemente copiato, senza copiare l'area di memoria heap.

Un copy constructor è spesso nella forma X(&X).

Per prevenire che un oggetto sia passato per valore c'è una tecnica semplice: basta definire il costruttore nella sezione privata della classe. In questo modo se l'utente proverà a passare o restituire un oggetto per valore il compilatore mostrerà un messaggio di errore perché il costruttore di copia è privato.

Es:

```
1 // Preventing copy-construction
2 class NoCC {
3     int i;
4     NoCC(const NoCC&); // No definition
5 public:
6     NoCC(int ii = 0) : i(ii) {}
7 };
8
9 void f(NoCC);
10
11 int main() {
12     NoCC n;
13 //! f(n); // Error: copy-constructor called
14 //! NoCC n2 = n; // Error: c-c called
15 //! NoCC n3(n); // Error: c-c called
16 } //:~
```

Esercizi da fare:

Exercise

- Extend the Polynomial class to implement the product between polynomials

Exercise

- Implement a Matrix class that stores a $n \times m$ matrix of double
- Overload sum (+), difference (-) and product (*) operators;

Il secondo lo facciamo la prossima lezione.

LEZIONE 17

Vediamo il programma sulle matrici. Il progetto è composto da Matrix.h, Matrix.cpp e dal driver testMatrix.cpp.

```
//Matrix.h
#ifndef MATRIX_H
#define MATRIX_H

class Matrix {
public:
    //constructors
    Matrix(int=3,int=3);
    //the Big Three
    Matrix(const Matrix&); //copy constructor
    ~Matrix();
    Matrix& operator=(const Matrix&);

    //operators
    Matrix& operator+(const Matrix&);
    Matrix& operator-(const Matrix&);
    Matrix operator*(const Matrix&);

    //get and set operators
    Matrix& setValue(int, int, double); //setta i valori della matrice e restituisce la nuova matrice
    double getValue(int,int); //restituisce il valore della matrice in una certa posizione

private:
    int nrow;
    int ncol;
    double** array;
};

#endif

.....
.....
.....



//Matrix.cpp
```

```

#include"Matrix.h"
#include<stdexcept>

//constructor
Matrix::Matrix(int r, int c){
    nrow=r;
    ncol=c;
    //alloco spazio per contenere una matrice di dimensione (nrow,ncol)
    array=new double* [nrow]; //in questo modo istanzio un vettore che contiene tanti puntatori a double quante
                                //sono le righe; ora per ciascuno degli elementi double* (ovvero per ciascuna riga)
                                //alloco memoria. In pratica vedo una matrice come un vettore di righe dove ogni riga punta alle c colonne che sono quelli della riga;
                                //quindi ho r righe tutte lunghe c elementi, ovvero il numero di colonne della matrice
    for (int i=0; i<r; ++i) {
        array[i]= new double[ncol]{}; //inizializzo anche a 0 ogni elemento con {}
    }
}

//Copy Constructor
Matrix::Matrix(const Matrix& m ) {
    nrow=m.nrow;
    ncol=m.ncol;
    //allocate memory
    array=new double* [m.nrow];
    for (int i=0; i<nrow; ++i) {
        array[i]=new double[ncol];
    }
    for (int i=0; i<nrow; ++i) {
        for (int j=0; j<ncol; ++j) {
            array[i][j]=m.array[i][j];
        }
    }
}

//Destructor
//cancello gli elementi al contrario (--> prima le colonne e poi le righe)
Matrix::~Matrix() {
    for (int i=0; i<nrow; ++i){
        delete[] array[i];
    }
    delete[] array;
}

//Operator =
Matrix& Matrix::operator=(const Matrix& m) {
    if (this == &m) {
        return *this;
    }
}

```

```

//release memory
for (int i=0; i< nrow; ++i) {
    delete[] array[i];
}
delete[] array;

nrow=m.nrow;
ncol=m.ncol;
array=new double* [m.nrow];
for (int i=0; i<nrow; ++i) {
    array[i]=new double[m.ncol];
}
for (int i=0; i<nrow; ++i) {
    for (int j=0; j<ncol;++j) {
        array[i][j]=m.array[i][j];
    }
}
return *this;
}

//Operator +
Matrix& Matrix::operator+(const Matrix& m){
    if (ncol==m.ncol && nrow==m.nrow) {
        for (int i=0; i<m.nrow; ++i){
            for (int j=0;j<m.ncol;++j) {
                array[i][j]+=m.array[i][j];
            }
        }
        return *this;
    }
    else throw std::invalid_argument ("Matrix 1 and 2 must have the same number of
rows and columns");
}

Matrix Matrix::operator*(const Matrix&m){
    if(ncol!=m.nrow) {
        throw std::invalid_argument("Matrix must be compatible");
    }
    Matrix newMat(nrow,m.ncol);
    for (int i=0; i<newMat.nrow; ++i){
        for (int k=0;k<newMat.ncol;k++) {
            //compute the scalar product between row i of object and column k
            newMat.array[i][k]=0.0;
            for (int j=0; j<ncol;++j){
                newMat.array[i][k]+=array[i][j]*m.array[j][k];
            }
        }
    }
    return newMat; //non posso restituire un riferimento poiché newMat viene eliminato alla fine della funzione
}

```

```

/*
 *   *   *
 *   *   *   *   *
 *   *   *   *   ==   *
 *   *   *   *   *
               (4,3)        (3,2)        (4,2)

*/
//Operator -
Matrix& Matrix::operator-(const Matrix& m){
    if (ncol==m.ncol && nrow==m.nrow) {
        for (int i=0; i<m.nrow; ++i){
            for (int j=0;j<m.ncol;++j) {
                array[i][j]=array[i][j]-m.array[i][j];
            }
        }
        return *this;
    }
    else throw std::invalid_argument ("Matrix 1 and 2 must have the same number of
rows and columns");
}

//Set e get
Matrix& Matrix::setValue(int n, int m, double value) {
    //dovremmo anche effettuare il check che n ed m siano indici compresi nella matr
ice
    array[n][m]=value;
    return *this;
}

double Matrix::getValue(int n, int m) {
    return array[n][m];
}

.
.
.
.

//testMatrix.cpp

```

```

#include"Matrix.h"
#include<iostream>
#include<cstdlib>
using std::cout; using std::endl;

int main(){
    int r=4;
    int c=4;
    Matrix A(r,c);
    Matrix B(r,c);
    Matrix C(r,c);
    srand(1);
    for (int i=0; i<r; i++){
        for (int j=0;j<c;j++){
            A.setValue(i,j,1.1);
            B.setValue(i,j,static_cast<double> (rand()%100)/100); //rand()%100 è un intero tra 0 e 100;
                                                                //divido per 100
così per ottenere un numero tra
                                                                //0 ed 1 e però p
rima devo fare una conversione
                                                                //da int a double
perché se no otterrei sempre 0
    }
}
cout<<"Matrix A "<<endl;
for (int i=0; i<r;i++){
    for (int j=0; j<c; j++) {
        cout<<A.getValue(i,j)<<"\t";
    }
    cout<<endl;
}
cout<<endl;
cout<<"Matrix B"<<endl;
for (int i=0; i<r; i++){
    for (int j=0;j<c;j++) {
        cout<<B.getValue(i,j)<<"\t";
    }
    cout<<endl;
}

cout<<"Matrix C before addition"<<endl;

for (int i=0; i<r; i++){
    for (int j=0; j<c; j++) {
        cout<<C.getValue(i,j)<<"\t";
    }
    cout<<endl;
}

```

```

C=A+B;

cout<<"Matrix C after addition"<<endl;
for (int i=0; i<r; i++){
    for (int j=0; j<c; j++) {
        cout<<C.getValue(i,j)<<"\t";
    }
    cout<<endl;
}

Matrix D=A*B;
cout<<"Matrix D after multiplication"<<endl;
for (int i=0; i<r; i++){
    for (int j=0; j<r; j++){
        cout<<D.getValue(i,j)<<"\t";
    }
    cout<<endl;
}
.
.
.
.
.
.

#Makefile

testMatrix:testMatrix.o Matrix.o
    g++ -o testMatrix testMatrix.o Matrix.o
testMatrix.o: testMatrix.cpp Matrix.h
    g++ -c testMatrix.cpp -std=c++11 -Wall -pedantic
Matrix.o: Matrix.cpp Matrix.h
    g++ -c Matrix.cpp -std=c++11 -Wall -pedantic

```

[./testMatrix.exe:](#)

Matrix A

1.1	1.1	1.1	1.1
1.1	1.1	1.1	1.1
1.1	1.1	1.1	1.1
1.1	1.1	1.1	1.1

Matrix B

0.41	0.67	0.34	0
0.69	0.24	0.78	0.58
0.62	0.64	0.05	0.45
0.81	0.27	0.61	0.91

Matrix C before addition

0.81	0.27	0.61	0.91
------	------	------	------

Matrix C before addition

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Matrix C after addition

1.51	1.77	1.44	1.1
1.79	1.34	1.88	1.68
1.72	1.74	1.15	1.55
1.91	1.37	1.71	2.01

Matrix D after multiplication

3.6242	2.6551	2.637	2.6756
4.1849	3.1777	2.7726	3.152
3.8743	2.7245	2.945	2.9372
4.4167	3.2456	3.0296	3.3932

Prima di passare avanti con gli argomenti, vediamo un ultimo esempio sull'uso della parola *Explicit* per costruttori con un unico argomento.

Es:

```
1 //A class without explicit one parameter constructor
2 // allow implicit conversion
3 #include <iostream>
4
5 using std::cout;
6
7 class Complex
8 {
9 private:
10     double real;
11     double imag;
12
13 public:
14     // Default constructor
15     Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}
16
17     // A method to compare two Complex numbers
18     bool operator == (Complex rhs) {
19         return (real == rhs.real && imag == rhs.imag)? true : false;
20     }
21 };
22
23 int main()
24 {
25     // a Complex object
26     Complex com1(3.0, 0.0);
27
28     if (com1 == 3.0) //implicit conversion
29         cout << "Same number\n";
30     else
31         cout << "Not Same number\n ";
32     return 0;
33 }
```

```
$ g++ Complex1.cpp -o Complex1 -std=c++11
$ ./Complex1
Same number
```

La classe Complex ha 2 variabili membro: parte reale e parte immaginaria; queste sono inizializzate a 0 per default con il costruttore di default.

In una espressione che richiede un Complex, come quella nella riga 28, visto che posso costruire un Complex a partire da un double il compilatore effettua una conversione implicita (una promozione): infatti confronto un Complex con un double e quindi il double viene promosso a Complex. E quindi viene invocato il default constructor che costruisce un oggetto Complex con un unico argomento che è il double 3.0 e che quindi viene interpretato come (3.0, 0.0). Questo però può generare anche effetti indesiderati ed allora per scongiurare questa situazione utilizziamo la parola chiave *explicit*: non diamo la possibilità al compilatore di effettuare conversioni esplicite. Nel nostro esempio se usassimo *explicit* avremmo un errore e questo errore andrebbe corretto invocando esplicitamente il costruttore con un unico argomento.

INHERITANCE & POLYMORPHISM

Abbiamo già visto che possiamo creare oggetti con al loro interno altri oggetti.

Ma oltre a questo possiamo creare delle classi che sono una specializzazione di una classe più generale e che quindi hanno un comportamento più specifico.

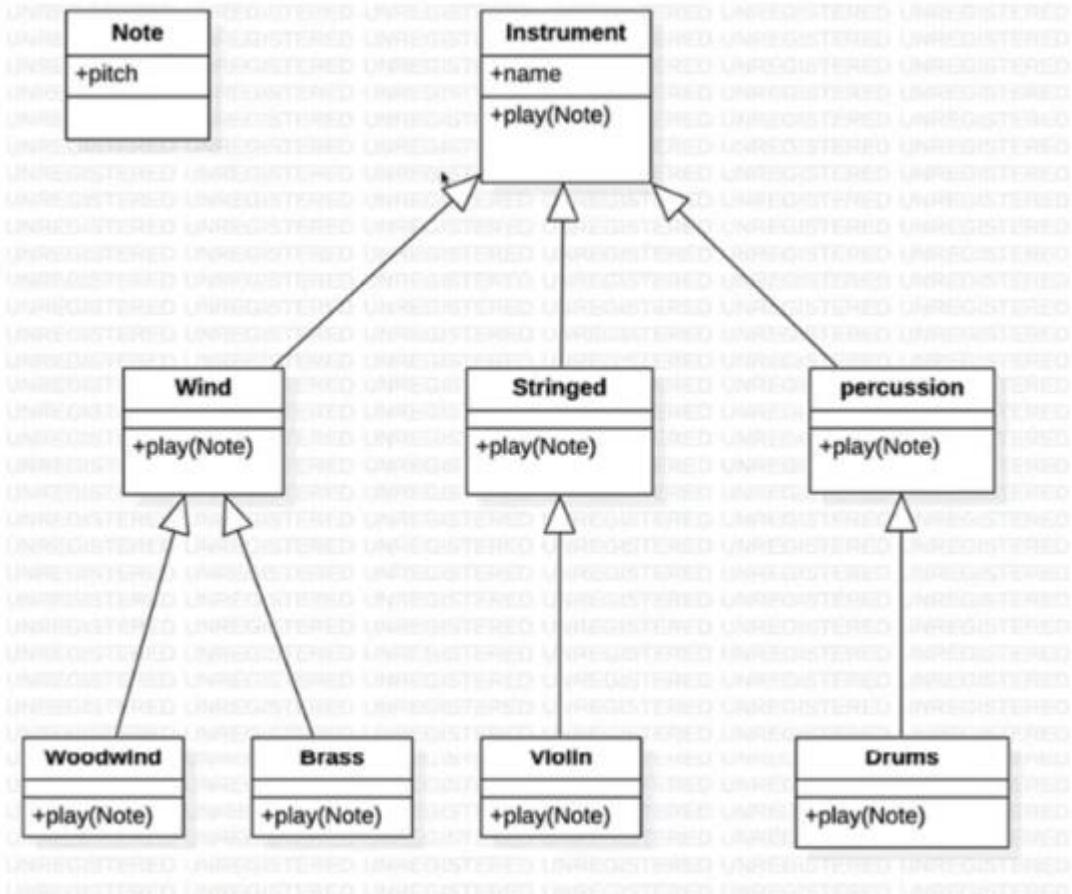
Possiamo allora costruire una gerarchia di classi in base alla specificità. Ad esempio possiamo costruire degli oggetti generici quali Impiegato, con data di nascita, nome, cognome, indirizzo etc.; potremmo poi specializzare vari tipi di impiegati come impiegati alle vendite, consulente, persona con rapporto saltuario con la compagnia, etc. e quindi possiamo specializzare la classe Impiegato in modo che abbia comportamenti diversi, ad esempio, per persone che sono al 100% nella compagnia che magari hanno studio e telefonino assegnato, rispetto a dei consulenti che magari hanno meno benefits ed hanno una remunerazione minore; e così via. Allo stesso modo potremmo avere una classe musicista che si specializza in pianista, sassofonista, bassista, etc. ed un bassista ha necessità di avere una spartitura in chiave di basso mentre un sassofonista in chiave di si-bemolle. E si possono fare migliaia di esempi così, anche sugli strumenti musicali ad esempio.

Allora per poter rappresentare una gerarchia di oggetti in cui un oggetto è un caso specifico di un altro oggetto si usa il concetto di ereditarietà.

Un oggetto derivato eredita alcune informazioni e ne specializza altre. Per fare in modo che i vari oggetti implementino in maniera specifica delle operazioni si usa il concetto di polimorfismo. Il polimorfismo ci permette di programmare in maniera generica in modo tale da chiedere, per esempio, ad uno strumento di suonare una nota indipendentemente da che strumento si tratti; ciascuno strumento poi avrà il suo modo di suonare ciascuna nota. Oppure si chiede al sistema di pagamento di calcolare il salario di ciascun dipendente; in base al tipo di dipendente poi verrà calcolato il salario, ma il programma di calcolo sarà sempre lo stesso: basta "premere il bottone *calcola_il_salario*" su ciascun oggetto e ciascun oggetto di tipo impiegato avrà la sua specializzazione in modo da calcolare la retribuzione. La possibilità di avere un'impostazione complessiva del programma che non va cambiata in base alle specifiche di una certa funzione è uno dei vantaggi della programmazione OO e permette la realizzazione di un software robusto e su cui è più facile fare manutenzione. Richiediamo quindi dei comportamenti agli oggetti e poi in base al tipo di oggetto il comportamento viene implementato in maniera specifica.

Allora: caratteristica fondamentale dei linguaggi OO è quella del riutilizzo di codice esistente; questo viene effettuato con i 2 concetti di composizione ed ereditarietà. La composizione l'abbiamo vista; l'ereditarietà consiste nello specializzare una classe esistente con un codice specifico. Quindi si parte da una classe esistente e la si specializza, la si migliora. Questo evita di definire un nuovo con nuove variabili e

funzioni membro, facendo ereditare all'oggetto in questione i membri di una classe già esistente. Quindi si crea una gerarchia di oggetti in cui gli oggetti più in alto rappresentano concetti più astratti di cui quelli più in basso sono delle specializzazioni.



La classe esistente è detta **CLASSE BASE**; da questa si può derivare una classe che ne eredita i dati membro e le funzioni membro e che viene detta **CLASSE DERIVATA**.

Relazioni:

- **is-a**, ovvero “è un”, rappresenta *l’inheritance*, ovvero l’ereditarietà. il rapporto che intercorre tra **classe base e classe derivata**. La classe derivata eredita tutte le proprietà della **classe base**; quindi ad esempio una macchina è un veicolo e quindi ha tutti gli attributi ed i comportamenti di un veicolo. Come nell’esempio di sopra si vede che una classe derivata può fare a sua volta da classe base per un’altra classe derivata.
- **has-a**, ovvero “ha un”, rappresenta la *composition*, ovvero la composizione di oggetti.

Sintassi:

```
1 class BaseClass {  
2     // Definitions for BaseClass  
3     ...  
4 };  
5 class DerivedClass: public BaseClass {  
6     // Definitions for DerivedClass  
7     ...  
8 };  
9 -
```

Questa è la sintassi per indicare che una classe è derivata da un'altra.

Questa è l'ereditarietà *public*; è quella più comune ma è possibile anche quella protetta e privata.

La classe derivata eredita tutti i membri della classe base: quelli *public*, *private* e *protected* (poi vedremo cosa vuol dire); TRANNE i costruttori e gli operatori di assegnazione, che devono essere specifici per ogni classe derivata.

Le classi derivate non possono cancellare ciò che ereditano e non possono scegliere cosa ereditare. Tuttavia, e questo è il polimorfismo, le classi derivate possono riscrivere (*override*) il comportamento e quindi le funzioni membro. Questo permette di invocare una sola volta una funzione su diversi tipi di oggetti derivati da una classe base in modo tale che questa funzione venga riscritta per ogni oggetto derivato.

Nella classe base possiamo definire dei membri *public* e protetti, a cui possono accedere le classi derivate. Le classi derivate però non possono accedere ai membri privati della classe base; questi membri ci sono ma la classe derivata non può accedervi.

La parola chiave (modificatore di accesso) *protected* sta ad indicare che le regioni protette possono essere accedute solo attraverso le funzioni membro della classe derivata, esattamente come quelle private, però possono essere accedute dai membri definiti nella classe derivata, come quelle pubbliche. Quindi le funzioni friend e le classi friend non possono accedere a queste regioni ma possono solo le classi derivate. Quindi in sostanza “*protected*” serve a dare accesso alle variabili membro di una classe base solo a classi derivate da questa classe base e non anche a classi e funzioni friend.

LEZIONE 18

EREDITARIETÀ

La scorsa lezione abbiamo cominciato a parlare dell'ereditarietà, ovvero del poter scrivere una classe che è una specializzazione di una più generale. La classe derivata eredita le proprietà, in termini di funzioni membro e dati membro, della classe base attraverso una relazione *is-a*, nel senso che la classe derivata è un caso speciale della classe base, magari con ulteriori variabili membro e con ulteriori funzioni membro che le permetteranno di mettere a disposizione altri servizi ai propri client oltre a quelli messi a disposizione dalla classe base.

La sintassi è questa:

```
1 class BaseClass {  
2     // Definitions for BaseClass  
3     ...  
4 };  
5 class DerivedClass: public BaseClass {  
6     // Definitions for DerivedClass  
7     ...  
8 };  
9 -
```

Con l'operatore ":" diciamo che la classe derivata a sinistra dei 2 punti è un caso speciale (un miglioramento) della classe base alla destra.

La classe derivata eredita tutte le funzioni e le variabili membro della classe base, sia pubbliche che protette che private, ad eccezione di:

- costruttori;
- operatore di assegnazione.

La classe derivata inoltre può decidere di *override*, ovvero di riscrivere, alcuni tra i dati e le funzioni membro della classe che eredita. In particolare i membri della classe base definiti *public* o *protected* possono essere acceduti in maniera diretta dalla classe derivata. La differenza tra *public* e *protected* è che i dati *public* sono accessibili all'esterno della classe base, mentre i dati *protected* sono accessibili solo alle classi derivate. Invece i membri della classe base definiti come *private* non sono accessibili alle funzioni membro definite nella classe derivata.

Vediamo un esempio.

Class Account

```
1 class Account {
2 public:
3     //Constructors
4     Account() : balance{0.0} { }
5     Account(double initial) : balance{initial} { }
6
7     //Member functions
8     void credit(double amt) { balance += amt; }
9     void debit(double amt) { balance -= amt; }
10    double getBalance() const { return balance; }
11
12 private:
13     //Member data
14     double balance;
15 };
```

Questa è una classe che definisce un account (bancario), quindi ha una variabile membro *balance* di tipo *double* e modifikatore di accesso *private*. La classe definisce dei servizi pubblici, in particolare il deposito *credit* di una certa quantità, che non fa altro che aumentare il *balance* di una certa quantità. Ovviamente va accertato che *amt* \geq 0, perché altrimenti non sarebbe un deposito ma un prelievo. E poi c'è il prelievo *debit*, che diminuisce il *balance* di una certa quantità *amt* (amount); va ovviamente verificato anche in questo caso che *amt* non ecceda il saldo. C'è poi la funzione *getBalance* che restituisce il saldo attuale. Quindi il seguente, ad esempio, può essere un client che utilizza la classe *Account.h* definita:

```
1 #include <iostream>
2 #include "Account.h"
3
4 using std::cout; using std::endl;
5
6 int main() {
7     Account acct(1000.0);
8     acct.credit(1000.0);
9     acct.debit(100.0);
10    cout << "Balance is: " << acct.getBalance() << endl;
11
12    return 0;
13 }
```

```
$ g++ -c account_main1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o account_main1 account_main1.o
$ ./account_main1
Balance is: 1900
```

Account mette a disposizione 2 costruttori: uno che non ha argomenti e che inizializza il saldo a 0 ed uno che ha un argomento di tipo double e che è il saldo iniziale.

In generale però esistono più tipi di conti bancari. Tutti hanno sicuramente il saldo ma alcuni permettono di fare dei prelievi con la carta di credito/debito tramite gli sportelli automatici, altri sono del tipo “saving”, nel senso che sono di risparmio e quindi non permettono di avere una carta di credito per prelevare ma magari danno degli interessi.

Allora definiamo una classe derivata dalla classe **Account**, ovvero la classe **CheckingAccount** che eredita tutto (tranne i costruttori e l'eventuale operatore di assegnazione) da **Account** e quindi avrà un servizio Credit, uno Debit ed uno getBalance ed eredita ovviamente anche le variabili membro; mette però a disposizione anche altre funzioni e definisce eventualmente anche altre variabili membro. Vediamolo.

```
18 class CheckingAccount : public Account {
19     //CheckingAccount is derived from class Account
20
21 public:
22     //Constructor
23     CheckingAccount(double initial, double atm) : Account(initial),
24     totalFees{0.0}, atmFee{atm} { }
25
26     void cashWithdrawal(double amt) {
27         totalFees += atmFee;
28         debit(amt + atmFee); //call to the member function of the base class
29     }
30     //get function
31     double getTotalFees() const { return totalFees; }
32     std::string type() const { return "Checking Account"; }
33 private:
34     /*private data members
35     double totalFees;
36     double atmFee;
37 };
```

Ci sono 2 variabili membro definite nella classe derivata che sono le **commissioni per il prelievo (atmFee)** e le **spese totali (totalFees)** **addebitate** sul conto di tipo **CheckingAccount**. Ovviamente anche **balance**, variabile membro privata di **Account**, è presente in **CheckingAccount**, però le funzioni membro definite in quest'ultima non possono accedervi perché privata. I servizi definiti dalla classe derivata sono ad esempio il prelievo di contanti dal distributore automatico, una funzione che mostra le spese ed una che mostra in una stringa “Checking Account”. In particolare vediamo che alla riga 27 viene utilizzata la funzione membro **debit** della classe base all'interno della funzione **cashWithdrawal** definita nella classe derivata, e questo è interessante. Vediamo poi che nella classe derivata è definito un costruttore per l'inizializzazione

delle variabili membro definite proprio all'interno di questa classe. In realtà viene inizializzata anche la variabile membro *balance* della classe base, però questa inizializzazione avviene tramite l'invocazione esplicita del costruttore della classe base. Se non avessimo invocato il costruttore della classe base sarebbe stato invocato il costruttore di default che inizializza il saldo a 0 (Account ()).

Analogamente definiamo un altro tipo di account che è quello di deposito, *SavingsAccount*, che non prevede spese di tenuta conto o spese di operazioni di prelievo ma mette un unico servizio a disposizione che è il calcolo degli interessi (oltre a mettere a disposizione i servizi offerti dalla classe *Account*).

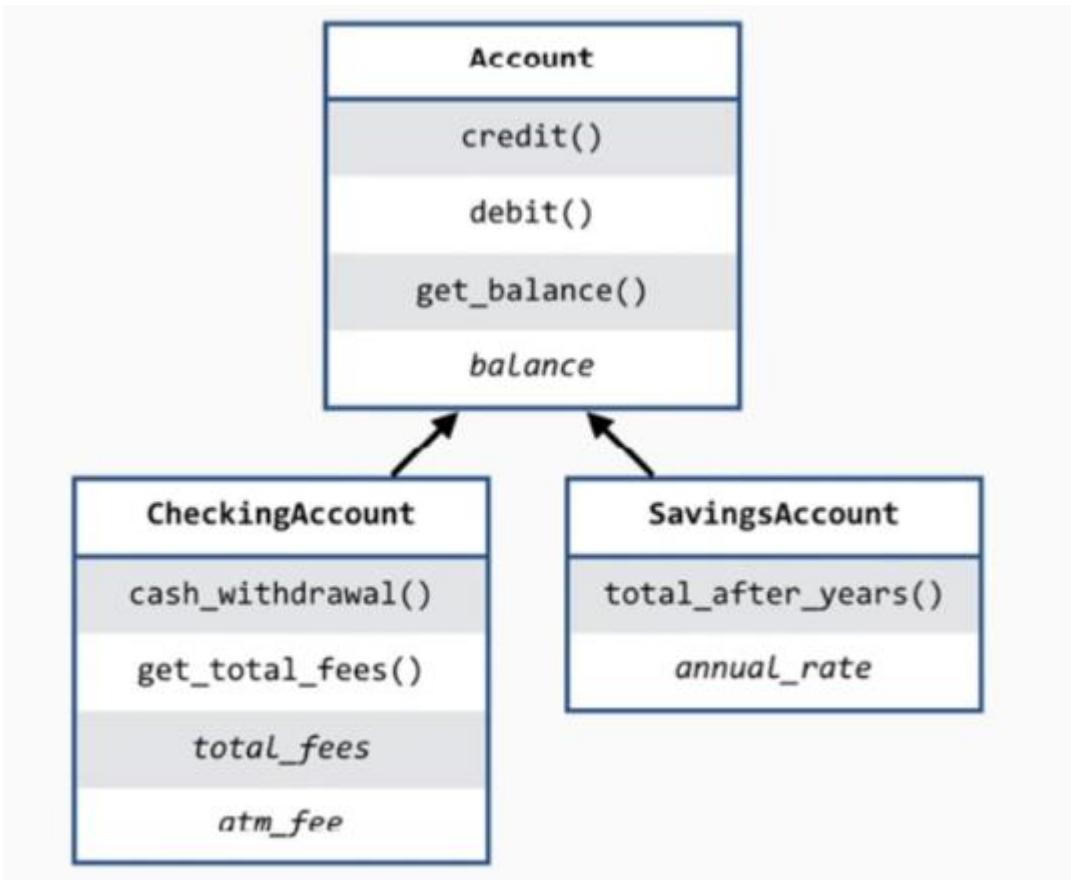
```
40 class SavingsAccount : public Account {  
41 //SavingsAccount is derived from class Account  
42  
43 public:  
44 //Constructor  
45 SavingsAccount(double initial, double rate) :  
46 Account(initial), annualRate{rate} {}  
47 // To be implemented here (exercise): compound interest calc  
48 double totalAfterYears(int years) const;  
49  
50 private:  
51 double annualRate;  
52 };  
--
```

Possiamo pensare di implementare noi la funzione *totalAfterYears* che prende in ingresso il numero di anni per cui si vuole calcolare la cifra che ci sarà sul conto dopo essi. Ovviamente questo richiede di modificare la variabile membro della classe base *balance* e va fatto agendo in maniera indiretta su di essa o eventualmente definendo la variabile *balance* come protetta all'interno della classe base.

Tabella riassuntiva sui modificatori di accesso:

Access modifier	Any function	Derived-class members	Same-class members
public	Yes	Yes	Yes
protected	No	Yes	Yes
private	No	No	Yes

Di seguito invece il diagramma UML del programma sugli account:



Se volessi complicare le cose potrei derivare da *SavingsAccount* delle classi che specializzino ancora di più l'account distinguendo tra conti di deposito vincolato (non permettono il prelievo prima di un numero di anni) e conti di deposito non vincolato.

Vediamo un client come può utilizzare questa gerarchia di classi:

```

1 //account_main2.cpp
2 //test inheritance
3 #include <iostream>
4 #include "Account2.h"
5
6 using std::cout; using std::endl;
7
8 int main() {
9     Account acct(1000.0);
10    acct.credit(1000.0);
11    acct.debit(100.0);
12    cout << "Account balance is: $" << acct.getBalance() << endl;
13
14    CheckingAccount checking(1000.0, 2.00);
15    checking.credit(1000.0); //calls the function of the base class
16    checking.cashWithdrawal(100.0); // incurs ATM fee
17    cout << "Checking balance is: $" << checking.getBalance() << endl;
18    cout << "Checking total fees is: $" << checking.getTotalFees() << endl;
19
20    SavingsAccount saving(1000.0, 0.05);
21    saving.credit(1000.0);
22    cout << "Savings balance is: $" << saving.getBalance() << endl;
23
24    return 0;
25 }
  
```

```

$ g++ -c account_main2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o account_main2 account_main2.o
$ ./account_main2
Account balance is: $1900
Checking balance is: $1898
Checking total fees is: $2
Savings balance is: $2000
  
```

L'oggetto *CheckingAccount* è inizializzato specificando sia il saldo iniziale che le commissioni per ciascun prelievo. Su questo oggetto possiamo i servizi offerti dalla classe base *Account*.

bene notare la differenza tra composizione di oggetti, che è di tipo *has-a* poiché c'è un oggetto che contiene un altro oggetto, ed ereditarietà, che è di tipo *is-a* poiché un oggetto derivato da una classe base fa parte di quella classe base nel senso che mette a disposizione gli stessi servizi + altri.

POLIMORFISMO

Potrebbe capitare invece che la classe derivata metta a disposizione lo stesso servizio della classe base ma con delle modalità diverse, nel senso che una classe derivata potrebbe riscrivere dei comportamenti della classe base. E quindi allora la classe derivata può implementare un certo servizio in una maniera piuttosto che in un'altra riscrivendolo rispetto a quello presente nella classe base (override). Questo è reso possibile grazie al polimorfismo, che ci permette di programmare in modo generale facendo in modo che gli oggetti si scambino tra di loro dei messaggi per attivare dei servizi ed in base al tipo di oggetto su cui è stato invocato un servizio, quel servizio assume una forma piuttosto che un'altra; da cui "polimorfismo". Se ad esempio ho la classe "impiegati", posso calcolare gli stipendi usando una stessa funzione "retribuzione" comune a tutte le classi derivate che però assume significati diversi in base alla specifica classe derivata di impiegato (consulente, etc.). Questo ci semplifica la vita poiché se un oggetto evolve in nuove tipologie di oggetti (ad esempio creiamo altri tipi di impiegati), le nuove tipologie di oggetti non devono fare altro che riscrivere lo stesso servizio messo a disposizione dagli altri oggetti.

Altro esempio: potremmo scrivere una classe "Poligono" da cui deriviamo "Triangolo", "Rettangolo", etc. Sfruttando il polimorfismo possiamo allora invocare gli stessi servizi: *.area()* e *.perimetro()* su ciascuna classe derivata per calcolare area e perimetro degli specifici poligoni.

Nel caso dell'account bancario possiamo pensare invece di avere una stessa funzione *type()* che specifica il tipo di account bancario tramite una stringa e che però assume significati diversi in base a su che tipo di oggetto derivato è invocata.

```
1 class Account {
2 public:
3 ...
4     std::string type() const { return "Account"; }
5 ...
6 };
7
8
9 class CheckingAccount : public Account {
10 public:
11 ...
12     std::string type() const { return "CheckingAccount"; }
13 ...
14 };
15
16
17 class SavingsAccount : public Account {
18 public:
19 ...
20     std::string type() const { return "SavingsAccount"; }
21 ...
22 };
```

Quello che ci può interessare però è cosa succede se proviamo ad usare un certo servizio non sapendo a priori su che tipo di oggetto lo stiamo invocando. Vediamolo con un esempio.

```
1 // Inheritance & upcasting
2
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 enum note { middleC, Csharp, Eflat }; // Etc.
7
8 class Instrument {
9 public:
10    void play(note) const {
11        cout << "Instrument::play" << endl;
12    }
13 };
14
15 // Wind objects are Instruments
16 // because they have the same interface:
17 class Wind : public Instrument {
18 public:
19    // Redefine interface function:
20    void play(note) const {
21        cout << "Wind::play" << endl;
22    }
23 };
24
25 void tune(Instrument& i) {
26     //this fulction will
27     //play a note for an give instrument
28     i.play(middleC);
29 }
30
31 int main() {
32     Wind flute;
33     tune(flute);
34 }
```

```
$ g++ Instrument_fail.cpp -o Instrument_fail
$ ./Instrument_fail
Instrument::play
```

In questo esempio abbiamo una classe *Instrument* che ha una funzione *play* che restituisce la nota suonata da un generico strumento. Possiamo definire delle classi derivate come *Wind* (strumenti a fiato) che riscrivono la funzione membro *play*.

La funzione *tune* prende in ingresso un riferimento ad un oggetto di tipo *Instrument*; nel main però la invochiamo su *flute* che è un oggetto di tipo *Wind*. Come è possibile? Semplicemente un oggetto di tipo *Wind* è anche un oggetto di tipo *Instrument*. In realtà però questo programma fallisce, tanto è vero che dà in output il messaggio della classe *Instrument* e non della classe derivata *Wind*. Per capire il perché di ciò dobbiamo capire **cosa è il BINDING**. Cerchiamo però prima di capire perché è sintatticamente corretto l'esempio. Abbiamo trattato un oggetto di tipo *Wind* come uno di tipo *Instrument*: questo si chiama **UPCASTING**, ovvero stiamo trattando un oggetto di una classe derivata come se fosse un oggetto di una classe base. “Upcasting” perché stiamo salendo in alto nella gerarchia delle classi. Il salire nella gerarchia implica però necessariamente una restrizione delle funzioni messe a disposizione dalla classe base

rispetto a quelle messe a disposizione da una derivata; è un processo di “**restrizione dell’interfaccia**”. L’errore nell’esempio è dovuto allora al fatto che quando definiamo le funzioni della classe base dobbiamo specificare quali possono essere riscritte dalle classi derivate. Il **binding** è la connessione tra invocazione di una funzione ed il corpo della funzione; è quello avviene con il JSR quando si lega l’indirizzo della prima istruzione della funzione chiamata a quello dell’istruzione corrente della funzione chiamante. Se è effettuato prima dell’esecuzione del programma è detto **early binding**, quando è eseguito dopo è detto **late binding**. Nel nostro esempio binding vuol dire capire quale funzione chiamare nella riga 28. Tuttavia noi a livello di compilazione non sappiamo su che tipo di oggetto viene invocata la funzione *tune*, tra i vari oggetti delle classi derivate. Potrebbe essere invocata su un oggetto “percussione”, su un oggetto “legno”, etc. Per questo il binding in questione va effettuato a tempo di esecuzione. Allora per questo il polimorfismo è utile, perché mi permette di non sapere su che oggetto sarà invocata una certa funzione. C’è però bisogno di un meccanismo che implementi il **dynamic binding**, cioè il legame tra la chiamata e lo specifico indirizzo della funzione invocata a livello di *runtime* e non di compilazione (quello che abbiamo chiamato anche “late binding”). La soluzione a questo problema, comune a tutti i linguaggi compilativi quale è il C++, viene effettuata mediante la definizione di funzioni dette **virtual**: le funzioni dichiarate come “virtual” vengono considerate dal compilatore come funzioni per cui va effettuato il *late (dynamic) binding*.

IMPORTANTE: Il *late binding* è possibile solo con le funzioni dichiarate come virtuali all’interno della classe base e solo quando si sta utilizzando l’indirizzo della classe base in cui questa funzione virtuale esiste. Quindi solo quando ho a disposizione un riferimento dell’oggetto della classe base sul quale invoco la funzione virtual posso usare il *late binding*. Riassumendo, le condizioni necessarie per il *late binding* sono 2: uso di funzioni virtual e passaggio per riferimento dell’oggetto della classe base alla funzione virtual. Per dichiarare una funzione come *virtual* basta antecedere la parola “*virtual*”.

L’esempio di prima con una funzione virtual diventa:



In realtà solo la dichiarazione (file.h) necessita della parola chiave “*virtual*”, non la definizione (file.cpp).

```

1 // Instrument_ok.cpp
2 // Late binding with virtual
3
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 enum note { middleC, Csharp, Eflat }; // Etc.
8
9 class Instrument {
10 public:
11     virtual void play(note) const {
12         cout << "Instrument::play" << endl;
13     }
14 };
15
16 // Wind objects are Instruments
17 // because they have the same interface:
18 class Wind : public Instrument {
19 public:
20     // Redefine interface function:
21     void play(note) const {
22         cout << "Wind::play" << endl;
23     }
24 };
25
26 void tune(Instrument& i) {
27     //this fulction will
28     //play a note for an give instrument
29     i.play(middleC);
30 }
31
32 int main() {
33     Wind flute;
34     tune(flute);
35 }
```

\$ g++ Instrument_ok.cpp -o Instrument_ok -std=c++11
\$./Instrument_ok
Wind::play

La ridefinizione di una virtual function in una classe derivata è detta riscrittura (overriding). Nel nostro esempio alla riga 21 avviene l'overriding della funzione *play* per la classe derivata *Wind*. La funzione virtuale va scritta come tale, ovvero come *virtual*, solo nel file.h e solo nella classe base; poi le funzioni delle classi derivate che matchano con quella della classe base dichiarata come *virtual* saranno interpretate proprio come virtuali e quindi saranno oggetto del *late binding* e questo proprio in virtù del polimorfismo. Quindi, in conclusione, quando ci si riferisce ad un oggetto della classe derivata utilizzando un puntatore (o meglio riferimento) ad un oggetto della classe base, allora si può invocare una funzione virtuale per quell'oggetto ed interpretare correttamente la funzione sul tipo di oggetto desiderato. Se non passiamo un riferimento allora l'oggetto della classe derivata viene copiato come oggetto della classe base e si ha quello che si dice “object slicing”, ovvero l'oggetto viene “affettato” nel senso che viene copiata solo la parte che è propria anche della classe base e quindi la funzione non viene invocata correttamente. Ergo, c'è o puntatore afforz.

Vediamo un altro esempio (senza implementazione della classe).

Instrument.h

```

1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 enum note { middleC, Csharp, Cflat }; // Etc.
5
6 class Instrument {
7 public:
8     virtual void play(note) const {
9         cout << "Instrument::play" << endl;
10    }
11    std::string what() const {
12        return "Instrument";
13    }
14    // Assume this will modify the object:
15    virtual void adjust(int) {}
16 };
17
18 class Wind : public Instrument {
19 public:
20     void play(note) const {
21         cout << "Wind::play" << endl;
22     }
23     std::string what() const { return "Wind"; }
24     void adjust(int) {}
25 };
26
27 class Percussion : public Instrument {
28 public:
29     void play(note) const {
30         cout << "Percussion::play" << endl;
31     }
32     std::string what() const { return "Percussion"; }
33     void adjust(int) {}
34 };
35
36 class Stringed : public Instrument {
37 public:
38     void play(note) const {
39         cout << "Stringed::play" << endl;
40     }
41     std::string what() const { return "Stringed"; }
42     void adjust(int) {}
43 };
44
45 class Brass : public Wind {
46 public:
47     void play(note) const {
48         cout << "Brass::play" << endl;
49     }
50     std::string what() const { return "Brass"; }
51 };
52
53 class Woodwind : public Wind {
54 public:
55     void play(note) const {
56         cout << "Woodwind::play" << endl;
57     }
58     std::string what() const { return "Woodwind"; }
59 };

```

testInstrument.cpp

```
1 //testInstrument.cpp
2 #include <vector>
3 #include <iostream>
4 #include "Instrument.h"
5 using std::cout;
6 using std::endl;
7
8 using std::vector;
9 // Identical function from before:
10 void tune(Instrument& i) {
11     // ...
12     i.play(middleC);
13 }
14
15
16 // Upcasting during array initialization:
17
18 int main() {
19     vector<Instrument*> Band;
20     Wind flute;
21     Percussion drum;
22     Stringed violin;
23     Brass flugelhorn;
24     Woodwind alto;
25     Woodwind tenor;
26     cout << "---- Rehearsal ----" << endl;
27
28     tune(flute);
29     tune(drum);
30     tune(violin);
31     tune(flugelhorn);
32     tune(alto);
33     tune(tenor);
34     cout << "---- Let's gig together ----" << endl;
35
36     Band.push_back(&flute);
37     Band.push_back(&drum);
38     Band.push_back(&violin);
39     Band.push_back(&flugelhorn);
40     Band.push_back(&alto);
41     Band.push_back(&tenor);
42     for(Instrument* i : Band) {
43         tune(*i);
44     }
45 }
```

Alla riga 19 del driver c'è una cosa importante: definiamo un vettore di puntatori ad *Instrument* e di nome *Band*. Questo lo facciamo perché vogliamo mandare lo stesso messaggio a tutti gli strumenti, ovvero quello di suonare una certa nota; poi sarà ogni strumento ad implementare a modo suo quella nota. Quindi rappresentiamo un insieme di oggetti derivati utilizzando dei puntatori agli oggetti della classe base. Se *Band* fosse un vettore di *Instrument* e non di *Instrument** succederebbe che quando facciamo un `push_back`, ovvero inseriamo un oggetto nel vettore, viene copiato quell'oggetto all'interno del vettore e quindi avviene lo “slicing”: vengono copiati solo i valori di *Instrument* contenuti all'interno di ciascun oggetto. Succederebbe questo:



```
37 ---- Rehearsal ----
38 Wind::play
39 Percussion::play
40 Stringed::play
41 Brass::play
42 Woodwind::play
43 Woodwind::play
44 ---- Let's gig together ----
45 Instrument::play
46 Instrument::play
47 Instrument::play
48 Instrument::play
49 Instrument::play
50 Instrument::play
51 bash-3.2$
```

Talvolta si usa una *pure virtual class*, nel senso che quando una classe base è troppo generica la si usa solo come interfaccia, ovvero come elenco delle funzioni, ed alle classi derivate si delega il compito di implementare le funzioni. In tal caso la classe viene dichiarata normalmente ma le funzioni sono seguite da “= 0”. Questa è la sintassi per definire funzioni virtuali. La classe viene detta anche classe “astratta”. Non si possono istanziare oggetti di classi astratte ma si possono istanziare oggetti di classi derivate dalle classi astratte.

Es:

```
1 class Shape {
2 public:
3     virtual double area() const { } // NO SENSIBLE IMPLEMENTATION FOR THIS
4 };
5
6
7 class Rectangle : public Shape {
8 public:
9     ...
10    virtual double area() const { return width * height; }
11    ...
12 private:
13     double width, height;
14 };
15
16 class Circle : public Shape {
17 public:
18     ...
19     virtual double area() const { return PI * radius * radius; }
20     ...
21 private:
22     double radius;
23
24 };
25
```

Non siamo capaci di calcolare l’area di un poligono generico. Allora la funzione area funge solo da interfaccia e quindi mostra il servizio “area”; ma l’implementazione della funzione che calcola l’area cambia poi da poligono a poligono.

LEZIONE 19

Abbiamo visto che le classi derivate ereditano le variabili membro e le funzioni membro dalla classe base. La particolarità del polimorfismo rispetto ad esempio alla composition è che possiamo specializzare delle classi così da fare dei compiti in maniera specifica per ogni classe derivata. Un programmatore invoca un messaggio su ogni oggetto e ciascun oggetto interpreterà quel messaggio in base a come è implementato quel servizio all'interno di quell'oggetto. Quando una funzione client invoca una virtual function (member) che è stata riscritta (overridden) passandole un riferimento all'oggetto su cui si vuole invocare la funzione allora si può realizzare effettivamente il polimorfismo ed è reso possibile il dynamic binding che fa sì che il client non debba conoscere a priori su quale tipo di oggetto invocare la funzione poiché il binding è fatto a tempo di esecuzione. C'è bisogno di passare il riferimento perché tutto ciò che non è passato per riferimento è copiato tramite una "shallow copy", ovvero tramite una copia membro a membro degli oggetti.

Ci sono volte in cui la classe base è talmente generica che dichiara funzioni ma non le implementa perché lascia l'implementazione alle classi derivate; in tal caso la classe è detta **classe astratta** ed utilizza le **pure virtual functions**, riconoscibili perché virtual e concluse con "`= 0`". Es:

```
virtual double area() const = 0; //pure virtual
```

Supponiamo di aver scritto una classe base ed una derivata; entrambe allocano memoria e quindi noi dobbiamo saper gestire la memoria richiesta al SO restituendola quando gli oggetti che stiamo utilizzando non ci servono più. Vediamo l'esempio:

```
1 //vиртдтор.h
2 class Base {
3 public:
4
5     Base() : base_memory(new char[1000]) {}
6     ~Base() { delete[] base_memory; }
7
8 private:
9     char *base_memory;
10 };
11
12
13 class Derived : public Base {
14 public:
15     Derived() : Base(), derived_memory(new char[1000]) {}
16     ~Derived() { delete[] derived_memory; }
17 private:
18     char *derived_memory;
19 };

1 //виртдтор.cpp
2 #include "virtdtor.h"
3 int main() {
4
5     // Note use of base-class pointer
6     Base *obj = new Derived();
7     delete obj; // calls what destructor(s)?
8     return 0;
9 }
```

Abbiamo una classe base (Base) ed una derivata (Derived) che ne è una specializzazione. La classe derivata ha ovviamente un costruttore che non fa altro che invocare il costruttore della classe base ed allocare memoria per un built-in array (1000 byte) che è una variabile privata. La classe base ha anch'essa un costruttore base che alloca altri 1000 byte; quindi sia Base class che Derived class allocano memoria, ed ogni volta che istanzieremo un oggetto della classe derivata sarà allocata memoria sia dalla classe derivata che da quella base. Dobbiamo allora sapere come rilasciare la memoria al SO. Concentriamoci sulla riga 6 del .cpp (driver).

```
6 Base *obj = new Derived();
```

Questa riga dichiara un puntatore ad un oggetto di classe Base e lo istanzia con un puntatore ad un oggetto di classe Derived. Quindi il puntatore è dichiarato come puntatore alla classe base ma è istanziato poi come puntatore alla classe derivata. Allora viene invocato il costruttore della classe Derived che invoca poi quello della classe Base che richiede 1000 byte di memoria prima ed altri 1000 poi. Quindi stiamo allocando memoria sia a livello di classe base che a livello di classe derivata. Se invochiamo il distruttore per l'oggetto istanziato viene invocato il distruttore della variabile puntata da *obj*, che è un oggetto della classe base. Il problema è che non c'è il binding dinamico perché il distruttore non è dichiarato come *virtual* → c'è bisogno del **virtual destructor**. Infatti senza distruttore virtual quello che succede è che il distruttore invocato è quello della classe base e non quello della classe derivata, quindi il programma non rilascia del tutto la memoria richiesta al SO ma solo quella richiesta al costruttore della classe base. Alla fine il distruttore è una funzione membro particolare e quindi per poter effettuare il polimorfismo c'è bisogno che anche il distruttore venga dichiarato come *virtual*, affinché venga invocato il distruttore della classe derivata e non quello della classe base.

Allora vediamo il programma corretto:



Dichiarando il distruttore della classe base e quello della classe derivata come *virtual* alla distruzione dell'oggetto derivato viene chiamato il distruttore dell'oggetto derivato della classe base prima e, automaticamente, quello dell'oggetto della classe base poi.

La regola allora è che **ciascuna classe con una member function virtuale deve sempre definire un distruttore virtuale**.

Ora vediamo un esercizio.

```
1 //vиртдтор2.h
2 class Base {
3 public:
4
5     Base() : base_memory{new char[1000]} { }
6     //now virtual
7     virtual ~Base() { delete[] base_memory; }
8
9 private:
10    char *base_memory;
11 };
12
13 class Derived : public Base {
14 public:
15     Derived() : Base(), derived_memory {new char[1000]} { }
16     //now virtual
17     virtual ~Derived() { delete[] derived_memory; }
18
19 private:
20    char *derived_memory;
21 };
22
23 //}
24 //виртдтор2.cpp
25 #include "virtdtor2.h"
26 int main() {
27
28     // Note use of base-class pointer
29     Base *obj = new Derived();
30     delete obj; // calls what destructor(s)?
31     return 0;
32 }
```

Employee class

Case study (Deitel)

- A company pays its employees weekly. The employees are of three types:

- Salaried employees are paid a fixed weekly salary regardless of the number of hours worked,
- commission employees are paid a percentage of their sales
- base-salary-plus-commission employees receive a base salary plus a percentage of their sales.

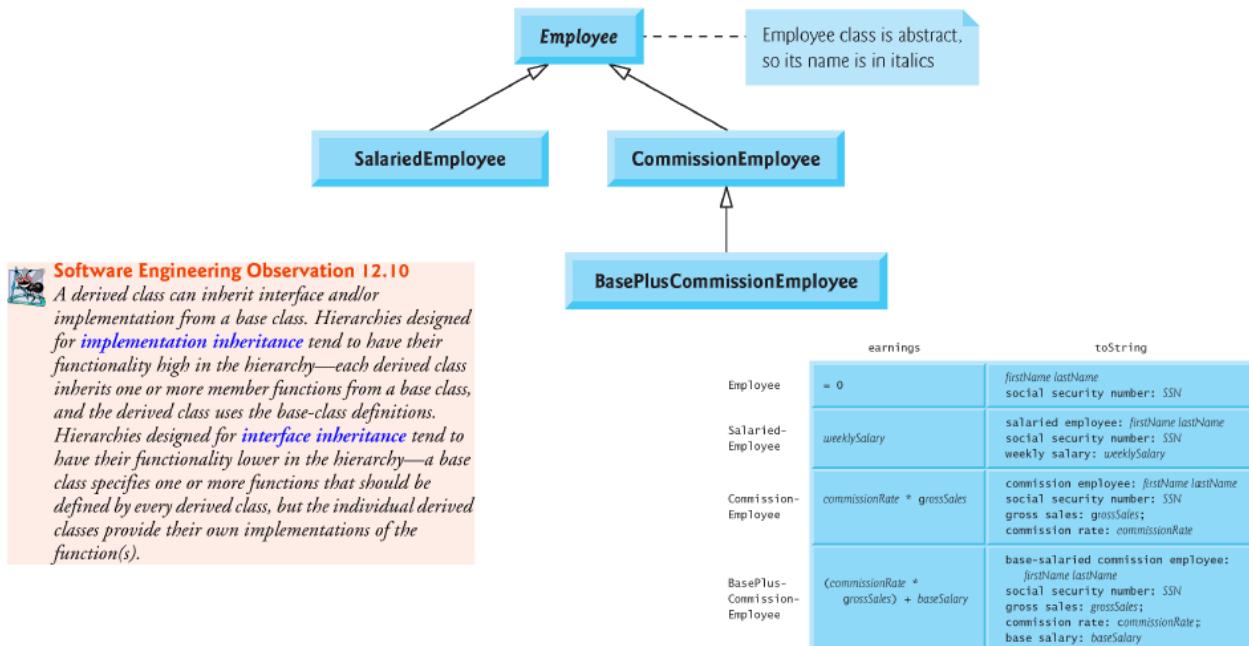
For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries. The company wants to implement a C++ program that performs its payroll calculations polymorphically.

C'è un'azienda che deve calcolare lo stipendio di ciascun impiegato. C'è quindi la classe "Impiegato", ma ovviamente ci sono vari tipi di impiegati:

- i salariati, con stipendio fisso settimanale;
- i venditori, pagati in percentuale alla quantità di vendite effettuate;
- impiegati "anziani" con un salario base + una percentuale sulle loro vendite.

Ovviamente ogni impiegato dovrà avere un nome, un cognome, una data di nascita, un indirizzo, un numero di telefono e un codice fiscale (SSN in America, ovvero Social Security Number). Quando dobbiamo calcolare lo stipendio dobbiamo distinguere ovviamente i tipi di impiegati. Magari in futuro la compagnia avrà un'altra categoria di impiegati come i consulenti, pagati in base al numero di ore di lavoro, e il programma può essere esteso. Quindi il programma client che calcola gli stipendi ha a disposizione una serie di tipi di impiegati e per ogni impiegato invoca il messaggio "calcola lo stipendio per quell'impiegato"; poi in base al tipo di impiegato il calcolo viene effettuato in maniera differente. Se appunto poi in futuro si vuole aggiungere un'altra categoria di impiegati il programma client non cambierà: è questo uno dei vantaggi principali della programmazione OO. Nel nostro esempio però ci sono situazioni in cui il client deve sapere su che tipo di oggetto sta invocando la funzione di calcolo. Infatti c'è la complicazione che la compagnia ha deciso per il periodo corrente di dare un premio soltanto alla terza tipologia di impiegati; quindi si dovrà controllare se l'impiegato è un commerciale "senior". Quindi dobbiamo implementare un programma C++ polimorficamente e quello che importa quindi è che ogni oggetto diverso di tipo Impiegato sappia calcolare il proprio stipendio.

Allora possiamo pensare di realizzare una gerarchia:



Ci sarà l'Impiegato con tutte le informazioni (nome, cognome, data di nascita e resto appresso detto prima); poi andrà distinto l'impiegato salariato da quello retribuito in percentuale alle vendite. Queste 2 classi derivate erediteranno tutte le informazioni di nome, cognome, etc. e poi dovranno implementare le funzioni di calcolo dello stipendio. Tuttavia dai commerciali si può derivare la terza tipologia di lavoratori, quella retribuita in parte in percentuale alle vendite ed in parte con un salario fisso settimanale. *Employee* sarà per noi una classe astratta perché non possiamo calcolare lo stipendio fin quando non so a quale classe afferisce l'impiegato specifico. Vediamo il programma.

```

1 // Employee.h
2 // Employee abstract base class.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 #include <string> // C++ standard string class
7
8 class Employee {
9 public:
10     Employee(const std::string&, const std::string&, const std::string &);
11     virtual ~Employee() = default; // compiler generates virtual destructor
12
13     void setFirstName(const std::string&); // set first name
14     std::string getFirstName() const; // return first name
15
16     void setLastName(const std::string&); // set last name
17     std::string getLastName() const; // return last name
18
19     void setSocialSecurityNumber(const std::string&); // set SSN
20     std::string getSocialSecurityNumber() const; // return SSN
21
22     // pure virtual function makes Employee an abstract base class
23     virtual double earnings() const = 0; // pure virtual
24     virtual std::string toString() const; // virtual
25 private:
26     std::string firstName;
27     std::string lastName;
28     std::string socialSecurityNumber;
29 };
30
31 #endif // EMPLOYEE_H
32

```

Alla riga 11 non facciamo altro che chiedere al compilatore di generare il distruttore di default. Il distruttore lo abbiamo definito come virtual perché la classe contiene delle funzioni virtuali. In particolare la funzione che calcola lo stipendio la dichiariamo *pure virtual*, perché non sappiamo come calcolare lo stipendio a questo livello generico e quindi ne mettiamo solo il prototipo. La funzione *toString()* invece vuole mostrare le informazioni in possesso per un impiegato ed ovviamente andrà riscritta per poter contenere informazioni anche sugli stipendi dei tipi di impiegati specifici. Tutte le funzioni elencate nel .h sono implementate così:

```
1 // Employee.cpp
2 // Abstract-base-class Employee member-function definitions.
3 // Note: No definitions are given for pure virtual functions.
4 #include <iostream>
5 #include "Employee.h" // Employee class definition
6 using std::string;
7
8 // constructor
9 Employee::Employee(const string& first, const string& last,
10                  const string& ssn)
11 : firstName(first), lastName(last), socialSecurityNumber(ssn) {}
12
13 // set first name
14 void Employee::setFirstName(const string& first) { firstName = first; }
15
16 // return first name
17 string Employee::getFirstName() const { return firstName; }
18
19 // set last name
20 void Employee::setLastName(const string& last) { lastName = last; }
21
22 // return last name
23 string Employee::getLastName() const { return lastName; }
24
25 // set social security number
26 void Employee::setSocialSecurityNumber(const string& ssn) {
27     socialSecurityNumber = ssn; // should validate
28 }
29
30 // return social security number
31 string Employee::getSocialSecurityNumber() const {
32     return socialSecurityNumber;
33 }
34
35 // toString Employee's information (virtual, but not pure virtual)
36 string Employee::toString() const {
37     return getFirstName() + " " + getLastname() +
38           "\nsocial security number: " + getSocialSecurityNumber();
39 }
40
```

Il distruttore è quello di default e quindi non lo implementiamo. Implementiamo il costruttore e tutti gli altri servizi della classe, tranne il calcolo degli stipendi che resta un prototipo.

```
1 // SalariedEmployee.h
2 // SalariedEmployee class derived from Employee.
3 #ifndef SALARIED_H
4 #define SALARIED_H
5
6 #include <string> // C++ standard string class
7 #include "Employee.h" // Employee class definition
8
9 class SalariedEmployee : public Employee {
10 public:
11     SalariedEmployee(const std::string&, const std::string&,
12                      const std::string&, double = 0.0);
13     virtual ~SalariedEmployee() = default; // virtual destructor
14
15     void setWeeklySalary(double); // set weekly salary
16     double getWeeklySalary() const; // return weekly salary
17
18     // keyword virtual signals intent to override
19     virtual double earnings() const override; // calculate earnings
20     virtual std::string toString() const override; // string representation
21 private:
22     double weeklySalary; // salary per week
23 };
24
25 #endif // SALARIED_H
26
27
```

La classe dell'impiegato salariato viene derivata dalla classe dell'impiegato e quindi ne eredita tutto tranne il costruttore. Viene riscritto il costruttore, che inizializza anche la variabile `weeklySalary` definita in questa classe derivata; ovviamente nel file .cpp sarà implementato il costruttore. Ancora una volta viene invocato il distruttore virtual di default (virtual perché ci sono delle funzioni virtuali nella classe). Vengono scritte le funzioni `set` e `get` per ottenere le informazioni sul salario che poi viene calcolato nella funzione riscritta `earnings()`. Anche la funzione `toString()` viene riscritta.

Vediamo il file .cpp. (pag. dopo)

```

1 // SalariedEmployee.cpp
2 // SalariedEmployee class member-function definitions.
3 #include <iomanip>
4 #include <stdexcept>
5 #include <sstream>
6 #include "SalariedEmployee.h" // SalariedEmployee class definition
7 //using namespace std;
8
9 // constructor
10 SalariedEmployee::SalariedEmployee(const std::string& first,
11                                     const std::string& last, const std::string&\n
12                                     ssn, double salary)           I
13                                     : Employee(first, last, ssn) {
14                                     setWeeklySalary(salary);
15 }
16
17 // set salary
18 void SalariedEmployee::setWeeklySalary(double salary) {
19     if (salary < 0.0) {
20         throw std::invalid_argument("Weekly salary must be >= 0.0");
21     }
22     weeklySalary = salary;
23 }
24
25 // return salary
26 double SalariedEmployee::getWeeklySalary() const { return weeklySalary; }
27
28 // calculate earnings;
29 // override pure virtual function earnings in Employee
30 double SalariedEmployee::earnings() const { return getWeeklySalary(); }
31
32 // return a string representation of SalariedEmployee's information
33 std::string SalariedEmployee::toString() const {
34     std::ostringstream output;
35     output << std::fixed << std::setprecision(2);
36     output << "salaried employee: "
37     << Employee::toString() // reuse abstract base-class function
38     << "\nweekly salary: " << getWeeklySalary();
39     return output.str();
}

```

Viene incluso ovviamente l'header corrispondente. Il costruttore è scritto sfruttando il costruttore della classe base *Employee* per le 3 variabili comuni a quella classe, ovvero nome, cognome e ssn, ed il quarto argomento viene inizializzato nel corpo del costruttore sfruttando la funzione *setWeeklySalary* (double) che fa anche validazione dell'argomento. C'è poi tra le varie cose la riscrittura di *earnings* che non fa altro che restituire il valore della variabile *salary* tramite *getWeeklySalary* (). La funzione *toString()*, riscritta anch'essa, stampa in *output* prima ciò che viene stampato dalla classe base, e quindi *Employee::toString()*, e poi le informazioni sul salario, che solo la classe derivata può dare. Similmente avviene per tutte le altre classi.

```

1 // CommissionEmployee.h
2 // CommissionEmployee class derived from Employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // C++ standard string class
7 #include "Employee.h" // Employee class definition
8
9 class CommissionEmployee : public Employee {
10 public:
11     CommissionEmployee(const std::string&, const std::string&,
12                         const std::string&, double = 0.0, double = 0.0);
13     virtual ~CommissionEmployee() = default; // virtual destructor
14
15     void setCommissionRate(double); // set commission rate
16     double getCommissionRate() const; // return commission rate
17
18     void setGrossSales(double); // set gross sales amount
19     double getGrossSales() const; // return gross sales amount
20
21     // keyword virtual signals intent to override
22     virtual double earnings() const override; // calculate earnings
23     virtual std::string toString() const override; // string representation
24 private:
25     double grossSales; // gross weekly sales
26     double commissionRate; // commission percentage
27 };
28
29 #endif // COMMISSION_H
30
31 // CommissionEmployee.cpp
32 // CommissionEmployee class member-function definitions.
33 #include <iomanip>
34 #include <stdexcept>
35 #include <sstream>
36 #include "CommissionEmployee.h" // CommissionEmployee class definition
37 using std::string;
38 using std::ostringstream;
39 using std::fixed;
40 using std::setprecision;
41 // constructor
42 CommissionEmployee::CommissionEmployee(const string &first,
43                                         const string &last, const string &ssn, double sales, double rate)
44     : Employee(first, last, ssn) {
45     setGrossSales(sales);
46     setCommissionRate(rate);
47 }
48
49 // set gross sales amount
50 void CommissionEmployee::setGrossSales(double sales) {
51     if (sales < 0.0) {
52         throw std::invalid_argument("Gross sales must be >= 0.0");
53     }
54     grossSales = sales;
55 }
56
57 // return gross sales amount
58 double CommissionEmployee::getGrossSales() const { return grossSales; }
59
60 // set commission rate
61 void CommissionEmployee::setCommissionRate(double rate) {
62     if (rate <= 0.0 || rate > 1.0) {
63         throw std::invalid_argument("Commission rate must be > 0.0 and < 1.0");
64     }
65     commissionRate = rate;
66 }
67
68 // return commission rate
69 double CommissionEmployee::getCommissionRate() const {
70     return commissionRate;
71 }
72
73 // calculate earnings; override pure virtual function earnings in Employee
74 double CommissionEmployee::earnings() const {
75     return getCommissionRate() * getGrossSales();
76 }
77
78 // return a string representation of CommissionEmployee's information
79 string CommissionEmployee::toString() const {
80
81     ostringstream output;
82     output << fixed << setprecision(2);
83     output << "commission employee: " << Employee::toString()
84     << "\ngross sales: " << getGrossSales()
85     << "; commission rate: " << getCommissionRate();
86     return output.str();
87 }
88
89 
```

```
1 // BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class derived from CommissionEmployee.
3 #ifndef BASEPLUS_H
4 #define BASEPLUS_H
5
6 #include <string> // C++ standard string class
7 #include "CommissionEmployee.h" // CommissionEmployee class definition
8
9 class BasePlusCommissionEmployee : public CommissionEmployee {
10 public:
11     BasePlusCommissionEmployee(const std::string&, const std::string&,
12         const std::string&, double = 0.0, double = 0.0, double = 0.0);
13     virtual ~BasePlusCommissionEmployee() = default; // virtual destructor
14
15     void setBaseSalary(double); // set base salary
16     double getBaseSalary() const; // return base salary
17
18     // keyword virtual signals intent to override
19     virtual double earnings() const override; // calculate earnings
20     virtual std::string toString() const override; // string representation
21 private:
22     double baseSalary; // base salary per week
23 };
24
25 #endif // BASEPLUS_H
26
```

```
1 // BasePlusCommissionEmployee.cpp
2 // BasePlusCommissionEmployee member-function definitions.
3 #include <iomanip>
4 #include <stdexcept>
5 #include <sstream>
6 #include "BasePlusCommissionEmployee.h"
7 using std::string;
8 using std::ostringstream;
9 using std::fixed;
10 using std::setprecision;
11 // constructor
12 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
13     const string& first, const string& last, const string& ssn,
14     double sales, double rate, double salary)
15 : CommissionEmployee(first, last, ssn, sales, rate) {
16     setBaseSalary(salary); // validate and store base salary
17 }
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary(double salary) {
21     if (salary < 0.0) {
22         throw std::invalid_argument("Salary must be >= 0.0");
23     }
24
25     baseSalary = salary;
26 }
27
28 // return base salary
29 double BasePlusCommissionEmployee::getBaseSalary() const {
30     return baseSalary;
31 }
32
33 // calculate earnings;
34 // override virtual function earnings in CommissionEmployee
35 double BasePlusCommissionEmployee::earnings() const {
36     return getBaseSalary() + CommissionEmployee::earnings();
37 }
38
39 // return a string representation of a BasePlusCommissionEmployee
40 string BasePlusCommissionEmployee::toString() const {
41     ostringstream output;
42     output << fixed << setprecision(2);
43     output << "base-salaried " << CommissionEmployee::toString()
44     << "; base salary: " << getBaseSalary();
45     return output.str();
46 }
47
48
```

Ora vediamo com'è fatto il main.

```
1 // testEmployee.cpp
2 // Processing Employee derived-class objects with static binding
3 // then polymorphically using dynamic binding.
4 #include <iostream>
5 #include <iomanip>
6 #include <vector>
7 #include "Employee.h"
8 #include "SalariedEmployee.h"
9 #include "CommissionEmployee.h"
10 #include "BasePlusCommissionEmployee.h"
11 using std::cout;
12 using std::vector;
13 void virtualViaPointer(const Employee* const); // prototype
14 void virtualViaReference(const Employee&); // prototype
15
16 int main() {
17     cout << std::fixed << std::setprecision(2); // set floating-point formatting
18
19     // create derived-class objects
20     SalariedEmployee salariedEmployee{
21         "Paul", "Chambers", "111-11-1111", 800};
22     CommissionEmployee commissionEmployee{
23         "Bill", "Evans", "333-33-3333", 10000, .06};
24     BasePlusCommissionEmployee basePlusCommissionEmployee{
25         "Miles", "Davis", "444-44-4444", 5000, .04, 300};
26
27     // output each Employee's information and earnings using static binding
28     cout << "EMPLOYEES PROCESSED INDIVIDUALLY USING STATIC BINDING\n"
29     << salariedEmployee.toString()
30     << "\nearned $" << salariedEmployee.earnings() << "\n\n"
31     << commissionEmployee.toString()
32     << "\nearned $" << commissionEmployee.earnings() << "\n\n"
33     << basePlusCommissionEmployee.toString()
34     << "\nearned $" << basePlusCommissionEmployee.earnings() << "\n\n";
35
36     // create and initialize vector of three base-class pointers
37     vector<Employee*> employees{&salariedEmployee, &commissionEmployee,
38         &basePlusCommissionEmployee};
39
40     cout << "EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING\n\n";
41
42     // call virtualViaPointer to print each Employee's information
43     // and earnings using dynamic binding
44     cout << "VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS POINTERS\n";
45
46     for (const Employee* employeePtr : employees) {
47         virtualViaPointer(employeePtr);
48     }
49
50     // call virtualViaReference to print each Employee's information
51     // and earnings using dynamic binding
52
53     cout << "VIRTUAL FUNCTION CALLS MADE OFF BASE-CLASS REFERENCES\n";
54
55     for (const Employee* employeePtr : employees) {
56         virtualViaReference(*employeePtr); // note dereferencing
57     }
58
59     // call Employee virtual functions toString and earnings off a
60     // base-class pointer using dynamic binding
61     void virtualViaPointer(const Employee* const baseClassPtr) {
62         cout << baseClassPtr->toString()
63         << "\nearned $" << baseClassPtr->earnings() << "\n\n";
64     }
65
66     // call Employee virtual functions toString and earnings off a
67     // base-class reference using dynamic binding
68     void virtualViaReference(const Employee& baseClassRef) {
69         cout << baseClassRef.toString()
70         << "\nearned $" << baseClassRef.earnings() << "\n\n";
71     }
72
73 }
```

Makefile:

```
1 # Makefile for the Employee project
2
3 CC=g++
4 CFLAGS=-std=c++11
5 DEPS = BasePlusCommissionEmployee.h CommissionEmployee.h Employee.h SalariedEmployee.h
6 OBJ1 = CommissionEmployee.o SalariedEmployee.o testEmployee.o BasePlusCommissionEmployee.o Employee.o
7 OBJ2 = CommissionEmployee.o SalariedEmployee.o testDynCasting.o BasePlusCommissionEmployee.o Employee.o
8
9 %.o: %.cpp $(DEPS)
10    $(CC) -c -o $@ $< $(CFLAGS)
11 all: testEmployee testDynCasting
12
13 testEmployee: $(OBJ1)
14    $(CC) -o $@ $^ $(CFLAGS)
15 testDynCasting: $(OBJ2)
16    $(CC) -o $@ $^ $(CFLAGS)
17
```

(non guardare dalla riga 15 in poi).

VIRTUAL TABLE

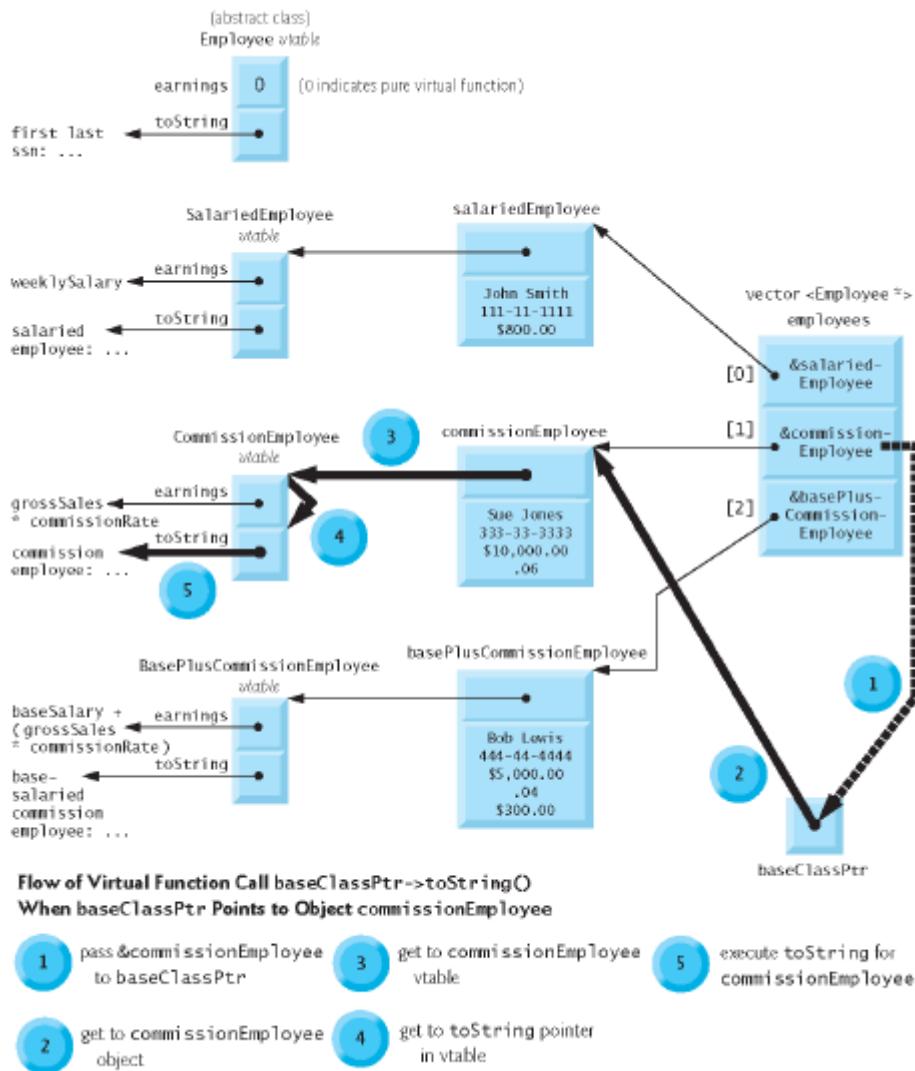
La realizzazione delle funzioni polimorfiche avviene attraverso una struttura di puntatori, ovvero:

Per ogni tipo di oggetto c'è una *vtable*, ovvero una tabella che contiene i puntatori alle funzioni virtuali della classe ed in particolare i puntatori contengono il primo indirizzo in memoria del codice della funzione; ogni oggetto ha in testa un puntatore alla *virtual table* (c'è una sola virtual table per tutti gli oggetti della classe poiché ovviamente le funzioni sono condivise all'interno della classe).

Allora quando invoco una certa funzione su un certo oggetto non faccio altro che prendere l'indirizzo dell'oggetto, ricavare da questo il tipo di oggetto ed in base al tipo di oggetto andare alla virtual table che implementa la funzione. Quindi c'è uno strato intermedio di puntatori che è a 3 livelli:

- 1) ogni oggetto è rappresentato mediante un riferimento a quell'oggetto;
- 2) ogni oggetto ha un puntatore alla virtual table;
- 3) la virtual table è fatta come un elenco di puntatori: puntatore alla prima funzione, poi alla seconda, poi alla terza e così via.

Attraverso questi 3 strati di puntatori avviene la realizzazione del polimorfismo.



In realtà però per come abbiamo implementato il programma non abbiamo considerato il fatto che la compagnia ha deciso di premiare i commerciali senior aggiungendo il 10% al loro salario. Per poter implementare questa cosa dobbiamo sapere se il l'impiegato in questione è un senior. Allora nel ciclo for del main dobbiamo inserire un if per stabilire su che tipo di oggetto stiamo calcolando lo stipendio. In questo caso possiamo usare il **downcasting**: downcasting vuol dire passare da classe base a classe derivata specifica. Nel nostro caso abbiamo un puntatore ad un oggetto di tipo classe base e facciamo il downcasting per andare più in basso nella gerarchia ed istanziamo un oggetto che sta più in basso nella gerarchia. L'**upcasting** è semplice, nel senso che un puntatore alla classe derivata lo definisco come puntatore alla classe base; tanto la classe derivata "fa parte" della classe base. Il downcasting invece è pericoloso, perché non si può prendere un impiegato generico e farlo diventare un `BasePlusCommissionEmployee`, mancherebbero delle informazioni per poterlo fare. Il downcasting quindi non è automatico e va fatto in maniera opportuna, ovvero con una parola chiave che è:

`dynamic_cast`

Lo *static_cast*, ovvero il cambiamento di tipo statico, viene effettuato a livello di compilazione; il *downcasting* non può essere effettuato con lo *static_cast* ma va effettuato con il *dynamic_cast* perché a tempo di compilazione non si saprà con che tipo di oggetto si ha a che fare, ma a tempo di esecuzione sì.

Allora nell'esempio va inserito nel main:

```
33     BasePlusCommissionEmployee* derivedPtr =
34         dynamic_cast<BasePlusCommissionEmployee*>(employeePtr);
35
36     // determine whether element points to a BasePlusCommissionEmployee
37     if (derivedPtr != nullptr) { // true for "is a" relationship
38         ...
39 }
```

Ovvero definisco un puntatore *derivedPtr* alla classe derivata e lo assegno ad *employeePtr* che è un puntatore alla classe base, poiché è quello che ho nel vettore definito prima. Quindi *employeePtr* lo promouovo ad essere un *BaseClassCommissionEmployee*: sto passando dalla classe base a destra alla classe derivata a sinistra; se facessi questo senza *dynamic_cast* il compilatore si arrabbierebbe. Quando uso il dynamic casting, se l'oggetto a destra è effettivamente un oggetto di classe derivata allora si effettua questo casting; nel caso contrario questa associazione produce un puntatore nullo. Da ciò l'if alla riga 37. Allora la riga 33 è un tentativo: vedo se *employeePtr*, puntatore alla classe base, lo posso istanziare come puntatore alla classe derivata; se sì allora posso usare ciò per aumentare il salario del 10% per un senior, altrimenti il puntatore alla classe derivata dichiarato viene istanziato come puntatore nullo. Quindi questo è un modo per controllare a runtime se si tratta di un *BasePlusCommissionEmployeeem*, ovvero un senior che deve avere un 10% in più; si fa il tentativo e se a tempo di esecuzione fallisce l'associazione con il downcasting allora non si tratta di un senior.

```
1 // testDynCasting.cpp
2 // Demonstrating downcasting and runtime type information.
3 // NOTE: You may need to enable RTTI on your compiler
4 // before you can compile this application.
5 #include <iostream>
6 #include <iomanip>
7 #include <vector>
8 #include <typeinfo>
9 #include "Employee.h"
10 #include "SalariedEmployee.h"
11 #include "CommissionEmployee.h"
12 #include "BasePlusCommissionEmployee.h"
13 using std::cout;
14 using std::endl;
15 using std::vector;
16 int main() {
17     // set floating-point formatting
18     cout << std::fixed << std::setprecision(2);
19
20     // create and initialize vector of three base-class pointers
21     vector<Employee*> employees{
22         new SalariedEmployee("John", "Smith", "111-11-1111", 800),
23         new CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06),
24         new BasePlusCommissionEmployee(
25             "Bob", "Lewis", "444-44-4444", 5000, .04, 300)};
```

```

27 // polymorphically process each element in vector employees
28 for (Employee* employeePtr : employees) {
29     cout << employeePtr->toString() << endl; // output employee
30
31     // attempt to downcast pointer
32
33     BasePlusCommissionEmployee* derivedPtr =
34         dynamic_cast<BasePlusCommissionEmployee*>(employeePtr);
35
36     // determine whether element points to a BasePlusCommissionEmployee
37     if (derivedPtr != nullptr) { // true for "is a" relationship
38         double oldBaseSalary = derivedPtr->getBaseSalary();
39         cout << "old base salary: $" << oldBaseSalary << endl;
40         derivedPtr->setBaseSalary(1.10 * oldBaseSalary);
41         cout << "new base salary with 10% increase is: $"
42             << derivedPtr->getBaseSalary() << endl;
43     }
44
45     cout << "earned $" << employeePtr->earnings() << "\n\n";
46 }
47
48 // release objects pointed to by vector's elements
49 for (const Employee* employeePtr : employees) {
50     // output class name
51     cout << "deleting object of "
52         << typeid(*employeePtr).name() << endl;
53
54     delete employeePtr;
55 }
56
57 }
58

```

Altro modo per effettuare il down-casting è quello di usare `typeid(*employeePtr.name())`, che restituisce il nome della classe a cui appartiene l'oggetto puntato dal puntatore; quindi il risultato sarà SalariedEmployee, CommissionEmployee, etc...

Esercizi da fare:

exercise

- Develop a polymorphic banking program. Create a vector of Account pointers to `SavingsAccount` and `CheckingAccount` objects. For each Account in the vector, allow the user to specify an amount of money to withdraw from the Account using member function `debit` and an amount of money to deposit into the Account using member function `credit`. As you process each Account, determine its type. If an Account is a `SavingsAccount`, calculate the amount of interest owed to the Account using member function `calculateInterest`, then add the interest to the account balance using member function `credit`. After processing an Account, print the up-dated account balance obtained by invoking base-class member function `getBalance`.
 - Write the three header files: `Account.h`, `SavingsAccount.h`, `CheckingAccount.h` and the relative .cpp implementation files
 - Write the driver program that instantiate an array of accounts and process it

Esercizio da svolgere

- Scrivere un programma c++ che permetta di gestire la bibliografia di un articolo scientifico.
 - Il programma deve gestire un elenco di pubblicazioni,
 - Tutte le pubblicazioni possono hanno un titolo, uno o poi' autori, ed un anno di pubblicazione
 - E possono essere dei
 - Libri, vanno memorizzati le informazioni sulla casa editrice e l'ISBN
 - Articoli su riviste, vanno memorizzati il titolo della Rivista (es IEEE Transaction of Computers), il numero della rivista, e le pagine (inizio e fine)
 - Articoli su atti di convegni, vanno memorizzati il titolo del convegno (es IEEE Conference on Machine Learning), la sede del convegno, e il numero di pagina
 - Il programma deve permettere di memorizzare una lista di pubblicazioni in un vettore, stampare la lista, ed calcolare il numero di pubblicazioni di ciascun tipo a partire dai dati memorizzati nel vettore.
 - La stampa deve essere fatta in ordine alfabetico di cognome (overload dell'operatore <)
 - Implementare l'ordinamento di un vettore di pubblicazioni
-

LEZIONE 20

FILE I/O- file sequenziali

Un SO ci permette di stampare piuttosto che su uno schermo, su un certo file (`cout`). Questa è la modalità di scrittura/lettura sequenziale, ovvero scriviamo e leggiamo da un file come facciamo su uno schermo. Una cosa più interessante è come scrivere/leggere da un file in maniera random. L'accesso random è più comodo ovviamente: ad esempio prima nelle cassette di musica per accedere ad una canzone bisognava passare per tutte le precedenti. Con i CD ROM ci è stata la rivoluzione che ha permesso di accedere in maniera randomica ai file. In generale l'accesso random è possibile o tramite un indice che tiene conto della posizione di un file o, in maniera più semplificata, immaginando che ogni informazione del file (ad esempio ogni canzone) occupi uno spazio fisso e quindi ad esempio salto i 4 record precedenti per saltare alla quinta canzone. Allora la cosa interessante è cercare di salvare le informazioni come dei file non testo ma binari. Quindi ad esempio 125 lo rappresentiamo non come sequenza di 1,2,5 ma come sequenza binaria a 4 byte in complemento a 2 della rappresentazione di 125.

Le funzioni che effettuano I/O su un file di tipo sequenziale sono esattamente le stesse che usiamo per fare l'I/O sullo schermo, solo va ridirezionato il flusso verso uno specifico file. Quindi prima va aperto un file in lettura o scrittura e poi si accede ad esso e quindi alle sue informazioni.

Il C++ vede ciascun file come una sequenza di bytes che termina con un carattere speciale che è l'EOF (End Of File). Quando un file viene aperto viene creato un oggetto ed un flusso è associato a quell'oggetto: c'è un flusso di informazioni che va dal programma al file se abbiamo aperto il file in output, o dal file verso il programma se abbiamo aperto il file in input. Ad esempio quando includiamo `<iostream>` vengono creati automaticamente gli oggetti: `cin`, `cout`, `cerr`, `clog`; questi oggetti sono i flussi che mettono in relazione un programma ed un file o un programma ed un device specifico. Ad esempio `cin` mette in comunicazione con la tastiera, `cout`, `clog`, `cerr` con lo schermo. Però ci sono diversi modi per "imbrogliare" e redirigere il flusso. Quindi ad esempio possiamo fare considerare al nostro programma come standard error su un particolare file un *pipe* (|). Quando vogliamo leggere o scrivere su un file dobbiamo includere i templates: `ifstream`, che dà tutte le funzionalità per leggere da un file, `ofstream`, che dà tutte le funzionalità per scrivere su un file, `fstream`, per leggere e scrivere.

SCRITTURA SU FILE

Nell'esempio successivo vediamo come creare un file sequenziale.

```

1 // opefile1.cpp
2 // Creating a sequential file.
3 #include <iostream>
4 #include <string>
5 #include <fstream> // contains file stream processing types
6 #include <cstdlib> // exit function prototype
7 using std::ofstream;
8 using std::ios;
9
10 int main() {
11     // ofstream constructor opens file
12     ofstream outClientFile("clients.txt", ios::out);
13
14     // exit program if unable to create file
15     if (!outClientFile) { // overloaded ! operator
16         std::cerr << "File could not be opened" << std::endl;
17         exit(EXIT_FAILURE);
18     }
19
20     std::cout << "Enter the account, name, and balance.\n"
21         << "Enter end-of-file to end input.\n? ";
22
23     int account; // the account number
24     std::string name; // the account owner's name
25     double balance; // the account balance
26
27     // read account, name and balance from cin, then place in file
28     while (std::cin >> account >> name >> balance) {
29         outClientFile << account << ' ' << name << ' ' << balance << std::endl;
30         std::cout << "? ";
31     }
32 }
```

L'obiettivo del programma è quello di creare un file che contenga una serie di informazioni su account bancari, raccolte come “triplette”. Allora includiamo *fstream* per poter usare le funzioni di lettura e scrittura su file. Linea 7 e linea 8 servono per includere ed utilizzare delle classi definite nella standard library, ovvero *ofstream*, che serve per scrivere in output, e *ios* che contiene una serie di servizi di I/O ed in particolare un insieme di funzioni (costanti) in cui viene rappresentato il tipo di operazione che si vuole effettuare. Alla riga 12 dichiariamo una variabile di tipo *ofstream*, *outClientFile*, inizializzata con il nome del file da aprire ed il tipo di modificatore di accesso; in realtà in questo modo abbiamo utilizzato il costruttore di default di *ofstream*, ma avremmo potuto anche procedere così:

```

ofstream outClientFile;
outClientFile.open("clients.txt", ios::out);
```

Ovvero avremmo potuto invocare la funzione *open* che crea la connessione tra il programma ed il file. Ad ogni modo dobbiamo specificare il file e la modalità di accesso ad esso. Dobbiamo però fare attenzione quando apriamo un file in output; infatti il contenuto del file viene azzerato, a meno che non lo apro in modalità *append*. Sia modalità *in* che *out* assumono che io apra il file in modalità sequenziale; se

specifico la modalità *binary* allora lo apro in modalità binaria. Se lo apro in modalità *append* (app) allora le operazioni di output vengono fatte alla fine del file.

- **ios::in** Open for input operations.
- **ios::out** Open for output operations.
- **ios::binary** Open in binary mode.
- **ios::ate** Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
- **ios::app** All output operations are performed at the end of the file, appending the content to the current content of the file.
- **ios::trunc** If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

Ovviamente quando noi chiediamo di scrivere un file al SO non sempre potremmo avere successo; potrebbe capitare che il SO mi risponda dicendomi che non posso avere accesso al file, il file potrebbe non esistere se lo apro in lettura, lo spazio disco potrebbe essere terminato, sto cercando di scrivere in una cartella che ha accesso in lettura ma non in scrittura, e così via. Insomma la creazione di questo flusso potrebbe fallire.

Se il file non esiste viene creato automaticamente; se esiste invece lo azzera e ricomincia a riscriverci, almeno che non si specifichi la modalità di accesso *append*.

La variabile *outClientFile*, che è un oggetto di tipo *ofstream*, se si trova all'interno di un'espressione che richiede un boolean come “==” o “!=” (gli operatori booleani per oggetti di questo tipo sono riscritti) allora sta ad indicare, rispettivamente, “se la richiesta di apertura del file nella modalità specificata ha avuto successo” ed il contrario. Quindi la variabile *outClientFile* è true se la richiesta al SO ha avuto successo, false nel caso contrario. Allora alla riga 15 viene effettuato proprio questo check. Nel caso in cui non sia stato possibile aprire il file allora il programma termina, e questo lo si fa mostrando a video un messaggio di errore tramite *cerr* ed *exit(EXIT_FAILURE)*. Altrimenti viene richiesto all'utente una serie di triplete account-name-balance, dove account è il codice del conto corrente. Ovviamente avendo dichiarato *name* come stringa non possiamo inserire spazi; se volessimo inserire spazi dovremmo utilizzare la funzione *getline*. Alla riga 28 il programma contiene ad effettuare un input di triplete da tastiera fin quando non viene inserito il carattere di fine file (su windows è *ctrl-Z*, su linux *ctrl-d*). Se invece l'operazione di trasferimento restituisce 0 o un numero <0 vuol dire che ho terminato l'input, poiché l'operazione mi restituisce il numero di trasferimenti effettuati dallo standard input ad una delle nostre variabili. Quindi fin quando un utente produce informazioni da inserire nel file allora quello che si legge da tastiera viene inserito nel file. Il file di output è proprio come abbiamo considerato lo standard output fino ad ora, solo che piuttosto

che redirigere la stampa dei valori letti da tastiera sullo schermo si trasferisce sul file specificato nella dichiarazione della variabile di tipo *ofstream*.

Ovviamente per fare bene il programma dovremmo fare dei controlli sul se *account* è un valore numerico, se *name* è una stringa consentita e se il *balance* è un valore numerico.

La funzione *exit* termina il programma corrente e restituisce un segnale al programma client che sta eseguendo il programma, ovvero in tal caso al SO. Quando il programma termina con successo restituisce un valore pari a 0. Un qualsiasi valore di ritorno del programma diverso da 0 rappresenta un errore. In questo caso EXIT_FAILURE rappresenta il valore di ritorno restituito al SO.

La riga 29 scrive i dati letti dallo standard input nel file .txt e lo fa tramite l'operatore “<<” che abbiamo utilizzato per la stampa sullo schermo.

Allora è sempre buona prassi chiudere il file con:

variabile_di_stream_associata_al_file .close() .

Infatti c’è un limite al numero di file che un certo programma può gestire.

Quando il programma finisce viene chiamato il distruttore di default della variabile associata al flusso, ma possiamo anche, come appena visto, chiudere esplicitamente il file con *.close()*.

Utilizzando *ios::app* invece che *ios::out*, come già detto, si accoderà il testo che si vuole inserire nel file al file stesso senza eliminarne il contenuto.

Vediamo come aprire il file clients.txt per poterlo leggere.

LETTURA DA FILE

```
1 // readfile.cpp
2 // Reading and printing a sequential file.
3 #include <iostream>
4 #include <fstream> // file stream
5 #include <iomanip>
6 #include <string>
7 #include <cstdlib>
8 using std::ifstream;
9 using std::ios;
10 void outputLine(int, const std::string&, double); // prototype
11
12 int main() {
13     // ifstream constructor opens the file
14     ifstream inClientFile("clients.txt", ios::in);
15
16     // exit program if ifstream could not open file
17     if (!inClientFile) {
18         std::cerr << "File could not be opened" << std::endl;
19         exit(EXIT_FAILURE);
20     }
21
22     std::cout << std::left << std::setw(10) << "Account" << std::setw(13)
23         << "Name" << "Balance\n" << std::fixed << std::showpoint;
24
25     int account; // the account number
26     std::string name; // the account owner's name
27     double balance; // the account balance
28
29     // display each record in file
30     while (inClientFile >> account >> name >> balance) {
31         outputLine(account, name, balance);
32     }
33 }
34
35 // display single record from file
36 void outputLine(int account, const std::string& name, double balance) {
37     std::cout << std::left << std::setw(10) << account << std::setw(13) << name
38         << std::setw(7) << std::setprecision(2) << std::right << balance << std::endl;
39 }
```

Stavolta definiamo una variabile non di tipo *ofstream* ma di tipo *ifstream*.

Fino alla riga 21 succede lo stesso che si è visto per la scrittura di file, ma fatto con la lettura. Ovviamente il file da aprire si deve trovare nella stessa cartella del programma a meno che non si specifichi il percorso. Alla riga 30 c'è “l'opposto” di quanto visto per la scrittura su file: c'è un while che contiene come condizione il flusso che va non dalla tastiera come visto prima ma dal file input alle 3 variabili account, name, balance. Ciò farà sì che avvenga la lettura del file fino a quando si hanno gli elementi da inserire. Questo se non si vuole invocare EOF, altrimenti basterebbe invocarlo con un while(!EOF); dopo vedremo.

Uso poi account, name, balance come input ad una funzione che stampa sullo schermo il numero dell'account indentato a sinistra con 10 spazi, il nome indentato a sinistra con 13 spazi, il balance indentato a destra e su cui usiamo doppia precisione e quindi 2 cifre decimali. Quello che esce se eseguiamo è una cosa del genere:

```
$ g++ readfile.cpp -o readfile -std=c++11
$ ./readfile
Account    Name        Balance
10          Michele      13.50
11          Giovanni    100.30
14          Franco       65.40
```

Showpoint invece serve a mostrare .0 decimale anche se il valore di balance è intero.

PUNTATORI GET E PUT

Quando apriamo un file è come se ci fosse un puntatore che mi dice qual è la posizione corrente in lettura ed in scrittura, che viene spostata in avanti man mano che leggiamo/scriviamo. La posizione di **get** è quella di lettura, quella di **put** è quella di scrittura. Se ad esempio apriamo un file in append sia il puntatore di get che quello di put saranno alla fine. Possiamo spostare questi puntatori con delle funzioni membri definite in ifstream e ofstream che sono:

- seek get, puntatore di input → **seekg;**
- seek put, puntatore di output → **seekp.**

Allora per saltare un certo numero di posizioni in lettura possiamo fare **seekg(n);** possiamo spostarci all'inizio o alla fine con **seekg(0)** e **seekg(end).** Oppure possiamo fare **seekg(n, ios::end)** per spostarci ad n byte dalla fine del file e analogamente **seekg(0, end)** sempre per spostarci alla fine.

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg(n);
// position n bytes in fileObject
fileObject.seekg(n, ios::cur);
// position n bytes back from end of fileObject
fileObject.seekg(n, ios::end);
// position at end of fileObject
fileObject.seekg(0, ios::end);
```

Es: in questo esempio vediamo come copiare il contenuto di un file in un altro file.

```
1 #include <iostream>
2 #include <fstream>
3 using std::endl;
4 using std::cout;
5 using std::ifstream;
6 using std::ofstream;
7 using std::ios;
8 int main() {
9     char *buffer = NULL; int length = 0;
10    {
11        ifstream is("hello.txt");
12        if(!is) {
13            cout << "Error opening file" << endl;
14            return 1;
15        }
16        is.seekg(0, ios::end); // move to *end* of file
17        length = is.tellg(); // get my offset into file
18        is.seekg(0, ios::beg); // move to *beginning* of file
19        buffer = new char[length];
20        is.read(buffer, length); // read file contents
21    } // is goes out of scope and is closed
22    {
23        ofstream os("hello2.txt");
24        if(!os) {
25            cout << "Error opening file" << endl;
26            delete[] buffer;
27            return 1;
28        }
29        os.write(buffer, length);
30        delete[] buffer;
31    } // os goes out of scope and is closed return 0;
32 }
```

Se il file hello.txt contiene ad esempio “Hello word!”, allora sarà creato un file hello2.txt, se non esiste, in cui viene scritto esattamente “Hello word!”.

La funzione `tellg` mi dice in che posizione si trova il puntatore `get` nel file. Ovviamente analogamente `tellp` mi dà la posizione del puntatore `put`.

Quindi: `seek` sposta il puntatore, `tell` dice dove sta. Queste 2 funzioni membro, specie `seek`, ci permettono di navigare i file in maniera random e non sequenziale.

Invece `read` è una funzione membro della classe `ifstream` (in realtà della classe superiore `istream`) che prende 2 parametri in ingresso: uno è un puntatore a `char`, l’altro è il numero di `char` che si vuole leggere. Per `write` vale lo stesso ma è una funzione membro che scrive invece che leggere ovviamente.

I file sono visti dal C come sequenze di byte e tipicamente quindi come sequenze di `char` visto che un `char` è rappresentato spesso su 8 bit.

In realtà potevamo procedere in maniera più intelligente, sfruttando gli argomenti del `main`. Ricordiamo che gli argomenti del `main` sono le informazioni che mettiamo sulla riga di comando. Allora invece che cablare all’interno del programma il nome del file da leggere e quello da scrivere li facciamo scegliere all’utente così che il programma è anche più facilmente utilizzabile. Gli argomenti sono:

- `int argc` ← numero di argomenti;
- `char* argv []` ← sequenza degli argomenti.

Nell’uso corretto il numero di argomenti sarà 3, poiché abbiamo `Programma`, `file1`, `file2`. Invece in `argv[0]` sarà il nome del programma e mi aspetto che in `argv[1]` c’è il file da cui copio ed in `argv[2]` il nome di quello in cui scrivo. A questo punto il programma sarà simile, solo che gli oggetti di tipo `ifstream` ed `ofstream` saranno istanziati con dentro non il nome del file in cui leggere/scrivere testo bensì `argv[1]` ed `argv[2]`. Quindi un programma fatto così copia `file1` in `file2`.

Vediamone l’implementazione.

```

1 #include <iostream>
2 #include <fstream>
3 using std::endl;
4 using std::cout;
5 using std::cerr;
6 using std::ifstream;
7 using std::ofstream;
8 using std::ios;
9 int main(int argc, char* argv[]) {
10    if (argc != 3){
11        cerr << "Usage: "<< argv[0] << " file1 file2\n";
12        exit(EXIT_FAILURE);
13    }
14    char *buffer = NULL; int length = 0;
15    {
16        ifstream is(argv[1]);
17        if(!is) {
18            cout << "Error opening file" << endl;
19            return 1;
20        }
21        is.seekg(0, ios::end); // move to *end* of file
22        length = is.tellg(); // get my offset into file
23        is.seekg(0); // move to *beginning* of file
24        buffer = new char[length];
25        is.read(buffer, length); // read file contents
26    } // is goes out of scope and is closed
27    {
28        ofstream os(argv[2]);
29        if(!os) {
30            cout << "Error opening file" << endl;
31            delete[] buffer;
32            return 1;
33        }
34        os.write(buffer, length);
35        delete[] buffer;
36    } // os goes out of scope and is closed return 0;
37 }

```

```

$ g++ fileposition2.cpp -o fileposition2 -std=c++11
$ ./fileposition2
Usage: ./fileposition2 file1 file2
$ ./fileposition2 hello.txt hello_new.txt
$ cat hello_new.txt
Hello World!

```

Esercizio da fare:

Case study (Deitel)

- Write a program that enables a credit manager to display the account information for those customers with zero balances (i.e., customers who do not owe the company any money), credit(negative) balances (i.e., customers to whom the company owes money), and debit (positive) balances (i.e., customers who owe the company money for goods and services received in the past).
- The program displays a menu and allows the credit manager to enter one of three options to obtain credit information. Option 1 produces a list of accounts with zero balances. Option 2 produces a list of accounts with credit balances. Option 3 produces a list of accounts with debit balances. Option 4 terminates program execution. Entering an invalid option displays the prompt to enter another choice.

Queste opzioni richieste rendono un po' più realistica l'organizzazione di un database perché tipicamente non è che si legge tutto il file, magari enorme, caricandolo in memoria, per trovare gli elementi desiderati. Allora dobbiamo far visualizzare un menu all'utente e selezionare solo i file scelti della voce scelta dall'utente.

Il programma dovrà essere fatto così (pseudocodice):

```
3 int main() {
4     //open the input file
5
6     while(user_not_choose 4){
7         while(!input_file.eof()){
8             //read from file an account
9             //if the account correspond to the request
10            //display the corresponding line
11        }
12        //reset input file and start from the beginning
13        //get next request
14    }
15 }
```

Ed immagino di dover scrivere 3 funzioni:

```
12 int getRequest(); //display the menu and gets the choice
13 bool shouldDisplay(int, double); //decide if the balance corresponds to the choice
14 void outputLine(int, const std::string&, double); //prints the current record
```

Selezionare 4 dal menu farà uscire l'utente. Ovviamente l'utente dovrà effettuare una scelta lecita (da 1a 4), e fin quando non lo farà bisogna riproporre il menu. La seconda funzione invece va vedere se il balance dell'account corrente dev'essere visualizzato o meno, e l'ultima funzione visualizza l'account corrente. Uso queste 3 funzioni nel main.

Il main sarà fatto così: (pag dopo)

```
18 int main() {
19     // ifstream constructor opens the file
20     ifstream inClientFile{"clients.txt", ios::in};
21
22     // exit program if ifstream could not open file
23     if (!inClientFile) {
24         cerr << "File could not be opened" << endl;
25         exit(EXIT_FAILURE);
26     }
27
28     // get user's request (e.g., zero, credit or debit balance)
29     int request{getRequest()};
30
31     // process user's request
32     while (request != 4) {
33         switch (request) {
34             case 1:
35                 cout << "\nAccounts with zero balances:\n";
36                 break;
37             case 2:
38                 cout << "\nAccounts with credit balances:\n";
39                 break;
40             case 3:
41                 cout << "\nAccounts with debit balances:\n";
42                 break;
43         }
44
45         int account; // the account number
46         std::string name; // the account owner's name
47         double balance; // the account balance
48
49         // read account, name and balance from file
50
51
52         // display file contents (until eof)
53         while (!inClientFile.eof()) {
54             // read account, name and balance from file
55             inClientFile >> account >> name >> balance;
56             // display record
57             if (shouldDisplay(request, balance)) {
58                 outputLine(account, name, balance);
59             }
60         }
61         inClientFile.clear(); // reset eof for next input
62         inClientFile.seekg(0); // reposition to beginning of file
63         request = getRequest(); // get additional request from user
64     }
65
66     cout << "End of run." << endl;
67 }
```

```

70 int getRequest() {
71     // display request options
72     cout << "\nEnter request\n"
73     << " 1 - List accounts with zero balances\n"
74     << " 2 - List accounts with credit balances\n"
75     << " 3 - List accounts with debit balances\n"
76     << " 4 - End of run" << std::fixed << std::showpoint;
77     int type; // request from user
78
79     do { // input user request
80         cout << "\n? ";
81         cin >> type;
82     } while (type < 1 || type > 4);
83
84
85     return type;
86 }
87
88 // determine whether to display given record
89 bool shouldDisplay(int type, double balance) {
90     // determine whether to display zero balances
91     if (type == 1 && balance == 0) {
92         return true;
93     }
94
95     // determine whether to display credit balances
96     if (type == 2 && balance < 0) {
97         return true;
98     }
99
100    // determine whether to display debit balances
101    if (type == 3 && balance > 0) {
102        return true;
103    }
104
105    return false;
106 }
107
108 // display single record from file
109 void outputLine(int account, const std::string& name, double balance) {
110     cout << std::left << std::setw(10) << account << std::setw(13) << name
111     << std::setw(7) << std::setprecision(2) << std::right << balance << std::endl;
112 }
```

In questo programma per leggere il file abbiamo utilizzato:

`while(!eof())`

questo è molto utile e lo useremo spesso.

Ci sono poi molti flag; `clear()` senza niente dentro li pulisce tutti tra cui quindi quello di `eof`.

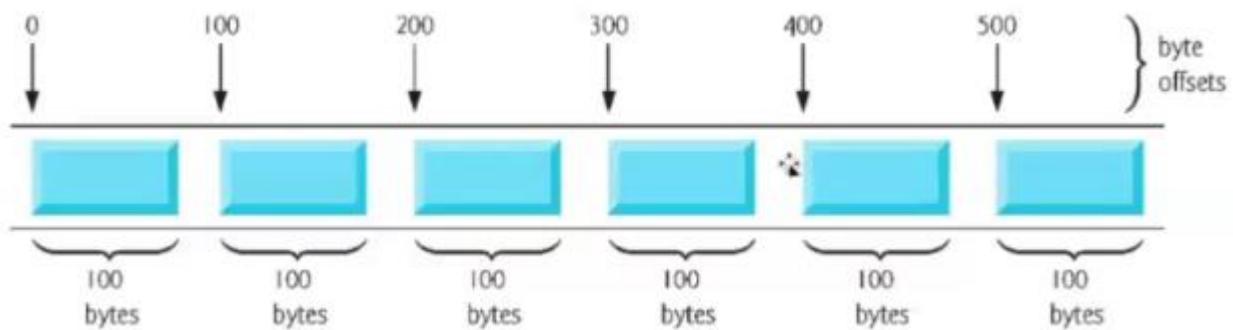
Piccola chicca: stiamo scrivendo il database per la banca; quindi il debito sarà della banca verso il cliente ed idem il credito.

LEZIONE 21

FILE I/O – File binari

Concentriamoci sull'accesso random ai file. Abbiamo già visto che tramite le funzioni `seekg` e `seekp` possiamo posizionare il cursore, ovvero il puntatore al file, in una posizione specifica all'interno di un file sequenziale. Il problema è che per poterci posizionare in una posizione dove c'è un'informazione di nostro interesse si pone il problema di cercare di avere il controllo su dove si trova l'informazione. I file sequenziali sono inappropriati per l'accesso istantaneo ad una certa informazione. L'approccio più semplice è quello di memorizzare le informazioni in lunghezza fissa; in questo modo si può calcolare facilmente la posizione dell'elemento d'interesse rispetto all'inizio del file. In questo modo si permette effettivamente l'accesso random. Un altro modo sarebbe quello di avere un indice, ma per ora non lo vediamo.

Quindi se supponiamo di avere tutti record della stessa dimensione, ad esempio 100 byte, è facile accedere ad un punto in mezzo al file senza scorrerlo fino a quel punto.



Certo, quando abbiamo degli oggetti che allocano dinamicamente questa ipotesi semplificativa cade subito e non possiamo più avere il controllo sulla dimensione degli oggetti da memorizzare. Utilizzando però delle librerie esterne si può fare una serializzazione degli oggetti, ovvero gli oggetti vengono sequenzializzati e spalmati su un certo numero di byte in modo poi che possano essere deserializzati facilmente.

Allora si utilizza la funzione `write`, che abbiamo già visto, per scrivere in uno stream un numero fissato di byte; quindi `write` prende in ingresso 2 parametri di cui il primo è un puntatore a `char`. Invece di scrivere il valore intero “number” in un file in questa maniera:

```
outfile << number;
```

ovvero con l'overloading dell'operatore di trasferimento, facciamo un'altra cosa: consideriamo la rappresentazione binaria di “number”, che sarà su 4 byte ed in complemento a 2, e passiamo proprio lei alla funzione `write`. Tuttavia abbiamo bisogno anche di un casting, che però non può essere statico, in quanto un puntatore ad `int` non può essere un puntatore a `char`, sono incompatibili; ed allora il puntatore ad `int` va

reinterpretato come puntatore a char, poiché il char è il tipo fondamentale nel senso che ogni elemento char è rappresentato con 1 Byte. Per questo uso `reinterpret_cast`. Come secondo parametro alla funzione `write` ci passo la dimensione in byte di `number` che sarà quella comune al tipo “intero” in questo caso. Ovviamente la scrittura comincerà da dove si trova il puntatore `get`. E quindi faremo:

```
outFile.write(reinterpret_cast<const char*>(&number), sizeof(number));
```

Questa cosa si chiama **SCRITTURA BINARIA** con `write`; consiste nella riscrittura di una variabile nella sua rappresentazione binaria, ovvero nella sua sequenza di byte, e si oppone a quella che avevamo visto in cui scrivevamo, ad esempio, la rappresentazione decimale del numero 125 come: carattere 1, carattere 2, carattere 5, tramite `outFile<<number`.

Ciò ci permetterà di rappresentare i nostri record a lunghezza variabile come record a lunghezza fissa (in alcuni casi) e ci semplificherà la vita.

Una curiosità è come funziona l'algoritmo jpeg per le immagini: è un algoritmo lossy, ovvero si perde un po' di qualità nella compressione. Jpeg sfrutta un algoritmo che si basa sulla FFT (Fast Fourier Transform) ed in particolare usa una trasformazione che si chiama “trasformazione di coseno”; considera l'immagine come un segnale, la rappresenta in un altro spazio che è lo spazio dei coefficienti della trasformata coseno e poi quantizza cercando di tagliare i coefficienti di scarso interesse. La cosa “importante” per noi è che questa conversione la effettua in binario: non si sa con quanti byte sarà rappresentata l'immagine perché dipende dal contenuto informativo dell'immagine, quindi anche se l'immagine ha dimensione di 512 x 512 non so a priori su quanti byte è rappresentata. E quindi il metodo della lunghezza fissa è una semplificazione che stiamo facendo, però la scrittura e la lettura in binario le possiamo comunque fare se sappiamo come leggere e scrivere le informazioni all'interno del file.

IMPORTANTE!

La cosa importante che dobbiamo tenere a mente è che `write` prende in ingresso un puntatore a `char`, e se non sto scrivendo un carattere ma ad esempio un numero devo metterci `reinterpret_cast` davanti al puntatore, e poi come secondo argomento ci vuole il numero di byte della variabile che stiamo scrivendo.

Possiamo provare a fare l'esempio del database bancario ma scrivendo nel database un file binario in cui rappresentiamo ogni record con: codice di conto, nome, cognome e balance. Quindi costruiamo una classe che contiene un intero, una stringa, una stringa ed un double.

Credit processing program

- Using random-access file-processing techniques, create a credit-processing program capable of storing at most 100 fixed-length records for a company that can have up to 100 customers. Each record should consist of an account number that acts as the record key, a last name, a first name and a balance. The program should be able to update an account, insert a new account, delete an account and insert all the account records into a formatted text file for printing.

Qui vediamo che iniziano i problemi; un intero lo sappiamo rappresentare in binario perché sono 4 byte, il double anche lo sappiamo rappresentare però come faccio con la stringa? Un cliente potrebbe chiamarsi Massimiliano ed un altro Ugo (omaggio al grande Massimo Troisi). Quindi per rappresentare nome e cognome e per avere il controllo sulla dimensione del record, ovvero di quanta memoria utilizzo per rappresentare ogni oggetto, dobbiamo fare un ragionamento. Ovvero dobbiamo stabilire un massimo di caratteri per il nome ed un massimo per il cognome.

```
1 // ClientData.h
2 // Class ClientData definition
3 #ifndef CLIENTDATA_H
4 #define CLIENTDATA_H
5
6 #include <string>
7
8 class ClientData {
9 public:
10    // default ClientData constructor
11    ClientData(int = 0, const std::string& = "",
12               const std::string& = "", double = 0.0);
13
14    // accessor functions for accountNumber
15    void setAccountNumber(int);
16    int getAccountNumber() const;
17
18    // accessor functions for lastName
19    void setLastName(const std::string&);
20    std::string getLastName() const;
21
22    // accessor functions for firstName
23    void setFirstName(const std::string&);
24    std::string getFirstName() const;
25
26    // accessor functions for balance
27    void setBalance(double);
28    double getBalance() const;
29 private:
30    int accountNumber;
31    char lastName[15];
32    char firstName[10];
33    double balance;
34 };
35
36 #endif
```

```

1 // ClientData.cpp
2 // Class ClientData stores customer's credit information.
3 #include <string>
4 #include "ClientData.h"
5 using std::string;
6
7 // default ClientData constructor
8 ClientData::ClientData(int accountNumberValue, const string& lastName,
9   const string& firstName, double balanceValue)
10  : accountNumber(accountNumberValue), balance(balanceValue) {
11    setLastName(lastName);
12    setFirstName(firstName);
13 }
14
15 // get account-number value
16 int ClientData::getAccountNumber() const {return accountNumber;}
17
18 // set account-number value
19 void ClientData::setAccountNumber(int accountNumberValue) {
20   accountNumber = accountNumberValue; // should validate
21 }
22
23 // get last-name value
24 string ClientData::getLastName() const {return lastName;}
25
26 // set last-name value
27 void ClientData::setLastName(const string& lastNameString) {
28   // copy at most 15 characters from string to lastName
29   size_t length(lastNameString.size());
30   length = (length < 15 ? length : 14);
31   lastNameString.copy(lastName, length);
32   lastName[length] = '\0'; // append null character to lastName
33 }
34
35 // get first-name value
36 string ClientData::getFirstName() const {return firstName;}
37
38 // set first-name value
39 void ClientData::setFirstName(const string& firstNameString) {
40   // copy at most 10 characters from string to firstName
41   size_t length(firstNameString.size());
42   length = (length < 10 ? length : 9);
43   firstNameString.copy(firstName, length);
44   firstName[length] = '\0'; // append null character to firstName
45 }
46
47 // get balance value
48 double ClientData::getBalance() const {return balance;}
49
50 // set balance value
51 void ClientData::setBalance(double balanceValue) {balance = balanceValue;}

```

```

1 // creditProcessing.cpp
2 // Creating a randomly accessed file.      Binary-mode is required
3 #include <iostream>                      If we want to write fixed-length records
4 #include <fstream>
5 #include <cstdlib>
6 #include "ClientData.h" // ClientData class definition
7 using std::ofstream;
8 using std::ios;
9
10 int main() {
11   ofstream outCredit{"credit.dat", ios::out | ios::binary};           Bitwise OR
12
13   // exit program if ofstream could not open file
14   if (!outCredit) {
15     std::cerr << "File could not be opened." << std::endl;
16     exit(EXIT_FAILURE);
17   }
18
19   ClientData blankClient; // constructor zeros out each data member
20
21   // output 100 blank records to file
22   for (int i{0}; i < 100; ++i) {
23     outCredit.write(
24       reinterpret_cast<const char*>(&blankClient), sizeof(ClientData));
25   }
26 }

```

Abbiamo utilizzato l'operatore *pipe*, ovvero l'OR logico, per aprire il file sia in modalità *out* che in modalità *binary*.

```

1 # Makefile for the creditProcessing project
2
3 CC=g++
4 CFLAGS=-std=c++11
5 DEPS = ClientData.h
6 OBJ = ClientData.o creditProcessing.o
7
8 %.o: %.cpp $(DEPS)
9         $(CC) -c -o $@ $< $(CFLAGS)
10 creditProcessing: $(OBJ)
11         $(CC) -o $@ $^ $(CFLAGS)
12

```

```

$ make
g++ -c -o ClientData.o ClientData.cpp -std=c++11
g++ -c -o creditProcessing.o creditProcessing.cpp -std=c++11
g++ -o creditProcessing ClientData.o creditProcessing.o -std=c++11
$ ./creditProcessing
$ ls
#Makefile#           ClientData.h          Makefile           creditProcessing      creditProcessing.o
ClientData.cpp       ClientData.o          credit.dat        creditProcessing.cpp

```

Ovviamente quando si visualizza il file su cui si è scritto non si capisce niente perché è scritto in binario (nel nostro caso con tutti zeri in codice ASCII di fatto) e compaiono simboli a cazzo. L'estensione .dat serve per riferirci a file generici.

Abbiamo scelto quindi, per mantenere fissa la lunghezza in bytes di ogni oggetto, di non utilizzare variabili membro di tipo stringa, ma di tipo vettori di char fissandone la lunghezza a priori per nome e cognome. Le funzioni set poi, sia per nome che per cognome, fanno la conversione da stringa a vettore di char tagliando eventualmente il nome se supera 9 caratteri ed il cognome se supera 14 caratteri; mette poi come ultimo carattere quello di fine stringa '/0' e quindi in questo modo si costruisce un vettore di char di 10 caratteri ed uno di 15 caratteri a partire da delle stringhe; in particolare grazie alla funzione *copy* della libreria *<string>*, che prende in ingresso: la stringa o l'array di char, come nel nostro caso, in cui copiare i caratteri della stringa su cui è invocata ed il numero di caratteri da copiare.

Aver scelto di rappresentare le stringhe nome e cognome come array di char a lunghezza fissa e non stringhe ci semplifica di molto la vita perché in questa maniera non abbiamo record a lunghezza variabile.

Comeabbiamo oramai capito le funzioni che scriviamo per la classe sono sicuramente quelle di set e get sulle variabili membro, che potrebbero anche fare altre cose come validare, più altre funzioni possibili. Ovviamente le funzioni non vengono rappresentate all'interno di ciascun oggetto, poiché comuni a tutta la classe; quindi le funzioni non richiedono alcuno spazio. L'oggetto occuperà in memoria un certo numero di Bytes per via delle variabili membro più eventualmente il puntatore alla v-table, nel caso in cui l'oggetto contenesse delle funzioni virtuali, e qualche puntatore

alla definizione della classe derivata nel caso in cui l'oggetto sia derivato da una classe base.

Particolarmente importante sarebbe fare la validazione su *accountNumber* in *setAccount*, poiché utilizzeremo questa variabile membro come indice per accedere alla posizione che vogliamo leggere. Quindi se io voglio che il database contenga da 1 a 100 non posso permettere di inserire come indice 100.000.

Per ora nell'esempio abbiamo creato un file binario "vuoto" ma con dei record di lunghezza fissa.

Fatto ciò creiamo un altro programma che apre il file appena creato, "credit.dat", lo legge, ci scrive sopra e si servirà della modalità binary perché considera tutti i record a lunghezza fissa.

Vediamo ora come LEGGERE da un file ad accesso random.

Quello che faremo è leggere tutti record ma stampare solo quelli non vuoti. Ci accorgiamo che un record è vuoto con il codice del conto (accountNumber) pari a 0. Infatti nel file ci sono comunque 100 record di tipo ClientData ma non tutti sono pieni.

```
1 // creditProcessingRead.cpp
2 // Reading a random-access file sequentially.
3 #include <iostream>
4 #include <iomanip>
5 #include <fstream>
6 #include <cstdlib>
7 #include "ClientData.h" // ClientData class definition
8 using std::ostream;
9 using std::ifstream;
10 using std::ios;
11 void outputLine(ostream&, const ClientData&); // prototype
12
13 int main() {
14     ifstream inCredit("credit.dat", ios::in | ios::binary);
15
16     // exit program if ifstream cannot open file
17     if (!inCredit) {
18         std::cerr << "File could not be opened." << std::endl;
19         exit(EXIT_FAILURE);
20     }
21
22     // output column heads
23     std::cout << std::left << std::setw(10) << "Acstd::count" << std::setw(16) << "Last Name"
24         << std::setw(11) << "First Name" << std::setw(10) << std::right << "Balance\n";
25
26     ClientData client; // create record
27
28     // read first record from file
29     inCredit.read(reinterpret_cast<char*>(&client), sizeof(ClientData));
30
31     // Read all records from file
32     while (inCredit) {
33         // display record
34         if (client.getAccountNumber() != 0) {
35             outputLine(std::cout, client);
36         }
37
38         // read next from file
39         inCredit.read(reinterpret_cast<char*>(&client), sizeof(ClientData));
40     }
41 }
42
43 // display single record
44 void outputLine(ostream& output, const ClientData& record) {
45     output << std::left << std::setw(10) << record.getAccountNumber()
46         << std::setw(16) << record.getLastName()
47         << std::setw(11) << record.getFirstName()
48         << std::setw(10) << std::setprecision(2) << std::right << std::fixed
49         << std::showpoint << record.getBalance() << std::endl;
50 }
```

Notare che abbiamo usato invece che EndOfFile direttamente la variabile di tipo ifstream. Infatti a questa è associato un bool che è true quando viene letto tutto il file, ovvero quando si arriva all'end of file.

```
1 # Maketfile for the creditProcessing project
2
3 CC=g++
4 CFLAGS=-std=c++11
5 DEPS = ClientData.h
6 OBJ = ClientData.o creditProcessingRead.o
7
8 %.o: %.cpp $(DEPS)
9     $(CC) -c -o $@ $< $(CFLAGS)
10 creditProcessingRead: $(OBJ)
11     $(CC) -o $@ $^ $(CFLAGS)
12
```

```
$ make
g++ -c -o ClientData.o ClientData.cpp -std=c++11
g++ -c -o creditProcessingRead.o creditProcessingRead.cpp -std=c++11
g++ -o creditProcessingRead ClientData.o creditProcessingRead.o -std=c++11
$ ./creditProcessingRead
Acstd::countLast Name      First Name    Balance
10      Mario            Rossi        10.00
21      Giorgio          Verdi        22.00
44      Franco           Biancho     13.50
$
```

Quindi: sappiamo inizializzare un file, sappiamo scrivere su un file e sappiamo leggere da un file. Possiamo allora immaginare di scrivere un programma per la gestione completa di un database che ci dovrebbe permettere di aggiungere un record, aggiornare un record esistente, eliminare un record esistente e stampare la lista degli account esistenti.

ESERCIZIO:

A transaction processing program

- Lets' write a complete program that allow tge management of a set of account records
- The program present a menu where the user is asked to:
 1. Enter a new record
 2. Update an existing record (ex deposit or debit some amount)
 3. Delete an exist record
 4. Print the list of existing accounts

Il main è fatto circa così, in pseudocodice:

```
1 main(){
2 // 
3 
4 fstream inOutCredit;
5 //open the file credit.dat in read and write, binary mode
6 //The main should
7 while(enterChoice != END){
8     //For every choice call the appropriate function
9     //1 - print
10    createTextFile(inOutCredit);
11 
12    //2 - update
13    updateRecord(inOutCredit);
14 
15    //3 - new
16    newRecord(inOutCredit);
17 
18    //4 delete
19    deleteRecord(inOutCredit);
20 }
21 }
22 
23 }
```

I servizi messi a disposizione dalla classe sono sempre gli stessi, quello che cambia è il main ovviamente. Vediamolo bene:

```
1 // transactionProcessing.cpp
2 // This program reads a random-access file sequentially, updates
3 // data previously written to the file, creates data to be placed
4 // in the file, and deletes data previously stored in the file.
5 #include <iostream>
6 #include <fstream>
7 #include <iomanip>
8 #include <cstdlib>
9 #include "ClientData.h" // ClientData class definition
10 using namespace std;
11 
12 enum class Choice {PRINT = 1, UPDATE, NEW, DELETE, END};
13 
14 Choice enterChoice();
15 void createTextFile(fstream& );
16 void updateRecord(fstream& );
17 void newRecord(fstream& );
18 void deleteRecord(fstream& );
19 void outputLine(ostream&, const ClientData& );
20 int getAccount(const char* const);
21 
22 int main() {
23 
24     // open file for reading and writing
25     fstream inOutCredit{"credit.dat", ios::in | ios::out | ios::binary};
26 
27     // exit program if fstream cannot open file
28     if (!inOutCredit) {
29         cerr << "File could not be opened." << endl;
30         exit (EXIT_FAILURE);
31     }
32 
33     Choice choice; // store user choice
34 
35     // enable user to specify action
36     while ((choice = enterChoice()) != Choice::END) {
37         switch (choice) {
38             case Choice::PRINT: // create text file from record file
39                 createTextFile(inOutCredit);
40                 break;
41             case Choice::UPDATE: // update record
42                 updateRecord(inOutCredit);
43                 break;
44             case Choice::NEW: // create record
45                 newRecord(inOutCredit);
46                 break;
47             case Choice::DELETE: // delete existing record
48                 deleteRecord(inOutCredit);
49                 break;
50             default: // display error if user does not select valid choice
51                 cerr << "Incorrect choice" << endl;
52                 break;
53         }
54 
55         inOutCredit.clear(); // reset end-of-file indicator
56     }
57 }
```

enterChoice

```
59 // enable user to input menu choice
60 Choice enterChoice() {
61     // display available options
62     cout << "\nEnter your choice\n"
63     << "1 - store a formatted text file of accounts\n"
64     << "    called \"print.txt\" for printing\n"
65     << "2 - update an account\n"
66     << "3 - add a new account\n"
67     << "4 - delete an account\n"
68     << "5 - end program\n? ";
69
70     int menuChoice;
71     cin >> menuChoice; // input menu selection from user
72     return static_cast<Choice>(menuChoice);
73 }
```

createTextFile

```
76 void createTextFile(fstream& readFromFile) {
77     ofstream outPrintFile("print.txt", ios::out); // create text file
78
79     // exit program if ofstream cannot create file
80     if (!outPrintFile) {
81         cerr << "File could not be created." << endl;
82         exit(EXIT_FAILURE);
83     }
84
85     // output column heads
86     outPrintFile << left << setw(10) << "Account" << setw(16)
87     << "Last Name" << setw(11) << "First Name" << right
88     << setw(10) << "Balance" << endl;
89
90     // set file-position pointer to beginning of readFromFile
91     readFromFile.seekg(0);
92
93     // read first record from record file
94     ClientData client;
95     readFromFile.read(
96         reinterpret_cast<char*>(&client), sizeof(ClientData));
97
98     // copy all records from record file into text file
99     while (!readFromFile.eof()) {
100         // write single record to text file
101         if (client.getAccountNumber() != 0) { // skip empty records
102             outputLine(outPrintFile, client);
103         }
104
105         // read next record from record file
106         readFromFile.read(
107             reinterpret_cast<char*>(&client), sizeof(ClientData));
108     }
109 }
```

updateRecord

```
112 void updateRecord(tstream& updateFile) {
113     // obtain number of account to update
114     int accountNumber{getAccount("Enter account to update")};
115
116     // move file-position pointer to correct record in file
117     updateFile.seekg((accountNumber - 1) * sizeof(ClientData));
118
119     // create record object and read first record from file
120     ClientData client;
121     updateFile.read(reinterpret_cast<char*>(&client), sizeof(ClientData));
122
123     // update record
124     if (client.getAccountNumber() != 0) {
125         outputLine(cout, client); // display the record
126
127         // request user to specify transaction
128         cout << "\nEnter charge (+) or payment (-): ";
129         double transaction; // charge or payment
130         cin >> transaction;
131
132         // update record balance
133         double oldBalance = client.getBalance();
134         client.setBalance(oldBalance + transaction);
135         outputLine(cout, client); // display the record
136
137         // move file-position pointer to correct record in file
138         updateFile.seekp((accountNumber - 1) * sizeof(ClientData));
139
140         // write updated record over old record in file
141         updateFile.write(
142             reinterpret_cast<const char*>(&client), sizeof(ClientData));
143     }
144     else { // display error if account does not exist
145         cerr << "Account #" << accountNumber
146             << " has no information." << endl;
147     }
148 }
```

newRecord

```
151 void newRecord(fstream& insertInFile) {
152     // obtain number of account to create
153     int accountNumber{getAccount("Enter new account number")};
154
155     // move file-position pointer to correct record in file
156     insertInFile.seekg((accountNumber - 1) * sizeof(ClientData));
157
158     // read record from file
159     ClientData client;
160     insertInFile.read(
161         reinterpret_cast<char*>(&client), sizeof(ClientData));
162
163     // create record, if record does not previously exist
164     if (client.getAccountNumber() == 0) {
165         string lastName;
166         string firstName;
167         double balance;
168
169         // user enters last name, first name and balance
170         cout << "Enter lastname, firstname, balance\n? ";
171         cin >> setw(15) >> lastName;
172         cin >> setw(10) >> firstName;
173         cin >> balance;
174
175         // use values to populate account values
176         client.setLastName(lastName);
177         client.setFirstName(firstName);
178         client.setBalance(balance);
179         client.setAccountNumber(accountNumber);
180
181         // move file-position pointer to correct record in file
182         insertInFile.seekp((accountNumber - 1) * sizeof(ClientData));
183
184         // insert record in file
185         insertInFile.write(
186             reinterpret_cast<const char*>(&client), sizeof(ClientData));
187     }
188     else { // display error if account already exists
189         cerr << "Account #" << accountNumber
190             << " already contains information." << endl;
191     }
192 }
```

deleteRecord

```
195 void deleteRecord(fstream& deleteFromFile) {
196     // obtain number of account to delete
197     int accountNumber{getAccount("Enter account to delete")};
198
199     // move file-position pointer to correct record in file
200     deleteFromFile.seekg((accountNumber - 1) * sizeof(ClientData));
201
202     // read record from file
203     ClientData client;
204     deleteFromFile.read(
205         reinterpret_cast<char*>(&client), sizeof(ClientData));
206
207     // delete record, if record exists in file
208     if (client.getAccountNumber() != 0) {
209         ClientData blankClient; // create blank record
210
211         // move file-position pointer to correct record in file
212         deleteFromFile.seekp((accountNumber - 1) * sizeof(ClientData));
213
214         // replace existing record with blank record
215         deleteFromFile.write(
216             reinterpret_cast<const char*>(&blankClient), sizeof(ClientData));
217
218         cout << "Account #" << accountNumber << " deleted.\n";
219     }
220     else { // display error if record does not exist
221         cerr << "Account #" << accountNumber << " is empty.\n";
222     }
223 }
```

outputLine and getAccount

```
220 void outputLine(ostream& output, const ClientData& record) {
221     output << left << setw(10) << record.getAccountNumber()
222         << setw(16) << record.getLastName()
223         << setw(11) << record.getFirstName()
224         << setw(10) << setprecision(2) << right << fixed
225         << showpoint << record.getBalance() << endl;
226 }
227
228
229
230
231
232 }
233
234 // obtain account-number value from user
235 int getAccount(const char* const prompt) {
236     int accountNumber;
237
238     // obtain account-number value
239     do {
240         cout << prompt << " (1 - 100): ";
241         cin >> accountNumber;
242     } while (accountNumber < 1 || accountNumber > 100);
243
244     return accountNumber;
245 }
```

Esercizi:

Esercizio da fare

- Gestione magazzino.
 - Scrivere un programma c++ che permetta la gestione di un magazzino
 - Il magazzino ha al piu' 100 articoli diversi
 - Per ogni articolo vogliamo rappresentare il codice (fra 0 e 99), il nome, la quantità ed il prezzo
 - I dati saranno memorizzati in un file "hardware.dat"
 - Il programma, una volta inizializzato l'nsieme dei record, permette di
 - inserire un record
 - Cancellare un record
 - Aggiornare un record
 - Stampare la lista di record presenti

Exercise (todo!)

- Implement a simple dictionary program that translates English words to Italian. The program should allow the user to repeatedly search for English words to translate, add new words to the dictionary, view the contents of the dictionary, and quit the program. The dictionary should be written to a file so that all changes made to the dictionary are persistent.
- The program can define an dictionary entry class containing the English word and the Italian word
- The dictionary is stored in a text file with a pair English-italian word on every row
- At the start the program load the dictionary in a vector of entries
- Then the menu present the various choices to the user
 - Implement the cases when the dictionary vector is sorted and the case when it is not sorted.
- Before the exit the program store the new dictionary in the text files

Questo esercizio lo dobbiamo fare in 2 modi: il primo è quello in cui il vettore del dizionario è disordinato e quindi gli elementi da ricercare vanno trovati tramite una ricerca sequenziale; nel secondo caso è più facile spostarsi nel vettore poiché è ordinato, ma è più difficile ordinare l'elemento nuovo appena inserito e quindi inserirlo nella giusta posizione. Quindi in realtà il primo caso è più semplice, il secondo è più difficile.

LEZIONE 22

L'ultimo programma assegnato chiedeva la creazione di un dizionario che traducesse da inglese ad italiano ed in cui quindi la parola inglese inserita dall'utente facesse da "chiave". Viene anche indicata la struttura dati da utilizzare, ovvero un vector. Tuttavia non è la struttura dati migliore da utilizzare, specie se è richiesto di mantenere il vettore ordinato. Infatti se da un lato possiamo usare la binary search con tempi logaritmici per ricercare elementi all'interno del vettore, dover ordinare il vettore ogni volta che si inserisce un nuovo elemento è una cosa abbastanza scocciante.

Esistono però delle strutture dati più efficienti che ci permettono di effettuare l'interrogazione ma anche l'aggiornamento, ovvero l'inserimento e la rimozione di un nuovo elemento, in maniera dinamica e quindi sono strutture dati **dinamiche**: aumentano e diminuiscono durante l'esecuzione del programma. Di queste strutture ne esistono tantissime ed abbiamo visto già array e vettori; adesso vedremo le liste a puntatori, alberi binari di ricerca, tabelle hash, e ne esistono molte altre. Quindi quello che ci importa è come queste strutture dati implementino l'inserimento di un elemento, la ricerca di un elemento, etc. ed in base alle nostre esigenze sceglieremo quella più idonea.

Vediamo come poteva essere implementato l'esercizio assegnato la scorsa volta. La struttura dati è una entry che contiene 2 stringhe, il file che contiene il dizionario è un file di testo. Quindi creiamo un file di testo, ad esempio eng_it.dict, così:

```
aunt      zia
car       auto
chair     sedia
college   scuola
come      vieni
father    padre
flower    fiore
food      cibo
fork      forchetta
fruit     frutta
```

Definiamo poi la struttura dati di base, che manterrà queste 2 informazioni (parola italiana - parola inglese).

Gestione di un dizionario

```
1 // dict.h
2 // declaration for type Word
3
4 #include <string>
5
6 #ifndef DICT_H
7 #define DICT_H
8 class Word {
9     public:
10     Word(std::string, std::string);
11     std::string getEngWord();
12     std::string getItWord();
13     void setEngWord(std::string);
14     void setItWord(std::string);
15
16     private:
17     std::string engWord;
18     std::string itWord;
19 };
20 #endif
```

```

1 // dict.cpp
2 // definition for class Word
3 #include "dict.h"
4
5 using std::string;
6
7 Word::Word(string eng, string it) :
8     engWord(eng), itWord(it) {}
9
10 string Word::getEngWord() { return engWord; }
11 string Word::getItWord() { return itWord; }
12 void Word::setEngWord(string word) { engWord = word; }
13 void Word::setItWord(string word) { itWord = word; }

```

Costruita questa struttura dati dovremo creare un vettore che contiene elementi di questo tipo e dovremo gestirlo in maniera ottimale. Il vettore in particolare serve a leggere il contenuto del file. Quindi definiremo un vettore di words e leggiamo dal file; questo è quello che fa per prima cosa il main. Poi dobbiamo proporre un menu:

- 1) ricerca;
- 2) aggiornamento;
- 3) visualizzazione del database;

Quindi servirà una funzione che effettua il caricamento, una che effettua la ricerca, una che effettua l'aggiornamento (aggiunta di una parola, non rimozione anche) ed una che salva il dizionario modificato nel file.

```

1 // dizionario.cpp
2 //program to implement a simple english-italian dictionary
3 #include <iostream>
4 #include <cstdlib>
5 #include <fstream>
6 #include <vector>
7 #include <limits>
8 #include <iomanip>
9 #include "dict.h"
10
11 using std::string;
12 using std::cout;
13 using std::endl; I
14 using std::ios;
15 using std::ifstream;
16 using std::ofstream;
17 using std::fstream;
18 using std::cin;
19 using std::setw;
20 using std::fixed;
21 using std::vector;
22 using std::cerr;
23 using std::left;
24 // utility function prototypes
25 int binSearchWord(const string, const vector<Word>&);
26 void addWord(const string, const string, vector<Word>&);
27 void loadDictionary(const string, vector<Word>&);
28 void writeDictionary(const string, const vector<Word>&);
29
30
31 int main() {
32     int choice{0};
33     const string DICTIONARY{"eng_it.dict"};
34     string word;
35     vector<Word> words; // list of words
36
37     // load dictionary words from file
38     loadDictionary(DICTIONARY, words);
39
40     cout << "\nWelcome to the English-Italian dictionary:" << endl;

```

Per fare bene questo esercizio avremmo dovuto anche controllare se la lista è di elementi è ordinata o meno e se, non è ordinata, ordinarla ad esempio con il quicksort. In questo primo pezzo di codice abbiamo usato loadDictionary(); vediamo com'è fatta.

```
160 void loadDictionary(const string dictFile, vector<Word>& words) {
161     // loads dictionary words from file into a vector
162     string engWord, itWord;
163     ifstream inFile(dictFile, ios::in);
164     if (!inFile) {
165         cerr << "Can't open dictionary. Quitting!" << endl;
166         exit(EXIT_FAILURE);
167     }
168
169     while (inFile >> engWord >> itWord) {
170         words.push_back(Word{engWord, itWord});
171     }
172
173 }
174 }
```

Viene passato dal file una coppia di elementi alla volta fino alla fine del file e viene inserita nel vettore *words* tramite un *push_back*; la coppia di elementi è identificata da una variabile della classe Word da noi definita e che prenderà in ingresso le 2 stringhe prelevate dal file. Questo finché è possibile, ovvero fino ad arrivare ad EoF. Ancora una volta, STIAMO ASSUMENDO CHE LE COPPIE SIANO ORDINATE. Questo perché se il file è stato costruito dal mio programma allora sarà sempre ordinato, perché le nuove coppie che inseriremo le inseriremo in maniera ordinata.

È il momento di approfondire un attimo come funzionano i vettori. I vettori sono degli array, solo che sono capaci di aumentare di dimensione e lo fanno in questa maniera: quando più della metà degli elementi sono occupati allora il vettore alloca un altro array che va a ricopiare tutti gli elementi del vecchio array nel nuovo. Quindi ogni volta che l'array inizia ad essere pieno la sua dimensione viene raddoppiata. Questa cosa è molto inefficiente, perché per aumentare le dimensioni dell'array con cui rappresento il vettore devo chiedere al SO memoria per allocare un certo numero di elementi e poi ricopiare di nuovo tutto l'array. Quindi è molto utile dal punto di vista dell'utente perché lo usiamo semplicemente come se fosse un vettore, ma non è la struttura dati più efficiente perché in realtà si basa su una struttura sottostante che è un array che può cambiare le dimensioni.

Tornando al programma, alla riga 36 abbiamo un vettore assunto ordinato. Proseguiamo nel main:

```

41 while (true) {
42     // display menu
43     cout << "\nWhat would like to do?:" << endl;
44     cout << "[1] Search" << endl;
45     cout << "[2] Add a word" << endl;
46     cout << "[3] View dictionary" << endl;
47     cout << "[4] Quit" << endl;
48     cout << endl;
49     cout << "Enter your choice: ";
50     cin >> choice;
51
52
53     switch (choice) {
54         case 1: { // search for word
55             string word;
56             cout << "Enter English word to search in small letters: ";
57             cin >> word;
58             int pos = binSearchWord(word, words);
59             if (pos < 0) {
60                 cout << "----\n" << endl;
61                 cout << "\n!" << word << "!!! << " not found.\n";
62                 cout << "----\n" << endl;
63             } else {
64                 Word w{words[pos]};
65                 cout << "----" << endl;
66                 cout << "\n!" << word << " means " << w.getItWord() << "\n" << endl;
}

```

Qui facciamo sì che il programma richieda all'utente una scelta, stampando il menu, fino a quando la scelta non è QUIT.

Per la ricerca è utilizzata la funzione *binSearchWord()* che effettua la ricerca binaria. Vediamola. In realtà potevamo riscrivere l'operatore <, però non lo abbiamo fatto proprio per vedere l'utilizzo della ricerca binaria. Però in generale conviene riscrivere l'operatore < facendo un overloading, poiché rende il programma più robusto e riutilizzabile.

```

120 int binSearchWord(const string searchWd, const vector<Word>& words) {
121     // searches the dictionary for a word
122     // return an empty string if the word is not found
123     // exercise: do a case-insensitive search
124
125
126     string itWord, engWord;
127     int n = words.size();
128     int left = 0;
129     int right = n-1;
130
131     while(left <= right){
132         int mid = (right+left)/2;
133         Word w{words[mid]};
134         if(w.getEngWord()== searchWd)
135             return mid;
136         if(w.getEngWord() < searchWd)
137             left = mid + 1;
138         else right = mid - 1;
139     }
140     return -1;
141 }

```

Torniamo al main.

```

72     case 2: { // add a word to the dictionary
73         string engWd, itWd;
74         cout << "Enter an English word: ";
75         cin >> engWd;
76         cout << "Enter the Italian meaning for '" << engWd << "': ";
77         cin >> itWd;
78         int pos = binSearchWord(engWd,words);
79         if (pos < 0) {
80
81             addWord(engWd,itWd,words);
82             cout << "" << engWd << "' added to dictionary." << endl;
83         } else {
84             cout << "\n" << engWd << "' is already in the dictionary." << endl;
85         }
86     }
87     break;

```

Nel caso 2, ovvero quello in cui l'utente vuole aggiungere una parola al dizionario, viene usata la ricerca binaria per vedere se la parola è già presente o meno nel vettore. Se non è presente allora viene utilizzata la funzione *addWord()* per aggiungere in maniera ordinata la parola alla lista di parole del dizionario. In realtà però è implementata in maniera maccheronica e poco efficiente; avremmo potuto capire dalla ricerca binaria che non ha avuto successo dove inserire l'elemento, ma non è affatto facile. Potrebbero chiedercelo in un colloquio Google (così, random).

```

145 void addWord(const string engWord, const string itWord,
146               vector<Word>& dict) {
147     // VERY INEFFICIENT!!!!
148     Word w{engWord,itWord};
149     int i=0;
150     while(i < dict.size()){
151         Word c{dict[i]};
152         if(c.getEngWord() < engWord)
153             i++;
154         else
155             break;
156     }
157     dict.insert(dict.begin()+i,w);
158 }

```

L'inserimento avrà comunque un costo lineare perché dovremo spostare nel caso peggiore tutti gli elementi per fare spazio al nuovo.

Quello che facciamo in *addWord* è una ricerca sequenziale: partiamo dal primo elemento e cerchiamo la prima posizione che contiene una parola maggiore della parola da cercare. L'inserimento lo effettuiamo a quel punto con la funzione *insert* che è una funzione di *vector* e che richiede l'oggetto da inserire nel dizionario (ovvero nel vettore) e la posizione in cui inserirlo. La posizione è rappresentata con un oggetto di tipo **iterator**; gli iterator (poi vedremo meglio) di una struttura dati sono dei particolari oggetti che permettono di esplorare la struttura dati in una maniera sequenziale. Gli iterator possono essere incrementati e sommati. Quindi *dict.begin()* vuol dire “posizione del puntatore al primo oggetto di *dict*”, *+i* fa stampare il puntatore alla *i*-esima posizione rispetto alla prima.

C'è poi il caso 3:

```
89     case 3: {
90         cout << endl;
91         cout << "Dictionary contents:" << endl;
92         for (int i{0}; i < words.size(); i++) {
93             Word wd = words[i];
94             cout << setw(12) << left << wd.getEngWord()
95                             << setw(12) << wd.getItWord() << endl;
96         }
97     }
98 }
```

Il caso 3 è quello in cui si mostra il contenuto del file ed è molto semplice, viene fatto un ciclo for per stampare gli elementi del vettore in cui sono contenuti i record del file.

C'è poi il caso 4 che deve terminare ed uscire dal programma; solo che prima di uscire dobbiamo salvare il dizionario.

```
100    case 4: { // quit
101        writeDictionary(DICTIONARY, words);
102        cout << "Bye!" << endl;
103        exit(EXIT_SUCCESS);
104    }
105 }
106
107 default: {
108     cout << "\nWrong choice. Must be a number between 1--3." << endl;
109     // clear cin and discard any extra characters
110     cin.clear();
111     //cin.ignore(numeric_limits<streamsize>::max(), '\n');
112 }
113 }
114 }
115 return 0;
116 }
```

A salvare il dizionario ci pensa la funzione *writeDictionary()*:

```
void writeDictionary(const string dictFile, const vector<Word>& words) {
    // writes the contents of the vector words to file
    // Should sort the words!
    string engWord, itWord;
    ofstream outFile(dictFile, ios::out);
    if (!outFile) {
        cerr << "Can't open dictionary. Quiting!" << endl;
        exit(EXIT_FAILURE);
    }

    for (int i{0}; i < words.size(); i++) {
        Word wd = words[i];
        engWord = wd.getEngWord();
        itWord = wd.getItWord();
        outFile << engWord << "\t" << itWord << endl;
    }
    outFile.close();
}
```

Questa funzione di specifico ha solo il tipo Word del vettore. Usando dei templates e la riscrittura dell'operatore << il programma diventerebbe del tutto generico.

CLASS TEMPLATES: LINKED LISTS

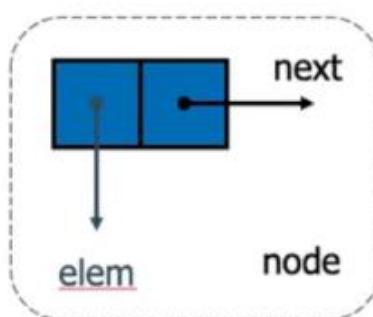
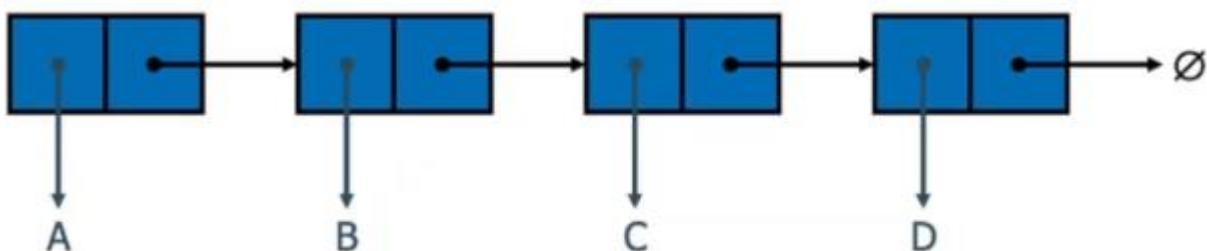
Il fatto che dobbiamo rappresentare in un vettore degli oggetti in maniera consecutiva è importante: dobbiamo occupare la memoria in maniera sequenziale.

Cosa sono le liste linkate, dette anche **liste a puntatori**?

Una lista a puntatori è una collezione di oggetti che si svincola dall'avere gli oggetti disposti in maniera consecutiva nelle caselle in memoria. L'idea di base è quella di avere degli oggetti che non sono collegati tra di loro in fila uno dietro l'altro ma tramite un *filo* che collega un oggetto a quello successivo, così da svincolarsi dalla posizione in cui questi oggetti sono memorizzati. Quindi nell'oggetto ci deve essere un'informazione su come raggiungere l'oggetto successivo; questo è il principio di base di una lista linkata (o lista a puntatori). Ovviamente, nel momento in cui si taglia un solo filo gli altri oggetti non li si trovano più: ne consegue che per raggiungere un oggetto si ha bisogno di attraversare gli oggetti precedenti, almeno che non si abbia un modo per evitarlo. Però il motivo per cui introduciamo le liste è che, svincolandoci dalla posizione in memoria degli oggetti, possiamo aggiungere o rimuovere un oggetto in tempo **O(1)**. Invece quando ragionavamo su caselle consecutive eravamo costretti a fare spazio ed a spostare ogni elemento, e nel caso peggiore dovevamo fare n spostamenti per posizionare l'elemento in prima posizione (se n è la dimensione dell'insieme di elementi prima dell'inserimento del nuovo elemento) → la complessità era di $O(n)$ → tempo inaccettabile. Una gestione dinamica fatta così, ad esempio con *vector*, può convenire solo laddove l'aggiornamento della struttura dati è raro.

In una lista concatenata è una collezione di oggetti, o *nodi*, in cui ogni nodo contiene:

- l'*informazione* da memorizzare nell'oggetto;
- un riferimento, o meglio un *puntatore*, all'oggetto consecutivo.



Quindi se io ho una lista posso aggiungere e rimuovere in testa ad essa in O(1), oppure se conosco l'elemento da rimuovere lo rimuovo in maniera molto semplice. Ad esempio, se nella lista in figura voglio rimuovere il nodo C basta che faccio puntare B a D, e basta quindi cambiare i collegamenti tra i nodi.

Come facciamo a capire se la lista è finita? Facile. L'ultimo elemento della lista punta al puntatore nullo. Quindi `nullPtr` è il segnale di fine lista ed è il contenuto di una delle 2 variabili del nodo, ovvero del puntatore al prossimo nodo.

Una lista vuota avrà una testa (`head`) che punterà ad un puntatore nullo.

Vediamo come implementare una lista concatenata.

Supponiamo che la mia lista voglia rappresentare delle stringhe (come nell'esempio del dizionario). Allora l'oggetto (o classe) "nodo" contiene 2 variabili membro: la stringa da rappresentare ed un puntatore al prossimo nodo: c'è una definizione ricorsiva (non proprio ricorsiva perché il nodo contiene una variabile che è riferimento ad un altro nodo e non il nodo stesso).

```
1 //Node.h
2 // a node in a list of strings
3 class StringNode {
4 private:
5     std::string elem; //element value
6     StringNode* next; //next item in the list
7
8     friend class StringLinkedList; //provide access to SingLinkedList
9 };
10
11 // StringLinkedList.h
12 #include "Node.h"
13 class StringLinkedList {                                     // a linked list of strings
14 public:
15     StringLinkedList();                                         // empty list constructor
16     ~StringLinkedList();                                       // destructor
17     bool empty() const;                                       // is list empty?
18     const string& front() const;                                // get front element
19     void addFront(const string& e);                            // add to front of list
20     void removeFront();                                       // remove front item list
21 private:
22     StringNode* head;                                         // pointer to the head of list
23 };
```

Creata la classe "Nodo" creiamo anche quella "Lista". Quest'ultima classe ha un'unica variabile membro che è il puntatore alla testa della lista, necessario poiché se si risale al primo elemento della lista allora si risale facilmente a tutti gli altri elementi. Notiamo che nell'esempio di sopra sulle liste di stringhe abbiamo definito nella classe "Nodo" la classe "Lista" come amica: questo perché quest'ultima ha bisogno di accedere alle variabili private della prima. Ci sono poi vari servizi messi a disposizione dalla classe "Lista", quali: un costruttore, un distruttore, una funzione booleana che serve a capire se la lista è vuota, una funzione per ottenere l'elemento in testa alla lista e altre, ma concentriamoci un attimo sull'ultima funzione menzionata. Questa funzione, `front`,

restituisce una stringa, ed in particolare un riferimento costante alla stringa. Perché per accedere ad un elemento della lista, tramite quindi un'interrogazione, uso una funzione che restituisce un riferimento costante ad un oggetto della lista? Perché è buona prassi procedere così? Perché in questa maniera il client che invoca la funzione riceve il riferimento all'oggetto, in questo caso alla stringa, senza però poter modificare l'oggetto memorizzato nella collezione. Il riferimento come al solito serve per evitare la copia dell'elemento del nodo che magari potrebbe essere una stringa molto lunga. L'ultimo *const* invece sta a dire che la funzione non deve alterare l'oggetto e quindi le variabili membro.

Gli altri servizi messi a disposizione dalla lista sono aggiunta e rimozione di elementi alla/dalla testa, tramite un tempo di O(1).

Aggiungere e rimuovere usando solo la testa della lista. Cosa ricorda? BEH OVVIAEMENTE LO STACK. La lista è la struttura dati ideale per rappresentare lo stack.

Vediamo com'è implementata la lista.

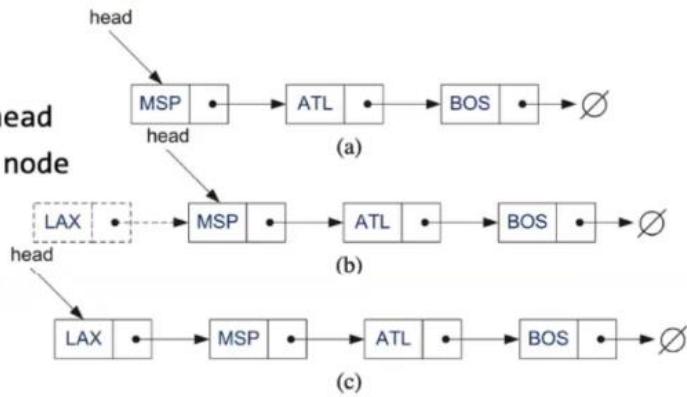
```
1 //StringLinkedList.cpp
2 StringLinkedList::StringLinkedList()           // constructor
3   : head{nullptr} { }
4
5 StringLinkedList::~StringLinkedList()          // destructor
6   { while (!empty()) removeFront(); }
7
8 bool StringLinkedList::empty() const           // is list empty?
9   { return head == nullptr; }
10
11 const string& StringLinkedList::front() const // get front element
12   { return head->elem; }
```

Il costruttore inizializza la lista ad una lista vuota ed infatti inizializza *head* con un puntatore nullo. Il distruttore usa la funzione *removeFront* che rimuove l'oggetto all'inizio; poi vediamo come. La funzione *empty* è banale; controlla se la lista è vuota e lo fa controllando se *head* contiene *nullPtr*. Un'osservazione sull'ultima funzione *front*: l'operatore “ \rightarrow ” fa accedere ad una porzione di memoria, nel senso che fa accedere alla variabile puntata dal puntatore. Tuttavia se la lista è vuota allora *head* avrà un puntatore nullo al suo interno e quindi con la freccia cerchiamo di accedere ad un elemento che non ha senso! Quindi ogni volta che usiamo la freccia su un puntatore dobbiamo fare attenzione controllando prima che il puntatore non sia nullo.

Vediamo bene come aggiungere elementi alla lista.

Add at the front

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



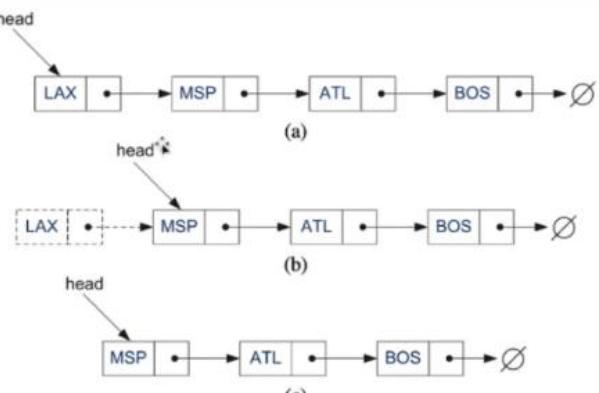
```
16 void StringLinkedList::addFront(const string& e) { // add to front of list
17     StringNode* v = new StringNode; // create new node
18     v->elem = e; // store data
19     v->next = head; // head now follows v
20     head = v; // v is now the head
21 }
```

MSP, ATL, BOS e LAX sono solo acronimi di aeroporti. Quello che ci importa è che sono delle stringhe. Quindi: creiamo un nuovo nodo. *Nell'element* del nuovo nodo ci mettiamo l'informazione, ovvero l'elemento da inserire, il *next* di questo nuovo nodo facciamo in modo invece che sia uguale ad *head*, e poi spostiamo *head* in modo che punti all'elemento appena inserito.

Vediamo ora la rimozione.

Removing at the front

1. Update head to point to next node in the list
2. Allow the system to reclaim the former first node



```
26 void StringLinkedList::removeFront() { // remove front item
27     if (empty())
28         throw std::runtime_error("List is empty");
29     StringNode* old = head; // save current head
30     head = old->next; // skip over old head
31     delete old; // delete the old head
32 }
```

Anche la rimozione avviene a partire dalla testa. Come prima cosa si controlla se la lista è vuota; in tal caso non si può rimuovere niente e quindi si lancia un'eccezione. Nel caso contrario salviamo il puntatore al primo elemento, spostiamo *head* in modo che punti a *next* (nell'immagine sarebbe più corretto se scrivessimo *head*=*head* \rightarrow *next*) ed eliminiamo il vecchio puntatore rilasciando la memoria al SO.

Oss:

puntatore \rightarrow elemento

equivale a:

*puntatore.elemento

Nel nostro caso abbiamo puntatori a membri di una classe e quindi usiamo la freccia, ma potremmo anche dereferenziare ed usare il punto.

LEZIONE 23

Abbiamo introdotto le liste linkate che sono una struttura dati importantissima perché dinamica (ovvero alloca dinamicamente la memoria) e che collega i vari nodi della lista attraverso dei link (per questo “linkata”).

La lista è fatta da nodi, ogni nodo ha una coppia composta da un’unità di informazione e da un link all’elemento successivo della lista. Con questo approccio si possono realizzare cose molto potenti. Infatti per ora colleghiamo ogni nodo al nodo successivo, ma potremmo in realtà costruire degli alberi in cui un nodo è collegato, ad esempio, al sottoalbero di sinistra ed al sottoalbero di destra, creando così una gerarchia. Oppure potremmo costruire un grafo: un grafo è un insieme di vertici collegati tra loro e quindi la lista è in grado di rappresentare tranquillamente un grafo; le mappe di Google maps sono composte da liste linkate e matrici, a livello di strutture dati. Esistono poi delle strutture dati astratte in cui definiamo un insieme di oggetti, i servizi messi a disposizione dagli oggetti e come possiamo implementare con delle strutture dati reali (come array, vettori, liste concatenate) le funzionalità della struttura dati astratta.

Tenere a mente che:

ALGORITMI + STRUTTURE DATI = PROGRAMMI .

Ovvero, i programmi sono in linea di massima degli algoritmi basati su strutture dati specifiche. Se cambiamo la struttura dati cambierà l’algoritmo e quindi il programma sarà più o meno efficiente in base al nuovo algoritmo.

CLASS TEMPLATES

L’ultima lezione abbiamo visto la realizzazione di una lista di stringhe. Passeremo ai templates che sono potentissimi perché permettono di generalizzare del tutto un programma. La riscrittura di un template viene effettuata a tempo di compilazione quando dichiariamo, ad esempio, che un certo nodo contiene un intero invece che una stringa. Infatti la riscrittura di un template dipende inevitabilmente dal client che ne fa uso e quindi il template viene riscritto a tempo di compilazione e va compilato insieme al client. Esempio di funzione template:

```
6 template <typename T>
7 T genericMin(T a, T b) { // returns the minimum of a and b
8     return (a < b ? a : b);
9 }
```

Questa funzione generica calcola il minimo tra 2 elementi. chiaro che cambierà tutto se invoco la funzione passando 2 stringhe o se la invoco passando 2 int; questo perché cambia l’operatore “<”. Quindi di fatto una funzione template la posso invocare su qualsiasi oggetto, basta che per quell’oggetto siano definiti, o eventualmente riscritti, gli operatori di cui la funzione fa uso.

Questo concetto può essere esteso alle classi: posso definire una classe generica che contiene al suo interno una variabile membro di tipo T, dove T è un type generico.

Es:

```
1 template <typename T> class BasicVector { // a simple vector class
2 public:
3     BasicVector(int capac = 10); // constructor
4     T& operator[](int i){ return a[i]; } // access element at index i
5 // ... other public members omitted
6 private:
7     T* a; // array storing the elements
8     int capacity; // length of array a
9 };

11 template <typename T> // constructor
12 BasicVector<T>::BasicVector(int n) {
13     capacity = n;
14     a = new T[capacity]; // allocate array storage
15 }
```

In questo esempio implementiamo noi i vettori definendo un template generico che chiamiamo *BasicVector* che mette a disposizione un costruttore + altre informazioni tra cui le variabili private della classe. Allora la classe contiene una variabile array di tipo generico T e la dimensione di questo array.

La sintassi è questa:

per definire la classe come template → `template <typename T> NameClass`

per scrivere il costruttore → `template <typename T> NameClass <T>::NameClass(...)`

Alla fine basta anteporre `template <typename T>` ed utilizzare come al solito lo scopo della classe a cui appartiene la funzione membro, in questo caso il costruttore.

```
18 int main(){
19     BasicVector<double> V(10);
20     for (int i= 0; i < 10; ++i)
21         V[i] = i*i/2.0;
22     for (int i= 0; i < 10; ++i)
23         std::cout << V[i] << " ";
24     std::cout << std::endl;
25 }
```

```
$ g++ BasicVector.cpp -o BasicVector -std=c++11
$ ./BasicVector
0 0.5 2 4.5 8 12.5 18 24.5 32 40.5
```

Questo di sopra è l'utilizzo di un oggetto *BasicVector* con tipo specificato come double. Adesso divertiamoci a rendere generica la Linked List vista la scorsa lezione.

LINKED LIST

```
1 //Node.h
2
3 template <typename NODETYPE> class LinkedList; //forward declaration of SLinkedList
4 // so it can be declared as friend
5
6 template <typename NODETYPE>
7 class Node { // singly linked list node
8 private:
9     NODETYPE elem; // linked list element value
10    Node<NODETYPE>* next; // next item in the list
11
12    friend class LinkedList<NODETYPE>; // provide SLinkedList access
13 };
14
```

NODETYPE è il nome del tipo generico con cui abbiamo costruito il template. La classe la chiamiamo Node e contiene 2 variabili membro che saranno un elemento di tipo NODETYPE ed un puntatore ad un Node contenente il tipo NODETYPE. Insomma ogni volta che c'è una funzione o una classe templata va specificato il tipo generico, nel nostro caso NODETYPE. Infine viene dichiarata come classe amica LinkedList, ma anche lei è templata e quindi ci vuole <NODETYPE> specificato e soprattutto va anche messa all'inizio una definizione di questa classe come classe templata.

Ora definiamo la linked list.

```
1 //LinkedList.h
2 //A generic LinkedList class template
3 #include "Node.h"
4 #include <iostream>
5
6 template <typename NODETYPE>
7 class LinkedList { // a singly linked list
8 public:
9     LinkedList(); // empty list constructor
10    ~LinkedList(); // destructor
11    bool empty() const; // is list empty?
12    const NODETYPE& front() const; // return front element
13    void addFront(const NODETYPE& e); // add to front of list
14    void removeFront(); // remove front item list
15 private:
16    Node<NODETYPE>* head; // head of the list
17 };
18
19 #include "LinkedList.cpp"
```

- We include here the implementation since it cannot be compiled independently from the client.
- When the client specializes the class with the desired type this can be compiled

Notiamo una particolarità: abbiamo incluso l'implementazione delle funzioni nel file header; questo perché quando il client include l'header deve includere anche l'implementazione delle funzioni, poiché non possiamo compilare l'implementazione in maniera indipendente dal client e linkare dopo.

Vediamola l'implementazione.

```
1 //LinkedList.cpp
2 //Implementation of the generic Linked list
3 //this file should be included in the client source file
4 //in order to avoid linking errors
5 template <typename NODETYPE>
6 LinkedList<NODETYPE>::LinkedList() // constructor
7   : head(nullptr) {}
8 template <typename NODETYPE>
9 bool LinkedList<NODETYPE>::empty() const // is list empty?
10 { return head == nullptr; }
11
12
13 template <typename NODETYPE>
14 const NODETYPE& LinkedList<NODETYPE>::front() const // return front element
15 { return head->elem; }
16
17 template <typename NODETYPE>
18 LinkedList<NODETYPE>::~LinkedList() // destructor
19 { while (!empty()) removeFront(); }
20
21 template <typename NODETYPE>
22 void LinkedList<NODETYPE>::addFront(const NODETYPE& e) { // add to front of list
23   Node<NODETYPE>* v = new Node<NODETYPE>; // create new node
24   v->elem = e; // store data
25   v->next = head; // head now follows v
26   head = v; // v is now the head
27 }
28 template <typename NODETYPE>
29 void LinkedList<NODETYPE>::removeFront() { // remove front item
30   if(head == nullptr){
31     throw std::invalid_argument("List is empty");
32   }
33   Node<NODETYPE>* old = head; // save current head
34   head = old->next; // skip over old head
35   delete old; // delete the old head
36 }
```

Al solito, prima di usare qualsiasi puntatore è sempre meglio verificare che quel puntatore non sia nullo. Questo controllo lo potremmo fare all'interno delle funzioni che usano il puntatore, come *removeFront* tramite la funzione *empty*. In questo caso non abbiamo invocato *empty*.

Allora scriviamo un generico main con la specializzazione della linked list:

```
1 #Makefile for testGenericLinkedList
2 CC=g++
3 CFLAGS=-std=c++11
4 DEPS = Node.h LinkedList.h LinkedList.cpp
5 OBJ = testGenericList.o
6
7 %.o: %.cpp $(DEPS)
8   $(CC) -c -o $@ $< $(CFLAGS)
9
10 testGenericLinkedList: $(OBJ)
11   $(CC) -o $@ $^ $(CFLAGS)
```

```
7 g++ -c -o testGenericList.o testGenericList.cpp -std=c++11
8 g++ -o testGenericLinkedList testGenericList.o -std=c++11
9 $ ./testGenericLinkedList
10 LAX
11 20
```

Fare quest'esercizio.

Exercise

- Extend the `LinkedList` example:
 - Implement the member function `size()` that returns the number of elements in the list
 - Implement the overloading of the `operator[](int i)` that returns the i -th element store in the list.
 - Implement the search function to locate the node containing a specific element
 - Implement the delete function that deletes an element of the list.

Questo esercizio ci permette, tramite la riscrittura dell'operatore `[]`, di accedere ad un elemento i -esimo della lista con la stessa sintassi con cui accederemmo ad un elemento di un array. Cancellare un elemento generico poi non è facile, perché serve conoscere il puntatore di prima e quello di dopo in modo da far puntare quello di prima a quello di dopo. Quindi per fare ciò dovrò scorrere la lista e mantenere il nodo attuale e quello precedente; quello successivo ce l'ho perché è il `next`. Quindi nello scorrimento devo mantenere 2 puntatori, perché mi serve sapere l'informazione sul nodo precedente. Questo ovviamente richiede anche il saper scorrere la lista fino a trovare l'elemento desiderato → search function(3° punto).

Addirittura possiamo scrivere una funzione template per manipolare gli oggetti specializzati di una classe template. Ad esempio, supponiamo di voler scrivere una certa funzione su una lista, che è un template; questa funzione sarà a sua volta template. Importante implementare tutte le funzioni su una lista come invertire gli elementi della lista, aggiungere e rimuovere dalla coda di una lista invece che dalla testa, e così via; le useremo spesso.

Es:

```
|   |
|   |
|   |
```

```

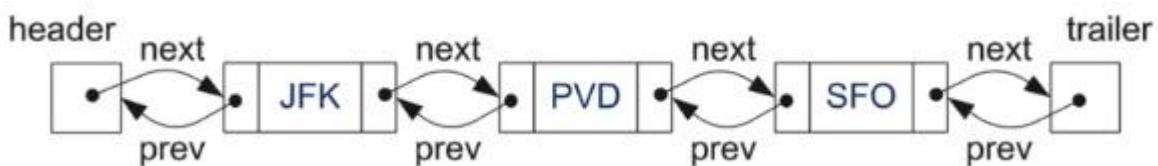
1 //testgenericList2.cpp
2 //function template to manipulate
3 //specialized class templates
4
5 #include <iostream>
6 #include <string>
7 #include "LinkedList.h"
8
9 using std::string;
10 using std::cout;
11
12 template <typename T>
13 void printFront(LinkedList<T>& L);
14
15 int main(){
16     LinkedList<string> L;
17     L.addFront("FCO");
18     L.addFront("LAX");
19     printFront(L);
20
21
22     LinkedList<int> L2;
23     L2.addFront(10);
24     L2.addFront(20);
25     printFront(L2);
26 }

```

Poiché noi siamo furbi possiamo semplificarcia la vita soprattutto per quanto riguarda aggiunta e delezione. Rimuovere un elemento generico da una lista è *time consuming* perché non abbiamo un metodo veloce di accedere al nodo precedente a quello attuale. Ma nessuno ci vieta di costruire una struttura dati Nodo dove ogni nodo contiene l'elemento (*elem*), il puntatore all'elemento successivo (*next*) ed il puntatore a quello precedente (*prev*). Una struttura dati composta da tanti nodi fatti così si chiama:

LISTA DOPPIAMENTE LINKATA

La cosa intelligente che possiamo fare è utilizzare 2 nodi “sentinelle”, che non contengono alcuna informazione, per identificare la testa (*header*) e la coda (*trailer*): in questa maniera posso scrivere l'aggiunta e la rimozione senza dover verificare se la lista è vuota o meno. Infatti se la lista è vuota avrà comunque un *header* ed un *trailer* e quindi il *next dell'header* sarà il *trailer* ed il *prev del trailer* sarà *l'header*.



Allora dobbiamo modificare la struttura dati “Nodo” prima e quella “Lista” poi, in modo tale da avere un puntatore in più sia nella prima (*prev*) sia nella seconda (*trailer*).

```

1 //DNode.h
2 template <typename T> class DLinkedList; //forward declaration
3
4 template <typename T>
5 class DNode {                                     // list element type
6     private:                                       // doubly linked list node
7         T elem;                                     // node element value
8         DNode* prev;                                // previous node in list
9         DNode* next;                                // next node in list
10        friend class DLinkedList<T>;                // allow DLinkedList access
11    };

1 //DLinkedList.h
2 #include "DNode.h"
3 #include <iostream>
4 template <typename T>
5 class DLinkedList {                           // doubly linked list
6     public:
7         DLinkedList();                         // constructor
8         ~DLinkedList();                      // destructor
9         bool empty() const;                  // is list empty?
10        const T& front() const;              // get front element
11        const T& back() const;               // get back element
12        void addFront(const T& e);          // add to front of list
13        void addBack(const T& e);           // add to back of list
14        void removeFront();                 // remove from front
15        void removeBack();                  // remove from back
16        std::string toString();             // local type definitions
17     private:                                // list sentinels
18         DNode<T>* header;                  // local utilities
19         DNode<T>* trailer;
20     protected:
21         void add(DNode<T>* v, const T& e); // insert new node before v
22         void remove(DNode<T>* v);        // remove node v
23    };
24
25 #include "DLinkedList.cpp"

```

Una lista doppiamente linkata potrebbe essere utile per rappresentare una coda. Ma ancora una volta, la coda, così come lo stack, è una struttura dati astratta che può essere implementata in vari modi. Quindi non confondiamo struttura dati astratta con struttura dati utilizzata per la rappresentazione.

In *DLinkedList.h* abbiamo inserito, oltre a tutte le funzioni che lavorano sulla coda della lista, 2 funzioni per aggiungere e rimuovere elementi in posizioni qualunque nella lista e le abbiamo dichiarate come funzioni *protected*, ovvero accessibili solo alle classi derivate (andava bene anche definirle come *private*). Allora *add* prende in ingresso 2 parametri che sono l'elemento da aggiungere e dove aggiungere l'elemento e quindi in particolare riceve in ingresso un nodo e l'elemento da inserire. Il *remove* analogamente rimuove un nodo dalla lista.

Vediamo ora l'implementazione della classe (che deve essere inclusa nell'header perché il main non può richiamare solo l'header proprio in virtù del fatto che i templates implementati nel .cpp si specializzano solo in seguito alla loro invocazione nel main).

Quindi se includiamo il .cpp nell'header il problema si risolve perché quando il main richiama il .h richiama pure il .cpp.

```
1 //DLinkedList.cpp
2 #include<iostream>
3 #include<sstream>
4 template <typename T>
5 DLinkedList<T>::DLinkedList() {           // constructor
6     header = new DNode<T>;                  // create sentinels
7     trailer = new DNode<T>;
8     header->next = trailer;                 // have them point to each other
9     header->prev = nullptr;
10    trailer->prev = header;
11    trailer->next = nullptr;
12 }
13
14 template <typename T>
15 DLinkedList<T>::~DLinkedList() {          // destructor
16     while (!empty()) removeFront();          // remove all but sentinels
17     delete header;                         // remove the sentinels
18     delete trailer;
19 }
20
21 template <typename T>
22 bool DLinkedList<T>::empty() const {        // is list empty
23     return (header->next == trailer);
24 }
25
26 template <typename T>
27 const T& DLinkedList<T>::front() const {   // get front element
28     return header->next->elem;
29 }
30
31 template <typename T>
32 const T& DLinkedList<T>::back() const {    // get back element
33     return trailer->prev->elem;
34 }
```

Il costruttore di una double linked list viene creato in modo da generare una lista vuota. Tuttavia la lista vuota contiene almeno 2 nodi, ovvero i sentinella, che sono *header* e *trailer* e quindi istanziamo questi nodi in modo tale che il *next* del primo punti al secondo ed il *prev* del secondo punti al primo. Quindi il costruttore alloca 2 puntatori a DNode, uno per istanziare la variabile membro privata header e l'altro per trailer. Ovviamente poi il *prev* di *header* punterà a *nullptr*, il *next* di *trailer* punterà a *nullptr*.

La funzione *empty* verificherà a questo punto semplicemente se il *next* della testa, ovvero di *header*, è uguale alla coda e quindi a *trailer*; in tal caso i nodi testa e coda sono consecutivi e quindi la lista è vuota.

Il distruttore rimuove tutti i nodi e pure quelli header e trailer.

Ci sono poi le 2 funzioni get per ottenere il primo elemento, ovvero quello dopo la testa, e l'ultimo elemento, ovvero prima della coda. (i nodi sentinella non conservano alcuna informazione).

Ora vediamo le funzioni *addFront* e *addBack* per aggiungere elementi alla testa e alla coda della lista (sono dichiarate *protected*) e la funzione *add* per aggiungere elementi nella lista in una posizione qualunque.

```

55 template <typename T>
56 void DLinkedList<T>::add(DNode<T>* v, const T& e) {
57     DNode<T>* u = new DNode<T>; u->elem = e;           // create a new node for e
58
59     u->next = v;                                         // link u in between v
60     u->prev = v->prev;                                    // ...and v->prev
61     v->prev->next = u;
62     v->prev = u;
63 }
64
65 template <typename T>
66 void DLinkedList<T>::addFront(const T& e){          // add to front of list
67     add(header->next, e);
68 }
69 template <typename T>
70 void DLinkedList<T>::addBack(const T& e) {          // add to back of list
71     add(trailer, e);
72 }
```

add riceve in ingresso il puntatore al nodo che sta subito dopo l'elemento che vogliamo inserire e l'elemento da inserire. Rispetto all'esempio, la funzione *add* aggiunge il nodo *u* prima del nodo *v* e ci mette dentro *e* come elemento. Il nuovo nodo *u* dovrà puntare a *v*, e quindi:

$u \rightarrow next = v$,

mentre il *prev* di *u* punterà al *prev* di *v* (perché *u* sta andando a prendere il posto di *v*):

$u \rightarrow prev = v \rightarrow prev$,

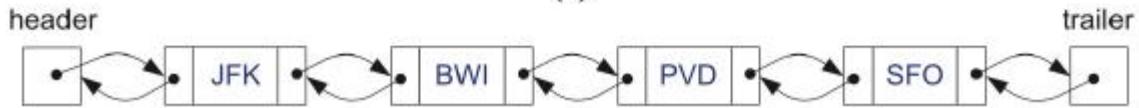
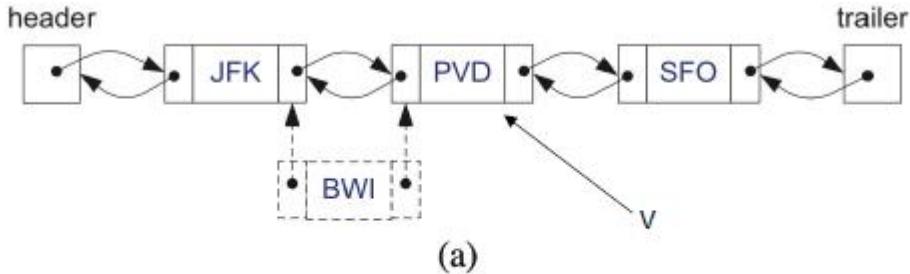
ed *u* dovrà essere puntato dal *next* del *prev* di *v*, e quindi:

$v \rightarrow prev \rightarrow next = u$.

Infine il *prev* di *v* deve puntare al nodo appena inserito, ovvero *u*:

$v \rightarrow prev = u$.

(Oss: Se la lista è vuota, alla funzione *add* passeremo *trailer*).



Se scambiamo riga 61 e riga 62 succede non riesce bene il tutto perché prima riscriviamo il *prev* di *v* e non avremmo più avuto accesso al *next* del *prev* di *v*. Quindi sono operazioni semplici ma vanno fatte con attenzione.

La cosa bella è che utilizzando le sentinelle questo codice funziona anche se la lista è vuota, quindi non dobbiamo fare nessun check! Infatti se la lista è vuota allora *v* è il *trailer* poiché aggiungiamo l'elemento tra testa e coda spostando la coda, praticamente.

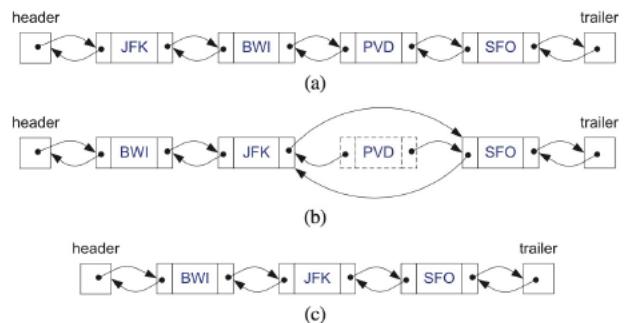
Le funzioni *addFront* e *addBack* sono semplici, ed anche queste funzioni “funzionano” se la lista è vuota. Si basano sulla funzione *add* a cui passiamo in *addFront* il *next* di *header*, ovvero il primo nodo (non sentinella), che va “spostato” per fare spazio a quello da inserire, e l’elemento del nodo da inserire. Se la lista fosse vuota il *next* di *header* sarebbe il *trailer* e quindi funziona uguale; alla funzione *add* di *addBack* passiamo invece *trailer*, così da aggiungere prima del *trailer* il nodo, e sempre l’elemento del nodo da inserire ovviamente.

Tutte queste sono operazioni che richiedono un tempo costante.

Vediamo ora la rimozione.

```

36 template <typename T>
37 void DLinkedList<T>::remove(DNode<T>* v) {           // remove node v
38     if(empty()){
39         throw std::runtime_error("The list is empty");
40     }
41     DNode<T>* u = v->prev;                           // predecessor
42     DNode<T>* w = v->next;                           // successor
43     u->next = w;                                     // unlink v from list
44     w->prev = u;
45     delete v;
46 }
47
48 template <typename T>
49 void DLinkedList<T>::removeFront() {                  // remove from front
50     remove(header->next);
51 }
52
53 template <typename T>
54 void DLinkedList<T>::removeBack(){
55     remove(trailer->prev);
56 }
```



Per rimuovere un elemento passo alla funzione *remove* un puntatore al nodo da rimuovere (nell'esempio di sopra è quello che contiene PVD). Ovviamente da una lista vuota non posso rimuovere niente, quindi viene fatto un check che la lista non sia vuota. Per rimuovere il nodo devo fare in modo che quello precedente punti a quello successivo del nodo in questione, e viceversa. Allora creo nella funzione un puntatore *u* al nodo precedente ed uno *w* a quello successivo rispetto a quello da rimuovere *v*: il next di *u* deve puntare all'elemento successivo a *v*, ovvero a *w*; invece il *prev* di *w* deve puntare all'elemento precedente a *v*, ovvero *u*. Dopodiché viene eliminato *v* per restituire al SO la memoria.

La rimozione in testa ed in coda sono semplici; si rifanno alla funzione *remove* e la rimozione di un elemento alla testa invoca il *remove* passando il *next* di *header*, quella di un elemento alla coda invoca il *remove* passando il *prev* di *trailer*.

Abbiamo scritto poi anche una funzione *toString()* per scorrere la lista e visualizzarne gli elementi.

```
74 template <typename T>
75 std::string DLinkedList<T>::toString() {
76     std::ostringstream out;
77     DNode<T>* u = header->next;
78     out << "Header <--> ";
79     while(u!=trailer) {
80         out << u->elem << " <--> ";
81         u = u->next;
82     }
83     out << "Trailer";
84     return out.str();
85 }
```

Molto semplicemente creiamo una variabile Nodo temporanea, a cui facciamo assumere il primo valore della lista, con cui scorriamo la lista fin quando non raggiunge il trailer.

A questo punto testiamo il programma.

```

1 #include <iostream>
2 #include "DLinkedList.h"
3
4 int main() {
5     DLinkedList<std::string> L;
6     L.addFront("ORD");
7     L.addFront("JFK");
8     L.addFront("SCO");
9     L.addBack("SFO");
10    std::cout << L.toString() << std::endl;
11 }

```

```

1 #Makefile for tesDLinkedList
2 CC=g++
3 CFLAGS=-std=c++11
4 DEPS = DNode.h DLinkedList.h DLinkedList.cpp
5 OBJ = testDLinkedList.o
6
7 %.o: %.cpp $(DEPS)
8     $(CC) -c -o $@ $< $(CFLAGS)
9
10 testDLinkedList: $(OBJ)
11     $(CC) -o $@ $^ $(CFLAGS)

```

```

8 $ make
9 g++ -c -o testDLinkedList.o testDLinkedList.cpp -std=c++11
10 g++ -o testDLinkedList testDLinkedList.o -std=c++11
11 $ ./testDLinkedList
12 Header <--> SCO <--> JFK <--> ORD <--> SFO <--> Trailer
13 $

```

Esercizio:

Exercise

- Assume that the type T has a comparison operator “<”
- Create an ordered doubly linked list:
 - Extend the DLinkedList class with a member function `insertOrder(const T& e)` that insert the element e in the correct order.

L'esercizio chiede di estendere la DLinkedList con una funzione membro `insertOrder` che riceve in ingresso un elemento da inserire nell'ordine corretto. Ciò è possibile solo assumendo che gli elementi della lista possano essere confrontati fra di loro → va riscritto l'operatore “<”. Allora ad esempio possiamo scorrere i nodi fino a quando non troviamo un nodo che contiene un elemento maggiore di quello da inserire.

```

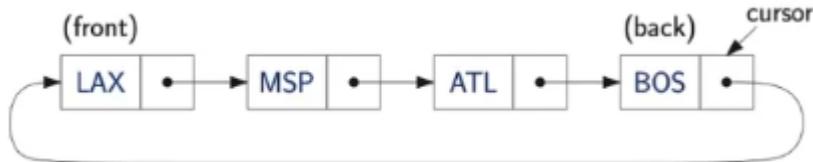
77 template <typename T>
78 void addOrder (const T & e) {
79     DNode<T>* u = header->next;
80     while(u!= trailer) {
81         if (e < u->elem ){
82             break;
83         }
84         u = u->next;
85     }
86     add(u,e);
87 }

```

(riscritto tra addBack e toString)

LISTE LINKATE CIRCOLARI (CIRCULAR LINKED LISTS)

Hanno come particolarità che un puntatore è un cursore e la lista non termina con un puntatore nullo ma dal primo puntatore della lista. La posizione puntata dal cursore si chiama *back*, la posizione successiva si chiama *front*; è facile intuire il perché: se si taglia l'ultimo link l'ultimo elemento diventa il trailer, il primo diventa l'header.



Vediamo allora com'è fatta la classe di un nodo di una lista circolare e quella della lista circolare vera e propria.

```
1 //CNode.h
2
3 template <typename T> class CircleList; //forward declaration
4 template <typename T> // element type
5 class CNode { // circularly linked list node
6 private:
7     T elem; // linked list element value
8     CNode* next; // next item in the list
9     friend class CircleList<T>; // provide CircleList access
10};
```

Il nodo contiene un elemento ed un link al nodo successivo. Viene poi dichiarata una classe amica che è quella della lista circolare e che vediamo qui di seguito:

```

1 //CLinkedList.h
2 #ifndef CLINKED_LIST_H
3 #define CLINKED_LIST_H
4
5 #include <iostream>
6 #include "CNode.h"
7 template <typename T>
8 class CircleList {           // a circularly linked list
9 public:
10    CircleList();            // constructor
11    ~CircleList();           // destructor
12    bool empty() const;     // is list empty?
13    const T& front() const;  // element at cursor
14    const T& back() const;   // element following cursor
15    void advance();          // advance cursor
16    void add(const T& e);   // add after cursor
17    void remove();           // remove node after cursor
18    std::string toString();
19 private:
20    CNode<T>* cursor;       // the cursor
21 };
22
23 #include "CLinkedList.cpp"
24
25 #endif

```

Sappiamo che abbiamo navigato tutta la lista quando siamo partiti dal nodo cursore e siamo tornati a quello stesso nodo.

Una funzione che mette a disposizione la lista circolare è quella dell'avanzamento del cursore (*advance*); ovviamente avendo solo il *next*, il cursore può solo essere spostato in avanti. Sono poi implementati il costruttore, il distruttore, *empty*, *front* e *back* (il *front* è il *next* del cursore, il *back* è esattamente il nodo puntato dal cursore), *add* e *remove* (effettuate dopo il cursore).

L'unica variabile membro che utilizza questa classe è un puntatore al cursore.

Vediamo l'implementazione delle varie funzioni.

```

1 #include <iostream>
2 #include <sstream>
3 template <typename T>
4 CircleList<T>::CircleList()           // constructor
5   : cursor{nullptr} { }
6
7 template <typename T>
8 CircleList<T>::~CircleList()          // destructor
9   { while (!empty()) remove(); }
10
11 template <typename T>
12 bool CircleList<T>::empty() const    // is list empty?
13   { return cursor == nullptr; }
14
15 template <typename T>
16 const T& CircleList<T>::back() const // element at cursor
17   { return cursor->elem; }
18
19 template <typename T>
20 const T& CircleList<T>::front() const // element following cur
21   { return cursor->next->elem; }
22
23 template <typename T>
24 void CircleList<T>::advance()        // advance cursor
25   { cursor = cursor->next; }
26
27 template <typename T>
28 void CircleList<T>::add(const T& e) { // add after cursor
29   CNode<T>* v = new CNode<T>;         // create a new node
30   v->elem = e;
31   if (cursor == nullptr) {             // list is empty?
32     v->next = v;                   // v points to itself
33     cursor = v;                   // cursor points to v
34   }
35   else {                           // list is nonempty?
36     v->next = cursor->next;       // link in v after cursor
37     cursor->next = v;
38   }
39 }
40
41 template <typename T>
42 void CircleList<T>::remove() {        // remove node after cursor
43   CNode<T>* old = cursor->next;      // the node being removed
44   if (old == cursor) {                // removing the only node?
45     cursor = nullptr;                // list is now empty
46   }
47   else {                            // link out the old node
48     cursor->next = old->next;       // delete the old node
49   }

```

```

51 | template <typename T>
52 | std::string CircleList<T>::toString(){
53 |     std::ostringstream out;
54 |     CNode<T>* u = cursor;
55 |     out << "Cursor --> ";
56 |     out << u->elem << " --> ";
57 |     u = u->next;
58 |     while (u != cursor){
59 |         out << u->elem << " --> ";
60 |         u = u->next;
61 |     }
62 |     out << "Cursor";
63 |     return out.str();
64 |

```

In realtà il client dovrebbe fare un check che la lista non sia vuota per tutte le funzioni di sopra. Questa struttura dati la utilizzeremo per implementare delle code.

Scriviamo poi un programma per testare la lista circolare:

```

1 #include <iostream>
2 #include "CLinkedList.h"
3 using std::string;
4 int main(){
5     CircleList<std::string> playList;
6     //tunes I should learn
7     std::cout << "Hey, this is our playList" << std::endl;
8     playList.add("Autumn leaves");
9     playList.add("Mo' Better Blues");
10    playList.add("Waltz for Debbie");
11    playList.add("So What");
12    playList.advance();
13    playList.advance();
14    std::cout << playList.toString() << std::endl;
15    playList.remove();
16    playList.add("Caravan");
17    std::cout << playList.toString() << std::endl;
18    return 0;
19 }

$ g++ testClinkedLists.cpp -o testClinkedLists -std=c++11
$ ./testClinkedLists
Hey, this is our playList
Cursor --> Waltz for Debbie --> Mo' Better Blues --> Autumn leaves --> So What --> Cursor
Cursor --> Waltz for Debbie --> Caravan --> Autumn leaves --> So What --> Cursor

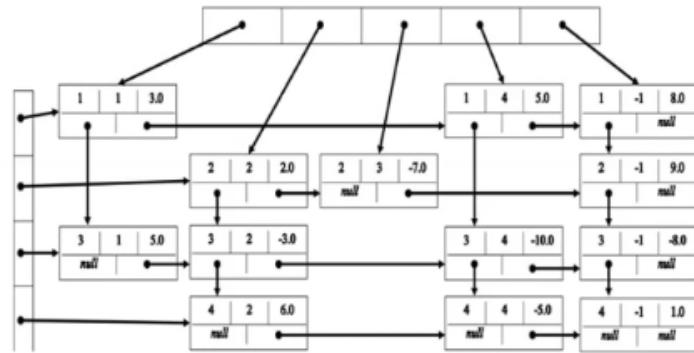
```

Nell'esempio c'è una playlist che continua a girare. In generale ci conviene usare una lista circolare quando dobbiamo andare avanti fino a ripartire dall'inizio.

Esercizio IMPORTANTE da fare (richiede 1 giornata di lavoro/1 e mezzo).

Progetto di Programmazione: Matrici Sparse

- Una matrice sparsa è una matrice in cui la gran parte dei valori è uguale a zero e solo poche posizioni sono occupate da valori diversi da zero.
- Le matrici sparse sono molto comuni in diversi ambiti scientifici
- Piuttosto che utilizzare un array bidimensionale per una matrice sparsa, una soluzione alternativa è quella di utilizzare delle liste a puntatori per le righe e le colonne.
- Un nodo conserva le coordinate ed il valore corrispondente più un puntatore al prossimo elemento lungo la riga e un puntatore al prossimo elemento lungo la colonna
- Scrivere un programma C++ per la rappresentazione di una matrice sparsa NxM.
- Il programma legge da un file testo che contiene nella prima riga le dimensioni NxM. Nelle rimanenti righe contiene le tripette (i,j,A[i,j])
- Come si potrebbe la somma di matrici sparse?



Questo esercizio è una classica applicazione delle liste a puntatori. Quando una matrice è molto sparsa e genericamente sparsa (nel senso che non so dove si concentrano gli zeri) allora rappresentiamo righe e colonne come delle liste a puntatori: ogni nodo contiene il valore, l'indice di riga e l'indice di colonna. E quindi ogni nodo ha 2 puntatori: uno che si muove in orizzontale ed uno che va in verticale. Oltre a questo ci servono 2 array: 1 array dei puntatori alle righe ed 1 array dei puntatori alle colonne.

Fare poi anche quest'esercizio.

Esercise (To DO!): Linked List reversal

- Reverse a Linked List.
- To reverse a linked list L
 - Create a new list T
 - Repeatedly
 - Extract the first element of L
 - Copy this element to the front of T
- You can see the pseudocode:

```
void listReverse(DLinkedList& L) {          // reverse a list
    DLinkedList T;                          // temporary list
    while (!L.empty()) {                   // reverse L into T
        string s = L.front();  L.removeFront();
        T.addFront(s);
    }
    while (!T.empty()) {                   // copy T back to L
        string s = T.front();  T.removeFront();
        L.addBack(s);
    }
}
```

LEZIONE 24

ADT (Abstract Data Type)

Abbiamo visto la scorsa lezione come rappresentare una collezione di oggetti con una struttura dati di tipo “linked list”, con tutte le varianti (liste linkate, liste doppiamente linkate, liste circolari). Questa struttura dati è una struttura dati concreta, ovvero che permette di rappresentare una collezione di informazioni e di aggiungere o rimuovere dinamicamente degli oggetti da questa collezione. Ciò lo chiamiamo in C++ un **container**, ovvero un insieme dinamico che cresce e decresce durante l'esecuzione del programma. Allora cerchiamo di pensare in maniera un po' più astratta ad una generica struttura dati che mette a disposizione un insieme di servizi e che chiameremo **ADT (Abstract Data Type)**. È compito dell'implementazione capire come le funzioni che forniscono i servizi della classe astratta debbano essere realizzate mediante una struttura dati concreta, come ad esempio un array, un vettore, una lista concatenata, etc... Quindi, aumentando l'astrazione, cerchiamo di capire quali sono le principali strutture dati astratte per poter poi fare la scelta migliore da bravi programmati, separando sempre di più l'interfaccia dall'implementazione.

Nel C++ c'è una libreria detta **STL (Standard Template Library)**, in cui sono definiti molti container, ovvero insiemi dinamici, con varie proprietà e funzioni che poi vanno implementate in maniera differente in base alla struttura dati concreta che si adotta. Noi vogliamo diventare BRAVI programmati e non sfruttare facilmente tutti questi container messi a disposizione da altri bravi programmati di C++. Allora adesso ci proponiamo di conoscere bene ed approfondire varie strutture dati:

- STACK

- CODE (=Queues)

- CODE DOPPIE (=Dequeues)

- MAPPE, come la tabella hash; la hash table è una struttura dati quasi magica! Le diamo una chiave e vogliamo accedere ad un elemento che ha una particolare chiave; la tabella mi restituisce in maniera molto efficiente l'oggetto con la particolare chiave → ricerca efficiente (quasi in tempo O(1)) fatta “per chiave”. In particolare per fare ciò questa tabella crea una mappa dallo spazio delle chiavi allo spazio delle posizioni nella tabella. Quindi questa tabella mette a disposizione un servizio di ricerca per chiave; poi come viene implementato è un'altra storia e l'efficienza dipende anche da questo. Ci sono poi le mappe ordinate che mettono a disposizione sia la ricerca per chiave che l'ordinamento; ed anche queste possono essere implementate in diverse maniere, tra cui quella della lista linkata a puntatori. Ovviamente però le ricerche sono lineari. Esistono invece delle strutture dati concrete come i binary search trees che permettono di mantenere la struttura dati con ricerca per chiave ed ordinamento in

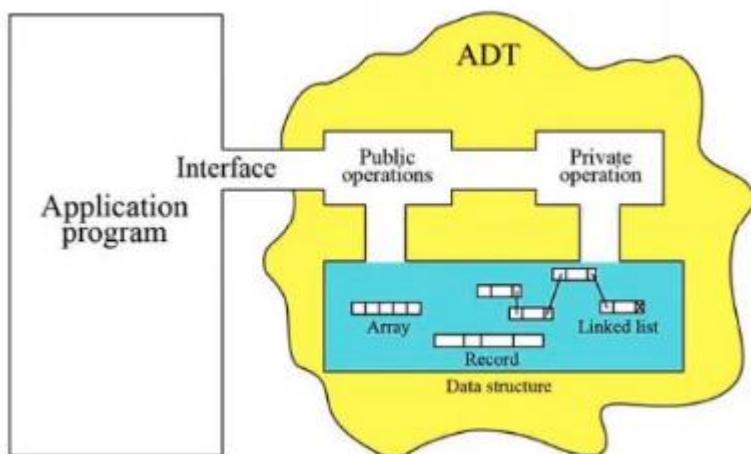
tempi (medi) logaritmici. Purtroppo però l'albero binario è bellissimo quando è bilanciato, meno bello quando è sbilanciato (proprio come la ricerca binaria che abbiamo già visto). Quindi è preferibile usare alberi binari bilanciati in cui anche il caso peggiore è logaritmico. Non credo le vedremo tutte queste cose.

La *STL* è una collezione di classi utili per strutture dati comuni. In aggiunta alla classe *string* fornisce strutture dati anche per i seguenti containers (insiemi dinamici di oggetti) implementati come class templates:

stack
queue
deque
vector
list
priority queue
set
map

Queste sono tutte strutture dati astratte.

Un tipo di dato astratto (ADT) lo possiamo immaginare come un insieme di oggetti a cui si può accedere mediante delle operazioni pubbliche implementate da strutture dati concrete. Le *public operations* sono quelle che usa il client per invocare dei particolari servizi al tipo di dato astratto.



Un ADT specifica:

- i dati da rappresentare;
- le operazioni sui dati;
- le condizioni di errore associate alle operazioni.

Example: ADT modeling a simple stock trading system

- The data stored are buy/sell orders
- The operations supported are
 - order **buy**(stock, shares, price)
 - order **sell**(stock, shares, price)
 - void **cancel**(order)
- Error conditions:
 - Buy/sell a nonexistent stock
 - Cancel a nonexistent order

STACK

Lo stack è utilizzato da moltissimi algoritmi e come già sappiamo ad ogni chiamata di una funzione. La struttura dati astratta “stack” colleziona un insieme di oggetti e mette a disposizione:

- operazioni di inserzione (**push**),
- operazioni di prelievo (**pop**),

utilizzando una *policy LIFO* (Last in – First out): l’elemento da estrarre è l’ultimo inserito, proprio come in una pila. PUSH inserisce un elemento sulla testa dello stack, POP preleva l’elemento sulla testa dello stack. Ci sono poi delle operazioni ausiliarie:

```
object top(): returns the last inserted  
element without removing it  
integer size(): returns the number of elements  
stored  
boolean empty(): indicates whether no elements  
are stored
```

Ovviamente potremmo conoscere l’elemento inserito 4 push fa, ma sarebbe inefficiente perché dovremmo fare 4 pop e 4 push; quindi in generale questa struttura dati astratta non è ideale per accedere agli elementi in base all’ordine di inserimento. Invece ci conviene utilizzare lo stack quando il nostro algoritmo richiede il supporto alle funzioni di sopra. Però, appunto, per realizzare un supporto all’ordinamento ed alla ricerca dovremmo utilizzare delle strutture dati astratte più complesse.

La STL fornisce l’implementazione di uno stack basata su una struttura dati concreta vector. Per usare le funzionalità dello stack basta includere <stack> ed usare la parola “stack” appartenente alla libreria standard.

```
1 #include <stack>
2 using std::stack; // make stack accessible
3
4 stack<int> myStack;
5
```

Le funzioni che mette a disposizione lo stack sono (quelle viste prima):

- `size()`: Return the number of elements in the stack.
- `empty()`: Return true if the stack is empty and false otherwise.
- `push(e)`: Push e onto the top of the stack.
- `pop()`: Pop the element at the top of the stack.
- `top()`: Return a reference to the element at the top of the stack.

Come possiamo implementare uno stack?

Possiamo utilizzare un array, in cui memorizziamo gli oggetti o i riferimenti agli oggetti da memorizzare, e poi una variabile *t* (top) per capire qual è la prossima casella libera in cui fare un push.

Allora l'interfaccia potrebbe essere fatta così: definiamo una classe di nome Stack mediante un template in cui si mettono a disposizione quelle 5 funzioni.

```
1 template <typename E>
2 class Stack {
3 public:
4     int size() const;
5     bool empty() const;
6     const E& top() const;
7     void push(const E& e);
8     void pop();
9 };
```

Questa è l'interfaccia generica dell'ADT; questa la implementiamo poi con una specifica struttura dati concreta come, appunto, un ARRAY di oggetti di tipo E.

```

1 //Astack.h
2 #ifndef ASTACK_H
3 #define ASTACK_H
4 #include <iostream>
5 #include <stdexcept>
6 template <typename E>
7 class ArrayStack {
8     static const int DEF_CAPACITY{10};      // default stack capacity
9 public:
10    ArrayStack(int cap = DEF_CAPACITY);    // constructor from capacity
11    int size() const;                      // number of items in the stack
12    bool empty() const;                    // is the stack empty?
13    const E& top() const;                  // get the top element
14    void push(const E& e);                // push element onto stack
15    void pop();                          // pop the stack
16    // ...housekeeping functions omitted
17 private:
18    E* S;                                // array of stack elements
19    int capacity;                         // stack capacity
20    int t;                                // index of the top of the stack
21 };

```

Utilizziamo come variabili private un array di oggetti di tipo generico E, un intero per la capacità massima dello stack ed una variabile intera *t* che indica il top dello stack.

Abbiamo poi definito un costruttore ed un distruttore. Vediamo l'implementazione delle funzioni (che abbiamo fatto nel .h, uno dei modi per compilare con il client).

```

23 template <typename E>                                // push element onto the stack
24 void ArrayStack<E>::push(const E& e) {
25     if (size() == capacity) throw std::runtime_error("StackFull");
26     S[++t] = e;
27 }
28 template <typename E>                                // pop the stack
29 void ArrayStack<E>::pop() {
30     if (empty()) throw std::runtime_error("StackFull");
31     --t;
32 }
33
34 template <typename E>
35 ArrayStack<E>::ArrayStack(int cap)
36     : S{new E[cap]}, capacity{cap}, t{-1} {}           // constructor from capacity
37
38 template <typename E>
39 int ArrayStack<E>::size() const
40 { return (t + 1); }                                    // number of items in the stack
41
42 template <typename E>
43 bool ArrayStack<E>::empty() const
44 { return (t < 0); }                                    // is the stack empty?
45
46 template <typename E>                                // return top of stack
47 const E& ArrayStack<E>::top() const {
48     if (empty()) throw std::runtime_error("Top of empty stack");
49     return S[t];
50 }

```

Il costruttore riceve in ingresso un intero che è la capacità; di default è posta a 10 se non specificata. Inizializza poi le variabili membro con un vettore di dimensione pari alla capacità in ingresso, la capacità in ingresso e *t* è inizializzata a -1; questo perché

rappresenta la posizione dell'ultimo oggetto inserito e quindi il prossimo oggetto lo inseriremo in posizione 0. Serve per inizializzare lo stack come vuoto.

PERFORMANCE & LIMITATIONS

Performances: Se il numero di elementi nello stack è n allora lo spazio usato è $O(n)$; ogni operazione è $O(1)$

Limitations: la dimensione massima dello stack va definita a priori ed è limitata ovviamente

La standard library utilizza un vector per implementare lo stack; quindi è comunque simile a ciò che abbiamo visto con gli array, solo che varia dinamicamente di capacità al variare degli elementi memorizzati.

Ora proviamo ad implementare lo stack usando le **LISTE LINKATE A PUNTATORI**.

```
1 //LStack.h
2
3 #ifndef LSTACK_H
4 #define LSTACK_H
5 #include "LinkedList.h"
6 #include <stdexcept>
7 template <typename E>
8 class LinkedStack {      // stack as a linked list
9 public:
10    LinkedStack();          // constructor
11    int size() const;       // number of items in the stack
12    bool empty() const;     // is the stack empty?
13    const E& top();         // the top element
14    void push(const E& e);  // push element onto stack
15    void pop();             // pop the stack
16 private:                // member data
17    LinkedList<E> s;       // linked list of elements
18    int n;                  // number of elements
19 };
```

La lista è ideale per implementare una strategia di tipo LIFO. Quindi la classe che implementa lo stack con una lista linkata avrà come variabili membro una variabile *LinkedList* con elementi di tipo *E* ed il numero di elementi. Come in tutto c'è un trade-off, in questo casto tra memoria occupata e numero di operazioni. La funzione *size()* la potremmo implementare tranquillamente scorrendo tutta la lista, con una complessità $O(n)$; con una variabile *n* che indica il numero di elementi la complessità diventa $O(1)$, però il prezzo lo si paga allocando una variabile in più in memoria. Quindi c'è un trade-off tra efficienza e memoria.

La classe *LinkedList* è sempre quella costruita nella lezione scorsa.

```
21 template <typename E>
22 LinkedStack<E>::LinkedStack()
23     // constructor
24     : S(), n{0} { }
25
26 template <typename E>
27 int LinkedStack<E>::size() const {
28     // number of items in the stack
29     return n;
30 }
31
32 template <typename E>
33 bool LinkedStack<E>::empty() const {
34     // is the stack empty?
35     return n == 0;
36 }

38 template <typename E>
39 const E& LinkedStack<E>::top() {
40     // returns the element at the top
41     if (empty())
42         throw std::runtime_error("Top of empty stack");
43     return S.front();
44 }
45
46 template <typename E>
47 void LinkedStack<E>::push(const E& e) {
48     // push element onto stack
49     ++n;
50     S.addFront(e);
51 }
52 template <typename E>
53 void LinkedStack<E>::pop() {
54     // pop the stack
55     if (empty())
56         throw std::runtime_error("Pop from empty stack");
57     --n;
58     S.removeFront();
59 }
```

Testiamo lo stack:

```

1 #include <iostream>
2 #include "LStack.h"
3
4 int main(){
5     LinkedStack<std::string> myStack;
6     myStack.push("one");
7     myStack.push("two");
8     myStack.push("three");
9     std::cout << myStack.top() << std::endl;
10    myStack.pop();
11    std::cout << myStack.top() << std::endl;
12    myStack.pop();
13    std::cout << myStack.top() << std::endl;
14    myStack.pop();
15    return 0;
16 }

```

```

$ g++ testLStack.cpp -o testLStack -std=c++11
$ ./testLStack
three
two
one

```

Il client definisce una variabile di tipo *LinkedStack* e di nome *myStack*. Dopodiché fa delle operazioni di push, visualizza l'elemento in cima e fa operazioni di pop.

Ma che ce ne frega dello stack? Mica lo usiamo?

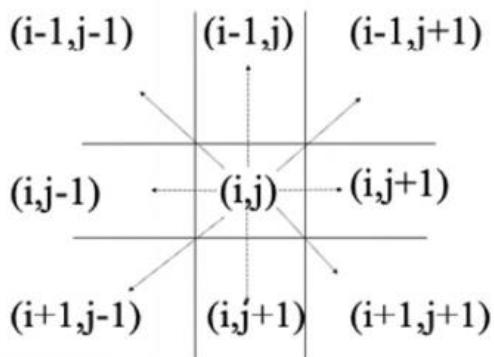
In realtà lo usiamo sempre, ogni volta che invochiamo una funzione e soprattutto quando invochiamo una funzione ricorsiva. Vediamo l'esempio del LABIRINTO (*maze*) che va risolto per forza con lo stack.

Problema

- Sviluppare un programma C++ per risolvere un labirinto



Le posizioni libere le rappresentiamo con 0, i muri con 1. Dobbiamo trovare un percorso dalla posizione (0,0) alla posizione (n,n). Quest'esempio è significativo per quanto riguarda il *backtracking*: fare una scelta, fallire, tornare al passo precedente. Allora lo stack serve proprio per memorizzare l'ultima mossa.



Dir	Mossa[dir].x	Mossa[dir].y
0	-1	0
1	-1	1
2	0	1
3	1	1
4	1	0
5	1	-1
6	0	-1
7	-1	-1

Il nostro dominio ha 8 mosse. Da una casella possiamo spostarci nelle 8 caselle adiacenti. Ad esempio il primo passo può essere andare a destra: se è libero allora mi sposto, a questo punto mi sposto a destra di nuovo se libero, se non è libero allora mi sposto in basso a destra, e così via. Insomma quando trovo un muro devo tornare indietro all'ultima scelta effettuata. Dunque, poiché per questo problema non c'è una strategia più intelligente di esplorare tutte le possibilità, cerco di esplorarle in maniera *depth*, andando in profondità e non appena si trova un muro tornando indietro all'ultima scelta per effettuarne un'altra. Ovviamente la mossa deve essere lecita, quindi dobbiamo fare in modo da non uscire dal rettangolo di gioco.

Lo stack quindi mi serve per memorizzare l'ultima mossa fatta. Quindi per questo esercizio lo stack va usato necessariamente, in maniera diretta o indiretta.

La procedura da seguire è circa questa:

```

Inizializza uno stack con le coordinate di entrata e direzione nord
while (stack non vuoto)
    (posizione,dir) := pop dallo stack
    while (esistono altre mosse dalla posizione corrente)
        posizione_succ := posizione + (Mossa[dir].x,Mossa[dir].y)
        if (posizione_succ == USCITA)
            then SUCESSO
        else if (posizione_succ ammissibile)
            then marca posizione_succ
            dir := dir + 1
            push nello stack (posizione,dir)
            posizione := posizione_succ
            dir = 0
        else dir = dir + 1
    
```

Nello stack ci sarà $(i, j, direzione)$, ovvero la posizione (i, j) e la direzione percorsa al passo precedente per giungere a quella posizione.

Questo è il modo esplicito di usare lo stack per risolvere il gioco. Il modo implicito sarebbe quello di fare delle chiamate ricorsive allo stesso algoritmo: posso definire una funzione ricorsiva che, per i che va da 0 a 7, se il prossimo passo è ammissibile in quella direzione allora chiama *maze* non a partire dalla posizione attuale ma da quella posizione successiva. In questo caso usiamo implicitamente lo stack con varie chiamate ricorsive.

Altro esempio dell'uso di uno stack: la valutazione di un'espressione.

- Ex.: Evaluating Arithmetic Expressions
 $14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$
- Operator precedence
 - $*$ has precedence over $+$ / $-$
- Associativity
 - operators of the same precedence group
 - evaluated from left to right
 - Example: $x - y + z$ evaluated as $(x - y) + z$
- Idea:
 - push each operator on the stack, but first pop and perform higher and equal precedence operations.

Questo lo vediamo solo teoricamente, come applicazione dello stack. Il modo in cui viene valutata un'espressione è basato sull'utilizzo di un doppio stack: uno per gli operandi ed uno per gli operatori. Per valutare un'espressione si usa un metodo che si chiama *notazione polacca postfissa*, ovvero una notazione in cui si riscrive l'espressione in modo da avere (operando1, operando2, operatore) ed in maniera tale che le operazioni, una volta che si prelevano dallo stack, possano essere realizzate nell'ordine definito dalla priorità degli operatori tramite le parentesi che non fanno altro che alterare la priorità. L'algoritmo di base per valutare un'espressione è stato inventato dallo stesso inventore dell'algoritmo di Dijkstra ed utilizza il doppio stack in modo da pushare gli operandi in uno stack e gli operatori nell'altro stack facendo in maniera tale da mettere al top dello stack gli operatori a più alta priorità.

- Two stacks
 - opStk holds operators
 - valStk holds values

Algorithm doOp()

```

x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )

```

Algorithm repeatOps(refOp):

```

while ( valStk.size() > 1 ∧ prec(refOp) ≤ prec(opStk.top()) )
    doOp()

```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

while there's another token z

if isNumber(z) **then**

 valStk.push(z)

else

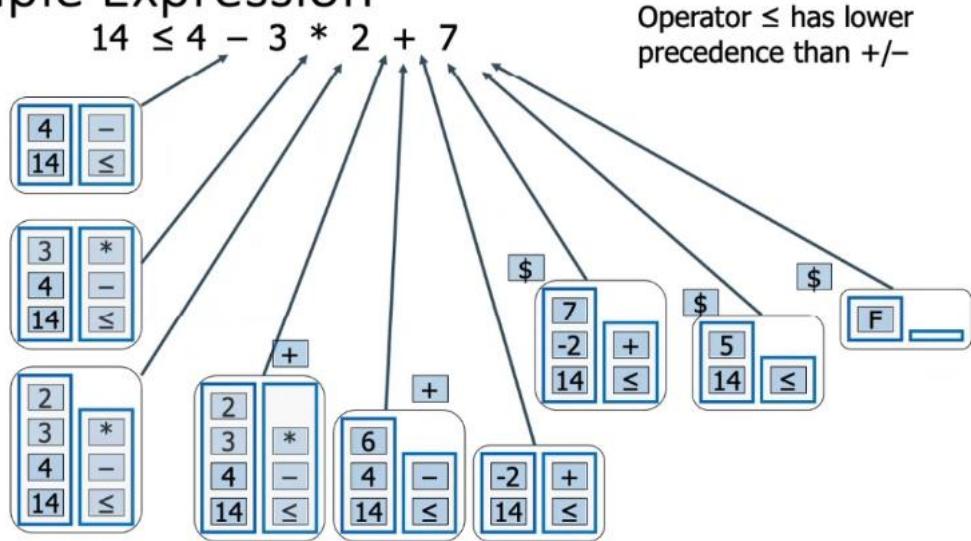
 repeatOps(z);

 opStk.push(z)

 repeatOps(\$);

return valStk.top()

Algorithm on an Example Expression



(leggere l'ordine delle operazioni di push e pop sugli stack da sinistra verso destra)

Quello che stiamo facendo è applicare implicitamente la notazione polacca postfissa e l'algoritmo si chiama Shunting-yard, inventato da Dijkstra.

Altro esempio, utilizzato nella finanza: valutare il valore di una quotazione che sta avendo un picco. Vogliamo sapere in sostanza se la quotazione sta raggiungendo un picco in base al numero di giorni consecutivi in cui l'azione è aumentata. E quindi si richiede di calcolare quello che è detto *span*: se il prezzo di un'azione in un certo giorno i è il massimo tra un certo numero di giorni consecutivi. In particolare lo span è il numero di giorni consecutivi in cui la quotazione cresce.

- [The stock span problem](#) is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.
- The span $S[i]$ of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.
- For example, if an array of 7 days prices is given as $\{100, 80, 60, 70, 60, 75, 85\}$, then the span values are: $\{1, 1, 1, 2, 1, 4, 6\}$

L'algoritmo stupido ha complessità $O(n^2)$, perché va indietro fino all'inizio (nel caso peggiore) per vedere quanti giorni consecutivi precedenti l'azione ha avuto un valore minore rispetto a quello attuale.

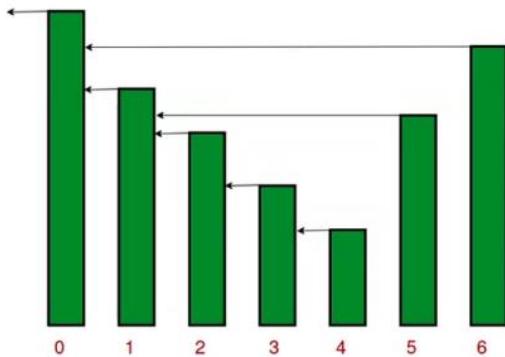
```

1 void calculateSpan(int price[], int n, int S[])
2 {
3     // Span value of first day is always 1
4     S[0] = 1;
5
6     // Calculate span value of remaining days
7     // by linearly checking previous days
8     for (int i = 1; i < n; i++)
9     {
10         S[i] = 1; // Initialize span value
11
12         // Traverse left while the next element
13         // on left is smaller than price[i]
14         for (int j = i - 1; (j >= 0) &&
15             (price[i] >= price[j]); j--)
16             S[i]++;
17     }
18 }
```

$O(n)$ $O(n^2)$

Se invece usiamo uno stack per memorizzare i prezzi delle giornate precedenti che si sono rivelati maggiori di quello del giorno attuale, allora la complessità diventa $O(n)$.

- We see that $S[i]$ on day i can be easily computed if we know the closest day preceding i , such that the price is greater than on that day than the price on the day i .
 - If such a day exists, let's call it $h(i)$, otherwise, we define $h(i) = -1$.
- The span is now computed as $S[i] = i - h(i)$. See the following diagram.



```

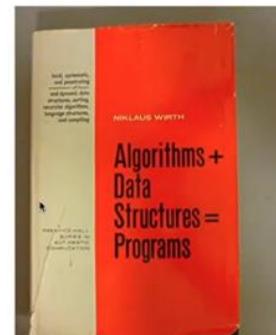
8 void calculateSpan(int price[], int n, int S[])
9 {
10    // Create a stack and push index of first
11    // element to it
12    stack<int> st;
13    st.push(0);
14
15    // Span value of first element is always 1
16    S[0] = 1;
17
18    // Calculate span values for rest of the elements
19    for (int i = 1; i < n; i++) {
20        // Pop elements from stack while stack is not
21        // empty and top of stack is smaller than
22        // price[i]
23        while (!st.empty() && price[st.top()] <= price[i])
24            st.pop();
25
26        // If stack becomes empty, then price[i] is
27        // greater than all elements on left of it,
28        // i.e., price[0], price[1], ..price[i-1]. Else
29        // price[i] is greater than elements after
30        // top of stack
31        S[i] = (st.empty()) ? (i + 1) : (i - st.top());
32
33        // Push this element to stack
34        st.push(i);
35    }

```

Every element of the array is added and removed from the stack at most once. So there are total $2n$ operations at most.



$O(n)$



LEZIONE 25

Classe bibliografia

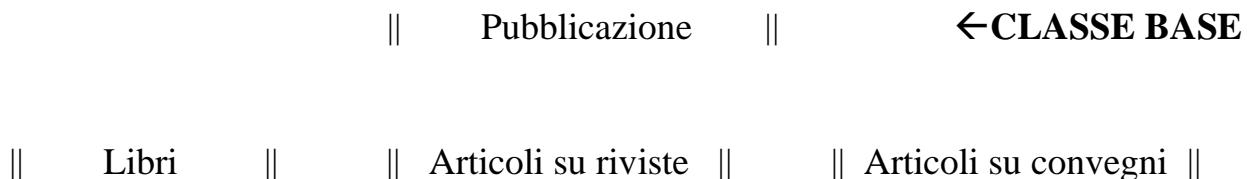
Riprendiamo un attimo un programma assegnato alla fine della lezione 19.

Un esercizio così lo dobbiamo saper fare, perché contiene un po' tutto e all'esame può uscire qualcosa di simile.

- Scrivere un programma c++ che permetta di gestire la bibliografia di un articolo scientifico.
- Il programma deve gestire un elenco di pubblicazioni,
 - Tutte le pubblicazioni hanno un titolo, uno o più autori, ed un anno di pubblicazione
 - E possono essere dei
 - Libri, vanno memorizzati le informazioni sulla casa editrice e l'ISBN
 - Articoli su riviste, vanno memorizzati il titolo della Rivista (es IEEE Transaction of Computers), il numero della rivista, e le pagine (inizio e fine)
 - Articoli su atti di convegni, vanno memorizzati il titolo del convegno (es IEEE Conference on Machine Learning), la sede del convegno, e il numero di pagina
 - Il programma deve permettere di memorizzare una lista di pubblicazioni in un vettore, stampare la lista, ed calcolare il numero di pubblicazioni di ciascun tipo a partire dai dati memorizzati nel vettore.
 - La stampa deve essere fatta in ordine alfabetico di cognome degli autori (overload dell'operatore <)
 - Implementare l'ordinamento di un vettore di pubblicazioni

Come implementare questo esercizio?

Potremmo fare una gerarchia di classi del tipo:



In particolare le variabili membro della classe base, comuni anche a quelle derivate quindi, saranno: titolo (\rightarrow stringa), anno di pubblicazione (\rightarrow int), uno o più autori (\rightarrow array di stringhe). Ovviamente ci saranno poi un costruttore, eventualmente un distruttore, funzioni get e funzioni set. Le classi derivate avranno delle variabili membro in più.

Il programma deve permettere di memorizzare in un vettore una lista di pubblicazioni, stampare la lista e calcolare il numero di pubblicazioni di ciascun tipo a partire dai dati memorizzati nel vettore stesso. Ovviamente il vettore lo dichiareremo come vettore di pubblicazioni, poiché deve contenere tutti e 3 i tipi di pubblicazioni. In particolare lo dichiareremo come puntatore a vettore di pubblicazioni. Per calcolare il numero di pubblicazioni di ciascun tipo, dobbiamo poi fare il *dynamic_cast* delle 3 classi derivate:

uno di questi 3 dovrebbe avere successo; quello che ha successo lo uso per incrementare il numero di pubblicazioni per quella classe derivata. Se avessimo usato uno *static_cast* non sarebbe stato un erroraccio, però dobbiamo saper usare il *dynamic_cast*; in generale per calcolare i numeri di pubblicazioni abbiamo queste 2 possibilità. Quindi quando usiamo l'ereditarietà o usiamo lo *static_cast* o usiamo il *dynamic_cast*: nel primo caso non ci preoccupiamo di controllare se il puntatore all'oggetto base punta davvero all'oggetto derivato specifico; nel secondo caso controlliamo a tempo di esecuzione e se il puntatore alla classe base veramente punta all'oggetto derivato specificato e se non è così allora lanciamo un'eccezione tipicamente (*throw*). Quindi *dynamic_cast* controlla che la conversione di classe ottenuta sfruttando l'ereditarietà sia valida.

Dobbiamo poi fare la stampa, ad esempio tramite un *toString()*, per la pubblicazione generica, e tramite dei *toString()* riscritti nelle classi derivate; quindi qui usiamo il polimorfismo.

Per quanto riguarda l'ordinamento, abbiamo saputo ordinare degli interi, ad esempio con il quicksort; ora dobbiamo ordinare un array di puntatori alla classe "Pubblicazione" in base all'ordine alfabetico degli autori. Se usiamo una struttura dati *vector* di *string*, possiamo sfruttare le funzioni implementate nella libreria e quindi usare l'operatore <, ma anche quello == se serve ad esempio. Allora per fare la stampa dobbiamo solo fare un overloading dell'operatore < tra le pubblicazioni sfruttando il < tra *vectors*. Se 2 autori hanno lo stesso nome possiamo ordinare ad esempio per anno.

Attenzione! Vogliamo ordinare le pubblicazioni, non gli autori!

Potremmo ordinare con il sort della libreria vectors. Dovremmo vedere sul manuale com'è fatta la funzione di ordinamento di quella libreria. Però cerchiamo di usare prima il nostro sort. Allora ci prendiamo il quicksort già scritto per ordinare degli interi e lo andiamo a riscrivere come template; l'unica cosa importante è che deve essere ben definito l'operatore <, che è l'unico che usiamo.

```
//Publication.h
#ifndef PUBLICATION_H
#define PUBLICATION_H
#include<string>
#include<vector>
#include<sstream>
#include<iostream>
using std::vector;
using std::cout;
using std::ostringstream;

class Publication{
public:
    //constructor
```

```

Publication(std::string ti, vector<std::string> au, int yr) : title{ti}, authors{au}, year{yr} {}

    //get and set functinos
    void setTitle(std::string ti){
        title=ti;
    }
    std::string getTitle() const{
        return title;
    }
    void setAuthors(vector<std::string> au){
        authors=au;
    }
    vector<std::string> getAuthors() const{
        return authors;
    }
    void setYear(int yr){
        year=yr;
    }
    int getYear() const {
        return year;
    }
    //overloading of operator <
    bool operator<(const Publication& B) {
        if (authors == B.getAuthors()){
            return year < B.getYear();
        }
        return authors<B.authors;
    }
    std::string toString(){
        ostringstream out;
        out<<"-\t";
        for (int i=0; i<authors.size(); i++){
            out<<authors[i];
            out<<((i==(authors.size()-1))? " " : ", ");
        }
        out<<"(";<<year;<<"): "<<title;
        return out.str();
    }

private:
    std::string title;
    vector<std::string> authors;
    int year;
};

#endif

```

```

//testPubs.cpp
#include "Publication.h"
#include <iostream>
#include <vector>
using std::vector;
using std::cout;

//QuickSort
template<typename T>
int partition(vector<T>& A, int p, int q);

template<typename T>
void quickSort(vector<T>& A, int p, int r){
    if (p<r){
        int q=partition(A, p, r);
        quickSort(A,p,q);
        quickSort(A,q+1,r);
    }
}

template<typename T>
void swap(vector<T>& A, int x, int y){
    T temp=A[x];
    A[x] = A[y];
    A[y]=temp;
}

template<typename T>
int partition(vector<T>& A, int p, int r){
    T x= A[p];
    int i=p-1;
    int j=r+1;
    while(true){
        do
            j=j-1;
        while (*x<*A[j]); //era A[j]>x, abbiamo riscritto in modo da avere solo l
        'operatore < //inoltre è un vettore di puntatori a pubblicazioni il
        nostro, quindi dereferenziamo
        do
            i=i+1;
        while (*A[i]<*x);
        if(i<j){
            swap(A,i,j);
        }
        else{
            return j;
        }
    }
}

```

```

        }
    }

int main(){
    Publication b1("A great book", {"Rossi M.", "Verdi C.", "Bianchi M."}, 2001);
    Publication b2("Another great book", {"Esposito M.", "Bianchi M."}, 2002);
    Publication b3("A great novel", {"Arnaldi M.", "Bianchi M."}, 2002);

    vector<Publication*> bib;
    bib.push_back(&b1);
    bib.push_back(&b2);
    bib.push_back(&b3);
    quickSort(bib, 0, bib.size()-1);
    for(int i=0; i<bib.size(); i++){
        cout<<bib[i]->toString()<<"\n";
    }
}

```

Makefile:

```

1 testPubs: testPubs.o
2     g++ -o testPubs testPubs.o
3 testPubs.o: testPubs.cpp Publication.h
4     g++ -c testPubs.cpp -std=c++11

```

E quello che esce fuori è questo:

```

PS C:\Users\ALESSANDROCOCCHIA\Desktop\Programmi_miei\L25-L9_Pubblicazioni> ./testPubs.exe
-      Arnaldi M., Bianchi M. (2002): A great novel
-      Esposito M., Bianchi M. (2002): Another great book
-      Rossi M., Verdi C., Bianchi M. (2001): A great book

```

Ora invece proviamo ad utilizzare la libreria `<vector>`. Per potere usare il suo sort dobbiamo includere anche `<algorithm>`. A *sort* dobbiamo passare l'inizio del vettore da ordinare, la fine e poi dobbiamo passare comunque una funzione scritta da noi che faccia il confronto per poter definire l'operatore `<`.

```

44 | bool compareBib(Publication* A, Publication* B){
45 |     return *A < *B;
46 |

```

Questa funzione sfrutta il `<` riscritto (*overloaded*) in `Publication.h`. `<vector>` ha bisogno di sapere la modalità con cui effettuare il confronto perché altrimenti di base confronterebbe gli indirizzi di tutti sti oggetti; quindi dobbiamo dereferenziare noi ed usare il nostro operatore `<`.

Richiamiamo sort:

```
64 | //quickSort(bib,0,bib.size()-1);
65 | std::sort(bib.begin(),bib.end(),
66 |             compareBib);
```

L'esito del programma è lo stesso.

Bene, ora dobbiamo fare le classi derivate.

Definiamo una classe *Book.h*.

```
//Book.h
#ifndef BOOK_H
#define BOOK_H
#include "Publication.h"
#include <string>
#include <vector>
#include <iostream>
using std::string;
using std::vector;

class Book : public Publication {
public:
    Book(string ti, vector<string> au, int yr, string pub, string is) : Publication(ti, au, yr), Publisher(pub), ISBN(is) {};
    string toString(){
        std::ostringstream out;
        out<<Publication::toString() << ". ";
        out<<Publisher<<", "<< ISBN;
        return out.str();
    }
private:
    string Publisher;
    string ISBN;
};

#endif
```

A questo punto modifichiamo il main, *testPubs*, con 2 ulteriori righe, oltre all'inclusione di *Book.h*:

```
Book b4("The Art of Computer Programming", {"Knuth D."}, 1968, "Academic Publishers",
", "89283-7237362");
```

```
bib.push_back(&b4);
```

Dovremmo creare anche le altre classi derivate, non lo facciamo per tempo.

Ora ci manca solamente il conteggio degli elementi. All'interno di un ciclo for, per ogni pubblicazione (book, journal, l'altro) potremmo andare a provare a fare il dynamic_casting ed incrementare valore del conteggio.

Noi invece adesso per mancanza di tempo andiamo semplicemente a contare il numero di oggetti di tipo Book istanziati; lo facciamo con una variabile statica, perché in questa maniera il suo valore è unico e condiviso tra tutti gli oggetti della classe e non può essere alterato, ed ogni volta che creiamo un Book, quindi nel costruttore di Book, incrementiamo questa variabile statica. Istanziamo poi anche un distruttore in cui decrementiamo questa variabile. Inizializziamo poi a 0 questa variabile statica nel main (non c'è bisogno di ridefinirla come *static*).

Nel main:

```
57 int Book::nBooks=0;
58
59 int main(){
60     Publication b1("A great book", {"Rossi M.", "Verdi C.", "Bianchi M."}, 2001);
61     Publication b2("Another great book", {"Esposito M.", "Bianchi M."}, 2002);
62     Publication b3("A great novel", {"Arnaldi M.", "Bianchi M."}, 2002);
63     Book b4("The Art of Computer Programming", {"Knuth D."}, 1968, "Academic Publishers", "89283-7237362");
64
65     vector<Publication*> bib;
66     bib.push_back(&b1);
67     bib.push_back(&b2);
68     bib.push_back(&b3);
69     bib.push_back(&b4);
70
71     //quickSort(bib, 0, bib.size()-1);
72     std::sort(bib.begin(), bib.end(), compareBib);
73
74     for(int i=0; i<bib.size(); i++){
75         cout<<bib[i]->toString()<<"\n";
76     }
77     cout<<"Number of books in bibliography: "<<Book::nBooks<<"\n";
```

In *Book.h*:

```
14     Book(string ti, vector<string> au, int yr, string pub, string is) : Publication(ti, au, yr), Publisher(pub), ISBN(:)
15     |    ++nBooks;
16     |};
17     ~Book(){--nBooks;}
```

Per implementare il *dynamic casting* ci conviene definire la classe *Publication* come *pure virtual* così da potere istanziare oggetti solo dei 3 tipi di pubblicazioni specificati; basta prendere una generica funzione della classe e porla = 0 (non una che usiamo, come *toString()*).

LEZIONE 26

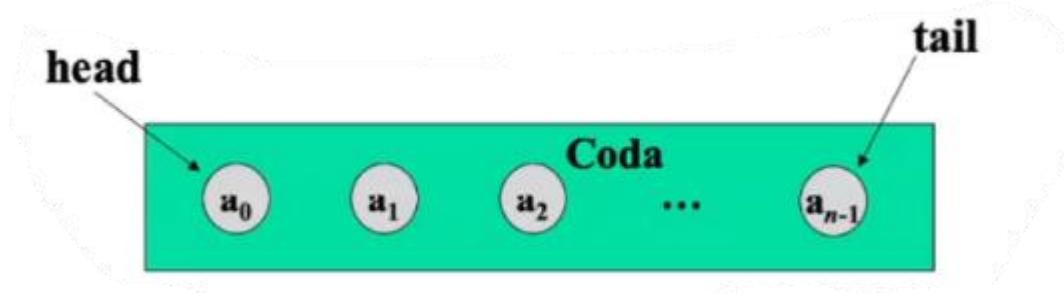
Abbiamo visto quell'esercizio la scorsa volta. Provare a farlo usando al posto dei vettori delle liste (normali o doppiamente) concatenate a puntatori. La complicazione starà nel fatto che non possiamo usare il `<` come abbiamo fatto coi vettori; con *vectors* potevamo confrontare vettori tra di loro con l'operatore `<` senza problemi. Per le liste non vale lo stesso. Dobbiamo scrivere noi un operatore `<` in modo da confrontare le liste. Sostanzialmente dobbiamo confrontare il primo elemento (l'autore) di una lista con il primo elemento di un'altra lista: se è minore allora l'operatore `<` ritorna *true*.

Possiamo usare sia la nostra implementazione della lista, sia quella della STL che alla fine ha più o meno le stesse funzioni membro (dopo la vediamo). In realtà la STL usa una lista doppiamente linkata come struttura dati sottostante.

CODE

Avevamo visto la struttura dati astratta “stack”; quella delle code è un'altra struttura dati astratta. Ricordiamo che quello che stiamo facendo è separare l'interfaccia dall'implementazione ed in generale ciò che ci dovrà rimanere di questo corso è proprio come farlo e come scegliere l'implementazione più adeguata.

Una **coda** è una struttura dati che implementa una policy di tipo **FIFO** (First in – First out), a differenza dello stack che implementava una policy di tipo LIFO. FIFO vuol dire che si estraggono gli oggetti nello stesso modo in cui essi sono inseriti: il primo ad essere inserito è il primo ad essere estratto.



La coda tipicamente ha 2 attributi: l'*head* (o *front*), ovvero l'inizio, e il *tail* (o *rear*), ovvero la fine. In pratica gli elementi entrano (**push**) dalla fine (*tail* o *rear*) ed escono (**pop**) dall'inizio (*head* o *front*).

Allora sulle code sono definite 3 funzioni:

- **ENQUEUE(o:element)** – Inserisce l’oggetto o al fondo della coda
- **DEQUEUE()** – Rimuove l’oggetto dalla testa della coda; viene generato un errore se la coda è vuota
- **FRONTO:** - Restituisce, senza rimuoverlo, l’elemento alla testa della coda; viene generato un errore se la coda è vuota

E poi ci sono le 2 funzioni di supporto:

- **SIZE()**
- **EMPTY(S: ADT)**

Anche la coda è semplice da implementare con un array o con una lista.

Il primo metodo, quello dell’**array**, è più semplice e prevede che un array venga riempito parzialmente e *head* sarà l’indice del prossimo elemento da estrarre, *tail* l’indice del prossimo elemento da inserire. Quindi man mano che vengono accodati oggetti il *tail* si sposta verso destra, man mano che vengono estratti oggetti il *head* si sposta a destra. Quando il *tail* arriva alla fine dell’array allora si torna all’inizio: per questo implementiamo la struttura con un array circolare.



È come se la posizione $N-1$ fosse collegata alla posizione 0 : per questo lo chiamiamo “array circolare”. Quindi l’incremento di *tail* è “modulo- N ”, nel senso che arrivato alla posizione $N-1$ dell’array riparte da 0 . E lo stesso vale per *head*.

- **Incremento: $head := head + 1 \bmod N$, $tail = tail + 1 \bmod N$**

All’**inizio** $head = 0$, $tail = 0$.

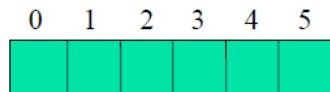
La condizione di coda piena (overflow) corrisponde a $head = tail + 1 \bmod N$

La condizione di coda vuota (underflow) corrisponde a $head = tail$.

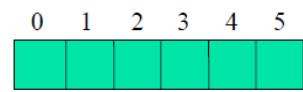
Sia il caso in cui la coda è piena che quello in cui la coda è vuota coincidono con la sovrapposizione di *tail* e *head*. Per distinguere allora i 2 casi si sacrifica una posizione nell’array. La condizione di coda vuota è semplicemente $head = tail$; la condizione di coda piena invece è quando $head = (tail + 1) \bmod N$; mod N è la divisione per N dove

viene calcolato non il risultato ma il resto. Per tutti i numeri $a < N$, $a \bmod N = a - 0*N = a$. Vediamo un esempio di *enqueue* (inserimento) e *dequeue* (estrazione) in una coda implementata con un array.

Esempio ($N=6$)



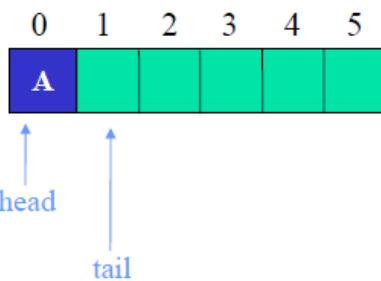
NEW



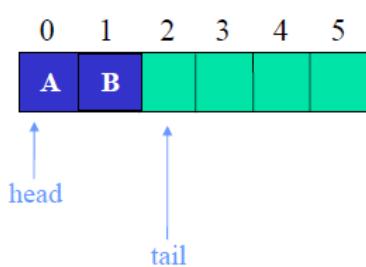
head
tail



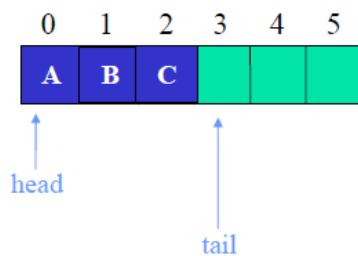
ENQUEUE(Q,B)



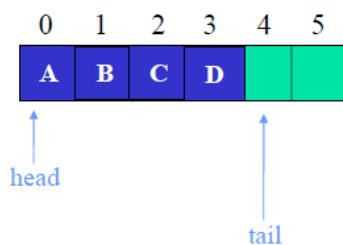
ENQUEUE(Q,B)



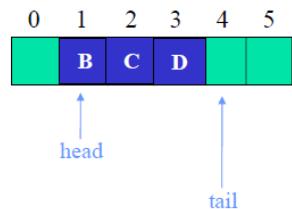
ENQUEUE(Q,C)



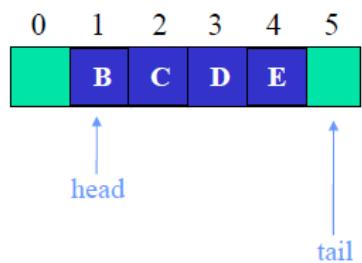
ENQUEUE(Q,D)



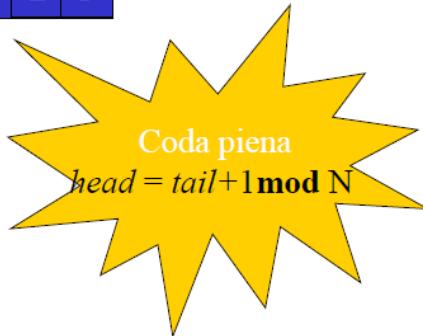
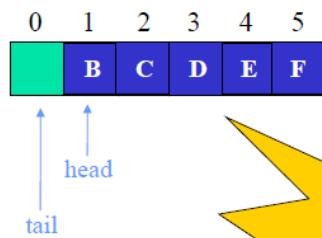
DEQUEUE(Q)



ENQUEUE(Q,E)

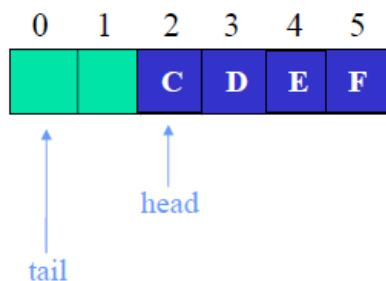


ENQUEUE(Q,F)

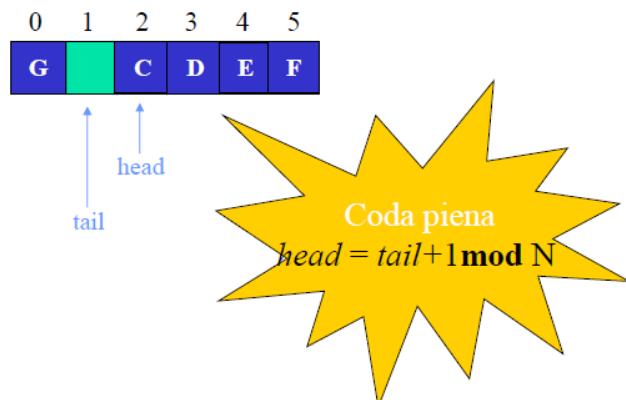


Vediamo che per dover rappresentare la condizione di coda piena, distinguendola da quella di coda vuota, non utilizziamo un elemento del vettore.

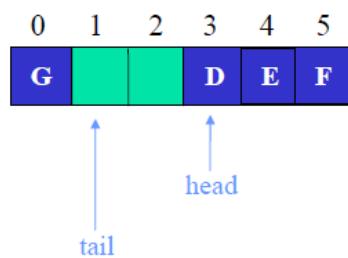
DEQUEUE(Q)



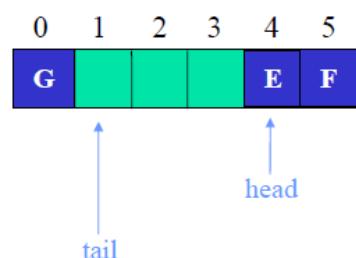
ENQUEUE(Q,G)



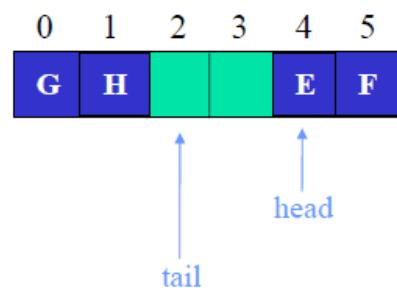
DEQUEUE(Q)



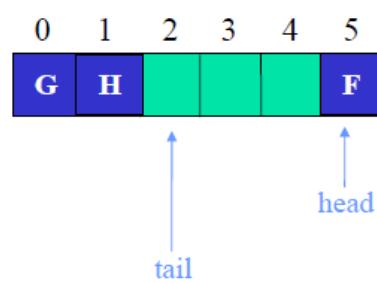
DEQUEUE(Q)



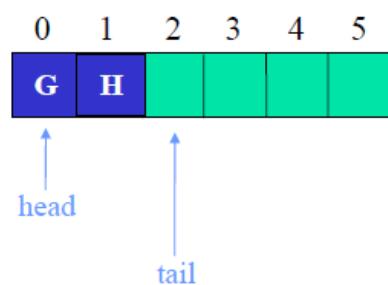
ENQUEUE(Q,H)



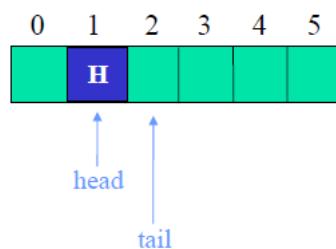
DEQUEUE(Q)



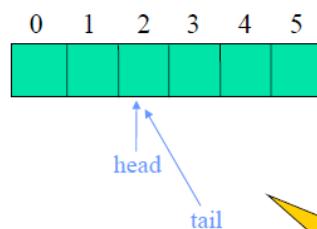
DEQUEUE(Q)



DEQUEUE(Q)



DEQUEUE(Q)



Fare come esercizio: implementare una coda con un array circolare.

- Esercizio da Fare:
- Implementare una coda mediante array circolare

```
Algorithm size():
    return n
Algorithm empty():
    return (n = 0)
Algorithm front():
    if empty() then
        throw QueueEmpty exception
    return Q[f]
Algorithm dequeue():
    if empty() then
        throw QueueEmpty exception
    f ← (f + 1) mod N
    n = n - 1
Algorithm enqueue(e):
    if size() = N then
        throw QueueFull exception
    Q[r] ← e
    r ← (r + 1) mod N
    n = n + 1
```

Però visto che non vogliamo essere vincolati a scegliere a priori la dimensione della coda e non vogliamo sacrificare un elemento dell'array per distinguere coda piena da coda vuota, può essere utile rappresentare una coda con una **lista concatenata**. Paghiamo ovviamente il fatto di dover mantenere dei puntatori in più (in un array non abbiamo la necessità di rappresentare un *next*), però questo ci permette di aggiungere elementi dinamicamente senza fissare una dimensione.

Tra le varie liste, quale conviene usare?

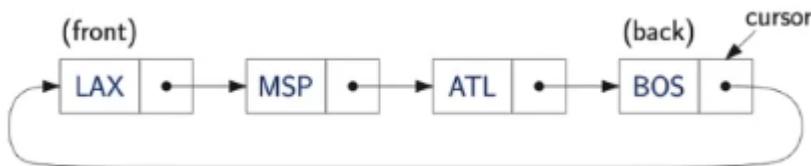
La lista *linked* è utile per inserire e rimuovere al front. Non converrebbe per inserire e rimuovere in coda, perché significherebbe partire dall'head ed arrivare alla coda scorrendo tutta la lista.

La lista *double linked* è molto utile in generale perché usa le sentinelle e quindi inserimento e delezione sono molto semplici ed infine abbiamo accesso sia alla testa che alla coda. Quindi perché non usarla? Perché in realtà non ci serve andare avanti e indietro nella lista, dobbiamo solo poter accedere a testa e coda.

Allora la migliore strategia potrebbe essere quella di usare liste linkate circolari.

La lista *circle linked* abbiamo visto che mantiene un puntatore *cursor* e fornisce due member functions, *back* e *front*, per restituire rispettivamente l'elemento puntato da *cursor* e quello immediatamente successivo. La cosa comoda è che un unico puntatore all'ultimo elemento, il cursore, fornisce sia la testa che la coda; e quindi la lista circolare è ideale per la creazione di una coda. Quindi il *tail* e il *head* non sono altro che il *back* e il *front* della lista circolare.

Ricordiamo un attimo la struttura di una *circle linked list*:



Oss:

La coda della STL invece è implementata tramite un *vector*.

CODA con CIRCULAR LINKED LIST:

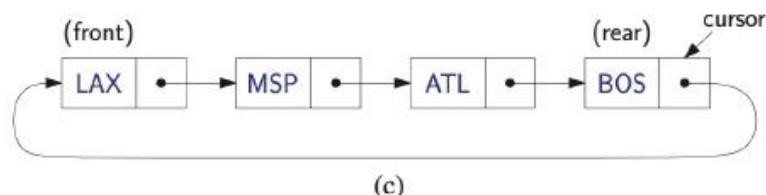
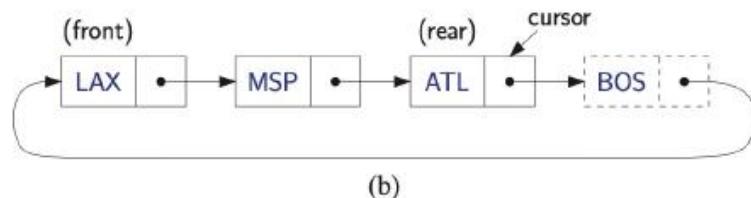
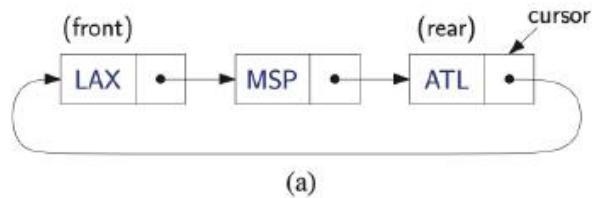
Andiamo ad includere la *circular linked list* già implementata e scriviamo una classe *LinkedQueue* di interfaccia per dichiarare la coda.

```
1 //LinkedQueue.h
2 #ifndef LINKED_QUEUE_H
3 #define LINKED_QUEUE_H
4 #include <stdexcept>
5 #include <iostream>
6 #include "CLinkedList.h"
7 template <typename E>
8 class LinkedQueue {           // queue as doubly linked list
9 public:
10     LinkedQueue();           // constructor
11     int size() const;        // number of items in the queue
12     bool empty() const;      // is the queue empty?
13     const E& front();        // the front element
14     void enqueue(const E& e); // enqueue element at rear
15     void dequeue();          // dequeue element at front
16 private:                     // member data
17     CircleList<E> C;       // circular list of elements
18     int n;                  // number of elements
19 };
```

Vediamo l'implementazione:

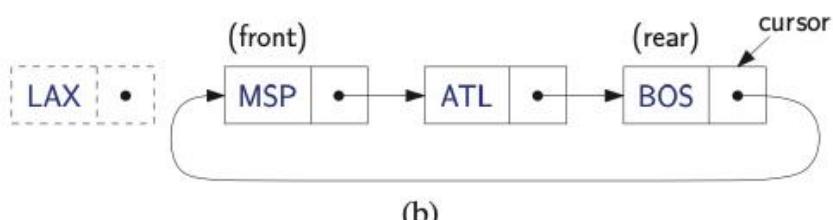
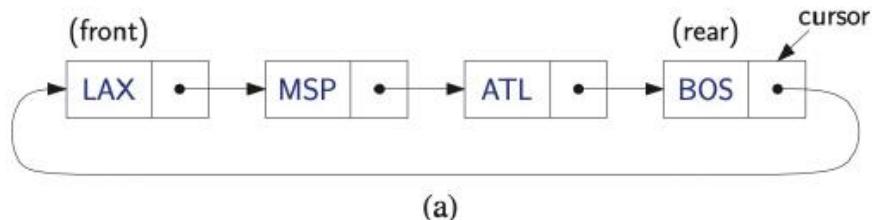
```
21 template <typename E>
22 LinkedQueue<E>::LinkedQueue()
23     // constructor
24     : C(), n{0} {}
25
26 template <typename E>
27 int LinkedQueue<E>::size() const {
28     // number of items in the queue
29     return n;
30 }
31
32 template <typename E>
33 bool LinkedQueue<E>::empty() const {
34     // is the queue empty?
35     return n == 0;
36 }
37
38 template <typename E>
39 const E& LinkedQueue<E>::front() {
40     // get the front element
41     if (empty())
42         throw std::runtime_error("front of empty queue");
43     return C.front();           // list front is queue front
44 }
```

Enqueue



```
46 template <typename E>
47 void LinkedQueue<E>::enqueue(const E& e) {
48     // enqueue element at rear
49     C.add(e);           // insert after cursor
50     C.advance();        // ...and advance
51     n++;
52 }
```

Dequeue



```
-- 54 template <typename E>
55 void LinkedQueue<E>::dequeue() {
56     // dequeue element at front
57     if (empty())
58         throw std::runtime_error("dequeue of empty queue");
59     C.remove();          // remove from list front
60     n--;
61 }
```

E poi scriviamo un main per testare la coda:

Test dell'implementazione

```
2 #include <iostream>
3 #include <string>
4 #include "LinkedQueue.h"
5
6 int main(){
7     LinkedQueue<int> Q;
8     Q.enqueue(1);
9     Q.enqueue(2);
10    Q.enqueue(3);
11    std::cout << Q.front() << std::endl;
12    Q.dequeue();
13    std::cout << Q.front() << std::endl;
14    Q.dequeue();
15    std::cout << Q.front() << std::endl;
16    Q.dequeue();
17    return 0;
18 }
```

```
$ g++ testLinkedQueue.cpp -o testLinkedQueue -std=c++11
$ ./testLinkedQueue
1
2
3
```

La libreria STL mette a disposizione, come già accennato, una struttura dati “coda” esattamente con le stesse funzioni (enqueue, dequeue e front), basata su *vectors*. Per usare questa struttura dati dobbiamo includere il template <queue>.

```
1 #include <queue>
2
3 using std::queue; // make queue accessible
4 queue<float> myQueue;
```

(DEQUE) Double-Ended Queues (si legge DEC)

Sono code a doppia estremità in cui gli inserimenti e le delezioni possono essere effettuati sia nel *front* che nel *back* della coda. La DEQUE è una struttura dati simile alla coda ma appunto ha come differenza rispetto a questa il fatto che inserimenti e delezioni possono essere fatti sia all'inizio che alla fine. La *deque* è sempre un *container*, ovvero un contenitore dinamico di oggetti che ci permette di aggiungere e rimuovere degli oggetti in certe posizioni. Tutte queste che stiamo vedendo in queste lezioni sono ADT che ci permettono di gestire oggetti in maniera dinamica con delle funzioni di supporto. Tuttavia non stiamo ancora parlando del supporto alla ricerca (*search*). Infatti queste strutture dati vengono utilizzate in algoritmi che non danno il supporto all'interrogazione e quindi alla ricerca. Non solo, ma ci sono altre strutture dati che danno supporto all'ordinamento, dando supporto all'elemento successivo ed al precedente; infatti, evidentemente, trovare in maniera rapida l'elemento precedente e quello successivo permette di implementare l'ordinamento.

Tornando alla *deque*, inserimento e delezione vengono fatti all'inizio (*front*) ed alla fine(*back*):

- ***insertFront(e)***: Insert a new element *e* at the beginning of the deque.
- ***insertBack(e)***: Insert a new element *e* at the end of the deque
- ***eraseFront()***: Remove the first element of the deque; an error occurs if the deque is empty
- ***eraseBack()***: Remove the last element of the deque; an error occurs if the deque is empty.

E poi abbiamo le funzioni di supporto:

- ***front()***: Return the first element of the deque;
- ***back()***: Return the last element of the deque;
- ***size()***: Return the number of elements of the deque.
- ***empty()***: Return true if the deque is empty and false otherwise.

Qual è la struttura concreta più idonea per l'implementazione di una coda a doppia terminazione (DEQ)? Evidentemente una lista doppiamente concatenata. Infatti le 2 sentinelle danno la possibilità di aggiungere e rimuovere sia in coda che in testa.

Anche qui in realtà la STL mette a disposizione un tipo di dato *deque* che può essere utilizzato includendo il template `<deque>`. Questa implementazione è basata molto su `<vectors>`, come molte strutture dati del C++ (ad esempio stack e coda).

```
1 | #include <deque>
2 | using std::deque; // make deque accessible
3 | deque<string> myDeque; // a deque of strings
```

Le operazioni principali della *deque* della libreria STL sono praticamente quelle che abbiamo visto prima:

The principal operations of the STL deque:

- `size()`: Return the number of elements in the deque.
- `empty()`: Return true if the deque is empty and false otherwise.
- `push_front(e)`: Insert `e` at the beginning the deque.
- `push_back(e)`: Insert `e` at the end of the deque.
- `pop_front()`: Remove the first element of the deque.
- `pop_back()`: Remove the last element of the deque.
- `front()`: Return a reference to the deque's first element.
- `back()`: Return a reference to the deque's last element.

Possiamo usare tranquillamente l'implementazione della STL, però è più furbo utilizzare una lista doppiamente linkata.

Deque con lista doppiamente linkata:

```
1 #include "DLinkedList.h"
2 //LinkedDeque.h
3 template <typename T>
4 class LinkedDeque { // deque as doubly linked list
5 public:
6     LinkedDeque();
7     int size() const;
8     bool empty() const;
9     const T& front();           // the first element
10    const T& back();           // the last element
11    void insertFront(const Elem& e); // insert new first element
12    void insertBack(const Elem& e); // insert new last element
13    void removeFront();          // remove last element first
14    void removeBack();           // remove last element
15 private:
16     DLinkedList<T> D;
17     int n;
18 };
```



L'unico commento da fare è che `n`, così come in altre occasioni, rappresenta il numero di elementi della struttura, in questo caso la deque implementata con la lista, e potremmo farne a meno e calcolare il numero di elementi con la funzione `size()`. Però ci conviene avere una variabile membro apposita, è più efficiente; al solito c'è sempre un trade-off tra efficienza ed occupazione in memoria.

Una Deque ha un duplice comportamento: se si estraе da sinistra si comporta da coda (FIFO), se si estraе da destra si comporta da stack (LIFO).

Implementazione delle funzioni:

```
template<typename T>
LinkedDeque<T>::LinkedDeque() : D(), n{0} {}
template<typename T>
int LinkedDeque<T>::size() const {
    return n;
```

```

}

template<typename T>
bool LinkedDeque<T>::empty() const {
    return n==0;
}

template<typename T>
const T& LinkedDeque<T>::front() {
    return (D.front());
}

template<typename T>
const T& LinkedDeque<T>::back() {
    return (D.back());
}

template<typename T>
void LinkedDeque<T>:: insertFront(const T& e) {
    return D.addFront(e);
    n++;
}

template<typename T>
void LinkedDeque<T>:: insertBack(const T& e){
    return D.addBack(e);
    n++;
}

template<typename T>
void LinkedDeque<T>:: removeFront(){
    if (empty()){
        throw std::runtime_error("deque empty");
    }
    return D.removeFront();
    n--;
}

template<typename T>
void LinkedDeque<T>:: removeBack(){
    if (empty()){
        throw std::runtime_error("deque empty");
    }
    return D.removeBack();
    n--;
}

```

Insomma, il lavoro sporco l'avevamo già fatto con l'implementazione della Doubly Linked List.

Esercizio da fare:

Esercise

- Implement Bubble sort on a doubly linked list

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
    until not swapped
end procedure
```

	pass	swaps	sequence
1st	$7 \leftrightarrow 2$	$7 \leftrightarrow 6$	(5, 7, 2, 6, 9, 3)
2nd	$5 \leftrightarrow 2$	$7 \leftrightarrow 3$	(5, 2, 6, 7, 3, 9)
3rd	$6 \leftrightarrow 3$		(2, 5, 6, 3, 7, 9)
4th	$5 \leftrightarrow 3$		(2, 3, 5, 6, 7, 9)

Questo è un altro algoritmo di ordinamento, chiamato *Bubble Sort*. Noi per ora abbiamo visto il *Quick Sort*, che mette gli elementi piccoli a sinistra e quelli grandi a destra tramite una funzione *partition*, il *Merge Sort*, che ordina prima la parte di sinistra e poi quella di destra, e l'*Insertion Sort* in cui si mantiene la lista sempre ordinata e si inserisce il prossimo elemento in maniera ordinata nella lista. Il Bubble Sort parte dal primo elemento e confronta ogni elemento con quello successivo: se è maggiore allora effettua lo scambio. In questa maniera alla fine del primo ciclo si troverà sicuramente come ultimo elemento nella lista quello più grande, ovvero il massimo. E quindi l'ultimo elemento è ordinato. Al secondo ciclo gli ultimi 2 elementi saranno ordinati, poiché il secondo massimo andrà nella posizione giusta, al terzo ciclo gli ultimi 3 saranno ordinati e così via.

Questa cosa ci costa n^2 ! In realtà alla seconda passata arriviamo ad $n-1$, alla terza passata ad $n-2$, alla quarta ad $n-3$ e così via; però la crescita è quella. Sarebbe la sommatoria, per i che va da 1 ad n , di $n-i$ e quindi è uguale a sommatoria per i che va da 1 ad n di i . Quanto costa questa sommatoria? la metà di n^2 , ma i fattori costanti asintoticamente non ci interessano.

Però questo algoritmo è interessante per le doubly linked list, poiché richiede il confronto con l'elemento successivo per ogni elemento ed in ogni posizione noi abbiamo effettivamente l'accesso all'elemento ed a quello successivo.

Come ci converrebbe procedere? O cambiamo solo gli oggetti ma non i puntatori, o cambiamo anche i riferimenti. Ovviamente la seconda strada è più difficile e ci può costare di più.

Per ricapitolare, *Merge Sort* e *Quick Sort* sono algoritmi $O(n \log(n))$, mentre *Insertion Sort* e *Bubble Sort* sono $O(n^2)$. Però tipicamente quello che si fa ad esempio è, visto che *Insertion Sort* ha dei fattori costanti molto bassi, applicare *Quick Sort* per dimensioni grandi, però arrivati ad un certo punto quando la dimensione è piccola si applica l'*Insertion Sort*. Si dimostra formalmente che per un algoritmo di ordinamento meno di $O(n \log(n))$ non si può fare. Però in realtà si può fare un ordinamento in tempo lineare $O(n)$, ma solo quando lo spazio delle chiavi, ovvero gli oggetti da ordinare, è limitato in un certo intervallo specifico (ad esempio ci sono solo numeri che vanno da 1 a 100). Quindi imponendo dei vincoli sulle chiavi si riesce ad implementare algoritmi più veloci.

ITERATORS

Gli *iterators* sono degli oggetti che permettono di esplorare uno specifico *container*.

La STL ha vari containers:

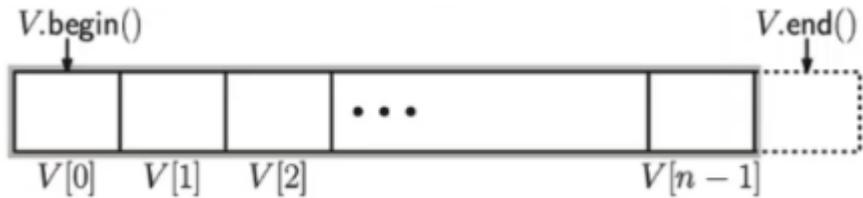
stack
queue
deque
vector
list
priority queue
set
map

Abbiamo accennato fino ad ora solo a stack, deque, queue e vector. Sono implementate dalla STL però anche le liste e le code a priorità; in queste ultime si estrae l'oggetto con la chiave maggiore e quindi con priorità maggiore; è come se in un insieme di clienti da servire ce ne fossero alcuni con priorità maggiore da servire (estrarre) prima; per poter implementare questa struttura bisogna saper implementare un algoritmo che calcoli il massimo → la *priority queue* è una struttura dati che supporta il massimo (o il minimo). Quindi di fatto una priority queue è una coda con supporto al massimo (o minimo). Esiste una struttura dati concreta, l'*heap*, che di fatto è un albero alla cui testa c'è o il massimo o il minimo e che permette di ottenere il massimo in tempo $O(1)$ e rimuovere elementi in tempo $O(\log(n))$. Ed esiste un algoritmo, che si chiama *heap sort*, basato su questa struttura dati. E pure la *priority queue* si implementa con una struttura *heap*.

Set e *map* invece danno il supporto alla ricerca ed all'ordinamento: nel *set*, così come nel *map*, ho la possibilità di cercare un oggetto → va implementato in maniera efficiente il search. Le mappe poi possono essere ordinate o meno, e nel secondo caso servono

per fare solo search. Nelle mappe ordinate invece c'è il supporto al predecessore ed al successore. I *set* poi si dividono in *set* e *multi-set*; nel secondo caso 2 oggetti possono avere la stessa chiave.

Gli **iterators** sono a loro volta delle classi messe a disposizione dalle varie classi implementate dalla STL elencate poco fa: ogni oggetto stack, deque, queue, list, vector, e così via, ha associato un oggetto che si chiama "iterator" che specifica una particolare posizione all'interno dell'insieme dinamico (container) ed ha l'abilità di navigare tra le posizioni per scorrere la struttura. Quindi, se **p** è un iterator, lo si può incrementare o decrementare (**++p**, **--p** o **p++**, **p--**) e lo si può dereferenziare (poiché di fatto l'iterator è una classe che contiene un puntatore ad un elemento dell'insieme dinamico) che permette di accedere all'oggetto puntato dall'iterator stesso. Inoltre ogni classe container STL mette a disposizione le funzioni membro **.begin()** e **.end()**, che ritornano proprio un iteratore; in particolare **end** punta alla posizione successiva all'ultimo elemento.



Vediamo un esempio:

```

1 #include <vector>
2 using std::vector;
3 int vectorSum1(vector<int> V) {
4     vector<int>::iterator p; // iterator of our vector of int
5     int sum = 0;
6     for (p = V.begin(); p != V.end(); ++p)
7         sum += *p;
8     return sum;
9 }
```

Usiamo in questo esempio un iterator per accedere ad un elemento di vector. Certo, per il vector possiamo usare l'operatore [] per accedere al container in questione, ma con la lista no ad esempio. La funzione di sopra non è scritta proprio bene perché V dovrebbe essere passato per riferimento. Se però non vogliamo dare la possibilità alla funzione di modificare gli elementi, la STL mette a disposizione i constant iterators, che sono iteratori che viaggiano tra gli oggetti ma non hanno la possibilità di alterare il contenuto della struttura dati.

Alcune funzioni della STL basate sugli iterators:

- `vector(p,q)`: Construct a vector by iterating between p and q , copying each of these elements into the new vector.
- `assign(p,q)`: Delete the contents of V , and assigns its new contents by iterating between p and q and copying each of these elements into V .
- `insert(p,e)`: Insert a copy of e just prior to the position given by iterator p and shifts the subsequent elements one position to the right.
- `erase(p)`: Remove and destroy the element of V at the position given by p and shifts the subsequent elements one position to the left.
- `erase(p,q)`: Iterate between p and q , removing and destroying all these elements and shifting subsequent elements to the left to fill the gap.
- `clear()`: Delete all these elements of V .

Vediamo invece un esempio sulla lista messa a disposizione dalla STL e la cui implementazione è basata su una doubly linked list:

```

20 //copy a List into a Vector
21
22 list<int> L; // an STL list of integers
23 // ...
24 vector<int> V(L.begin(), L.end()); // initialize V to be a copy of L
25 ■

```

LEZIONE 27

Avevamo solo accennato, ma andiamo più nello specifico:

PRIORITY QUEUE

La priority queue è una coda a priorità ed è una struttura astratta che serve per fare push e pop ma tramite una policy che si basa sul dare priorità agli oggetti non in base all'ordine di arrivo nella coda ma in base ad una priorità dell'oggetto stesso: si estrarre (pop) dalla coda l'oggetto che ha più alta priorità; e la priorità è una proprietà dell'oggetto in base alla quale stabiliamo chi oggetto ha precedenza su chi. Questa proprietà non è altro che una chiave associata ad un certo oggetto. Quindi abbiamo una serie di oggetti che mettono a disposizione l'operatore “ $<$ ” in base al quale ordinarli rispetto alla precedenza stabilita ed il *pop* tiene conto della priorità all'interno di questo insieme di oggetti. Le code a priorità sono utilizzate molto dai SO che devono gestire tantissimi processi che richiedono accesso alle risorse del sistema e per questo qualsiasi processo porta con sé una priority.

Poiché ogni volta si estraе il massimo, una cosa molto furba potrebbe essere ricavare “a gratis” un algoritmo di ordinamento. Ovviamente potremmo adattare l’operatore “ $<$ ” per ottenere il minimo piuttosto che il massimo.

Allora vale la pena soffermarsi su una struttura dati che si chiama HEAP che è basata su una rappresentazione di alberi binari e che ci permette di estrarre in maniera efficiente massimo e minimo da un insieme e di ottenere un algoritmo di ordinamento che si chiama **heap sort**: questo algoritmo ha complessità $O(n \log(n))$ sia nel caso peggiore che nel caso medio e non richiede spazio aggiuntivo in memoria (come succede per il Merge Sort). Però prima vedremo le funzionalità che offre la ADT *Priority Queue*.

Una coda con priorità è un tipo di dati astratto ADT per mantenere un insieme S di elementi, ciascuno con un valore associato detto **chiave**

Sulla coda con priorità sono definite le seguenti operazioni

- **Push(S, x)** inserisce l'elemento x nell'insieme S ($S \leftarrow S \cup \{x\}$)
- **Top(S)** restituisce l'elemento di S con la chiave più grande
- **Pop(S)** restituisce e rimuove l'elemento di S con la chiave più grande

Di fatto se implementassimo la coda a priorità con una lista così da fare in modo che i push mettano l'elemento da inserire sempre al top della lista, in tal caso il pop sarebbe semplice però il push non necessariamente perché l'elemento va inserito in ordine. Per questo si utilizza la struttura dati heap, più efficiente, per bilanciare *pop* e *push* e per poter fare quindi sia push che pop in tempo O(log(n)).

La STL mette a disposizione stesso nella classe <queue> la possibilità di definire una *priority queue*.

```
1 #include <iostream>
2 #include <queue>
3
4 using std::priority_queue;
5 using std::cout;
6
7 int main()
8 {
9     priority_queue<int> pq;
10    pq.push(10);
11    pq.push(30);
12    pq.push(20);
13    pq.push(5);
14    pq.push(1);
15
16    cout << "pq.size() : " << pq.size() << "\n";
17    cout << "pq.top() : " << pq.top() << "\n";
18    cout << "We can use a priority que to sort: -->>\n";
19    while(!pq.empty()) {
20        cout << pq.top() << "\t";
21        pq.pop();
22    }
23    cout <<"\n";
24    return 0;
25 }
```

Sono implementate sempre le funzioni di *top*, *push* e *pop*, però *pop* e *top* operano, differentemente da una coda senza priorità, sugli elementi massimi. Ci sono poi le funzioni di supporto che sono *size* ed *empty*. Quindi alla riga 17 il top non restituirà il primo elemento inserito (come nella coda) o l'ultimo elemento inserito (come nello stack), bensì l'elemento massimo. E allora con il while alla riga 19 non facciamo altro che stampare una sequenza di oggetti ordinata in ordine decrescente, e quindi di fatto facciamo un ordinamento.

Se eseguiamo esce questo:

```
^Micheles-MacBook-Pro:L15 ceccarelli$ ./a.out
pq.size() : 5
pq.top() : 30
We can use a priority que to sort: -->
30      20      10      5      1
Micheles-MacBook-Pro:L15 ceccarelli$
```

Noi come la implementeremo una coda a priorità? Potremmo con un vettore, con una lista, etc., ma come già detto useremo una struttura **heap** (heap sta per “mucchio”).

PRIORITY QUEUE WITH HEAP

Stesso la STL implementa la coda a priorità con una struttura heap. “Heap” perché si tratta di un mucchio di oggetti di cui si assume che in testa ci sia il massimo.

```
Selection-Sort(A[1..n]):
  For i = n downto 2
    A:   Trova l'elemento più grande fra A[1..i]
    B:   Scambialo con A[i]
```

- Il passo **A** prende $\Theta(n)$ e **B** prende $\Theta(1)$: in totale $\Theta(n^2)$
- **Idea:** utilizzo di una **struttura dati** per effettuare sia **A** che **B** in $O(\lg n)$, bilanciando il lavoro per un migliore compromesso ed un tempo totale $O(n \log n)$

L'heap non è altro che un particolare tipo di albero binario che può essere rappresentato senza lista a puntatori ma mediante array.

Un **albero binario** è un insieme finito di elementi detti **nodi**. Questo insieme è vuoto o consiste di un nodo detto **radice** e due alberi binari disgiunti detti **sottoalbero di destra** e **sottoalbero di sinistra**.

- Le radici di tali sottoalberi sono detti **figli** del nodo radice.
- C'è un **arco** da ciascun nodo a ciascuno dei suoi figli.
- Un nodo è chiamato **genitore (padre)** dei suoi figli.

Un albero binario ce lo immaginiamo come un insieme di nodi collegati tra di loro con dei collegamenti, ovvero con dei puntatori, però di fatto questi puntatori non è detto che debbano esistere fisicamente. Noi dobbiamo solo implementare il concetto di un insieme definito ricorsivamente in base proprio alla definizione ricorsiva di albero binario; ovvero un albero è:

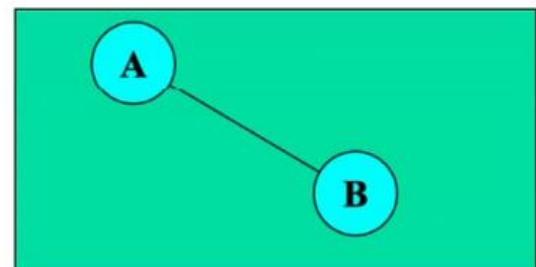
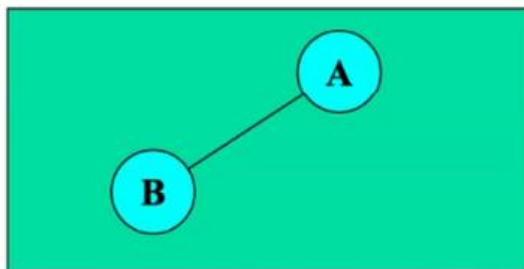
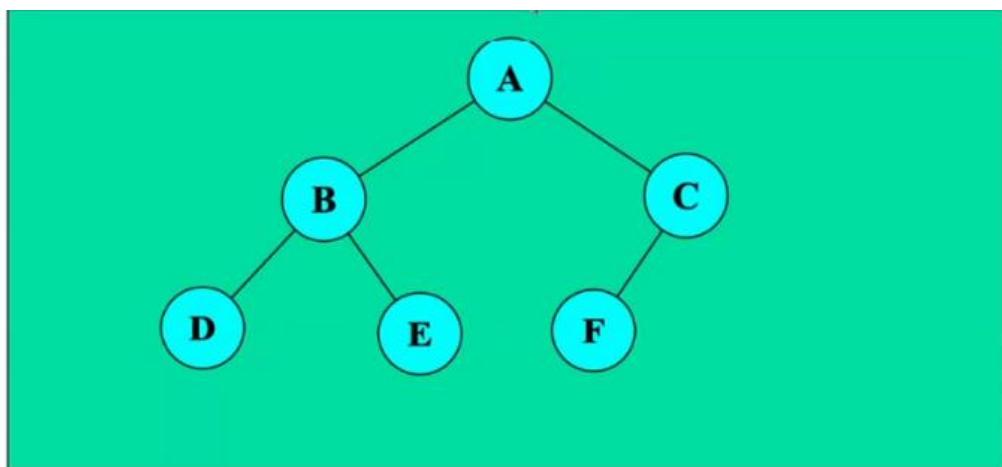
- un insieme vuoto;

oppure

- è composto da un nodo, detto radice, e da 2 alberi binari disgiunti detti sottoalbero di destra e sottoalbero di sinistra.

Possiamo immaginare già che la maggior parte delle funzioni definite sugli alberi binari saranno ricorsive ed opereranno sull'albero visitando un nodo e visitando poi il sottoalbero di destra e quello di sinistra, e ciò scaturisce proprio dalla definizione ricorsiva di albero binario.

Le radici dei sottoalberi sono dette figli del nodo radice. Ciascun nodo è collegato ai suoi figli per mezzo di archi ed un nodo è padre dei suoi figli.



Nell'esempio B è figlio sinistro di A ed è la radice del sottoalbero di sinistra, C è figlio destro di A ed è la radice del sottoalbero di destra. D ed E sono a loro volta degli alberi binari che hanno un sottoalbero di sinistra ed un sottoalbero di destra vuoti: vengono dette foglie dell'albero. Mentre i nodi che hanno almeno un figlio non vuoto, come B e C, vengono detti nodi interni all'albero. Anche i 2 insiemi di sotto sono alberi binari.

Noi tipicamente visiteremo un albero a partire dalla radice per arrivare fino ad un certo nodo facendo un percorso. Quindi un percorso di un albero è una sequenza di nodi tali che ogni elemento di questa sequenza è un figlio del nodo precedente nella sequenza.

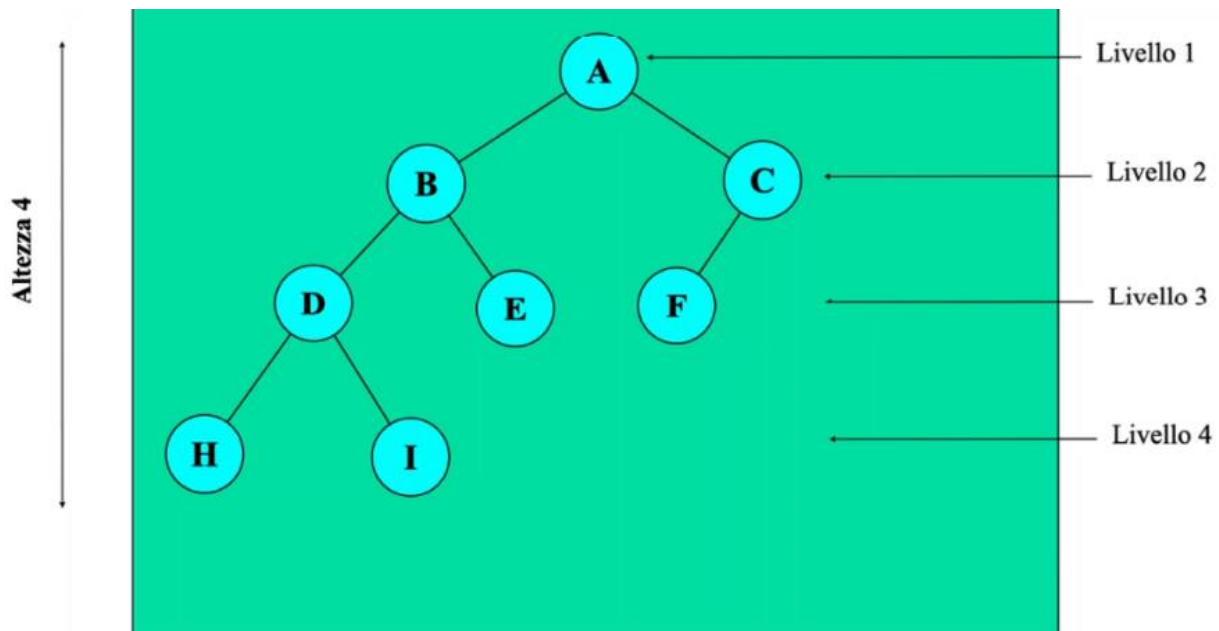
Se n_1, n_2, \dots, n_k è una sequenza di nodi dell'albero tali che n_i è il genitore di n_{i+1} , allora questa sequenza è detta un **percorso** da n_1 ad n_k . La **lunghezza** del percorso è $k-1$.

- Se c'e' un percorso da un nodo R ad un nodo M allora R è un **antenato** di M , ed M è un **discendente** di R .
- Tutti i nodi di un albero sono discendenti della radice

Diamo un altro po' di nomenclature:

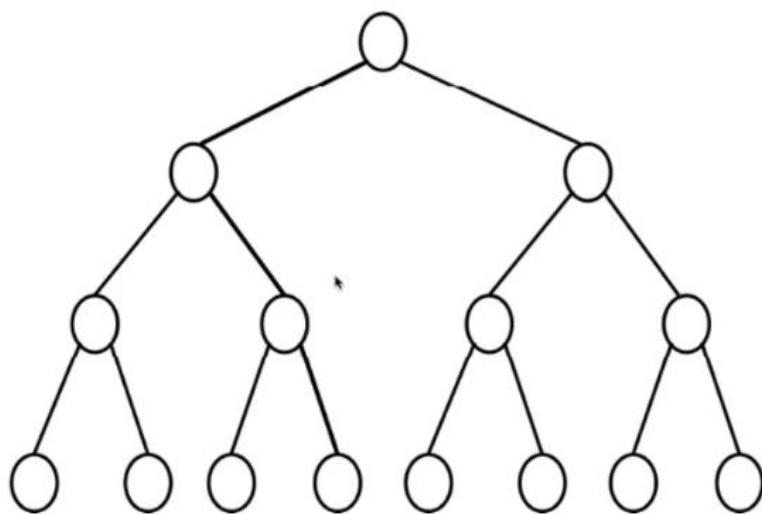
- La **profondità** di un nodo M dell'albero è la lunghezza del percorso dalla radice ad M .
- L'**altezza** di un albero è uno più la massima profondità dei nodi nodi dell'albero
- Tutti i nodi a profondità d sono al **livello** $d+1$ dell'albero.
- Un nodo **foglia** è un nodo che ha due figli vuoti.
- Un nodo **interno** è un nodo che ha almeno un figlio non vuoto.
- Il **grado** di un nodo è il numero di figli non vuoti.

Per "alberi binari" intendiamo il fatto che il grado può essere al più 2, ovvero ogni nodo può avere al più 2 figli non vuoti. Ovviamente ne esistono diversi.



Quello che vogliamo è avere alberi quanto più bilanciati possibili, perché altrimenti un albero molto sbilanciato equivarrebbe ad una lista.

Un albero binario di altezza k che ha esattamente $2^{k+1}-1$ nodi è detto **albero binario pieno di profondità k .**

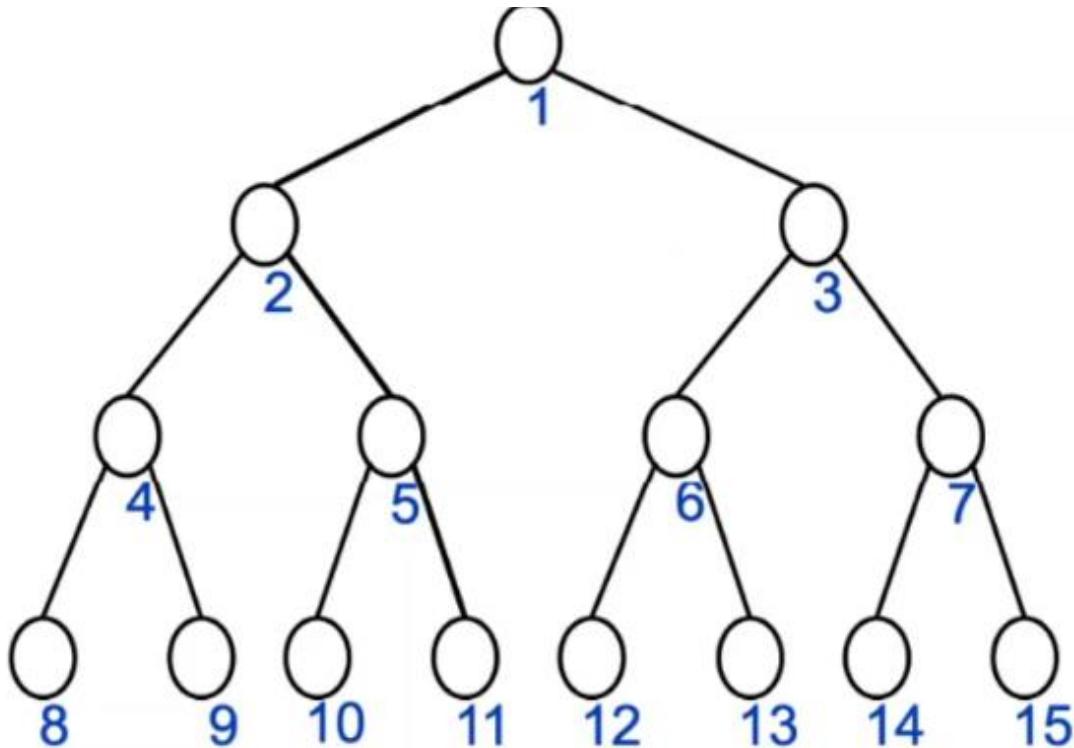


Quindi un albero binario con k livelli ha un numero di livelli massimo pari a:

$$1+2+4+8+16+32+\dots 2^k = \sum_{i=1}^{k+1} 2^{i-1} = \sum_{i=0}^k 2^i = \frac{1-2^{k+1}}{1-2} = 2^{k+1}-1.$$

In un albero binario pieno tutte le foglie si trovano all'ultimo livello.

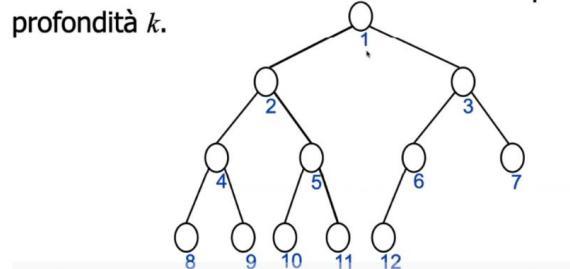
Se io so a priori il numero di nodi su livello, supponendo quindi che l'albero sia pieno, posso proprio evitare di rappresentare i collegamenti e rappresentare solamente i nodi numerati in un certo modo, ovvero numerati sequenzialmente.



Questo trucco ci permette di evitare di rappresentare esplicitamente la relazione di padre-figlio: infatti se l'albero binario è pieno, i figli del nodo 2 saranno il nodo 4 ed il nodo 5 mentre quelli del nodo 3 saranno il nodo 6 ed il nodo 7; e così via con tutti i nodi. Nodo padre → nodi figli = (nodo con numero doppio del padre, nodo successivo).

Sfruttando questa proprietà quindi possiamo utilizzare un array di 16 elementi per rappresentare un albero di altezza 4. Però non sempre siamo così fortunati da avere un albero binario pieno. Un albero binario pieno in generale ha un numero di nodi che è una potenza di 2 – 1 elemento. Però possiamo usare un albero binario completo, che è un albero non pieno necessariamente ma che adotta la numerazione di un albero binario pieno. Es:

Un albero binario con n nodi e profondità k è detto **completo** se i suoi nodi corrispondono ai nodi numerati da uno ad n nell'albero pieno di profondità k .



Quindi:

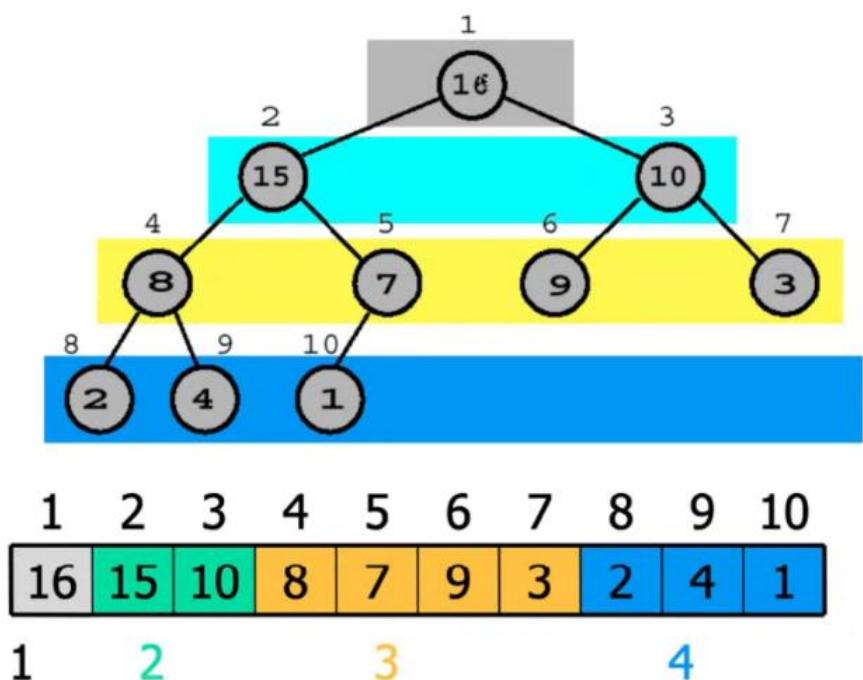
In un albero binario completo tutte le foglie si trovano su al più due livelli adiacenti

I nodi di un albero binario completo possono essere efficientemente **rappresentati in un array A** in cui il nodo numerato i è memorizzato in $A[i]$

Non c'è bisogno di alcun puntatore quindi, poiché il nodo i ha il figlio sinistro in posizione $2i$ e quello destro in posizione $2i+1$ (numerando i nodi da 1 ad n).

Es:

I nodi di un albero binario completo possono essere efficientemente **rappresentati in un array A** in cui il nodo numerato i è memorizzato in $A[i]$



Ovviamente, dato un albero completo di n nodi, la sua profondità massima k sarà tale che:

$$n \leq 2^k - 1$$



$$k = \lceil \log(n + 1) \rceil$$

Questo ci interessa perché se sviluppiamo algoritmi che al più dipendono dall'altezza massima dell'albero, allora questi avranno un tempo logaritmico.

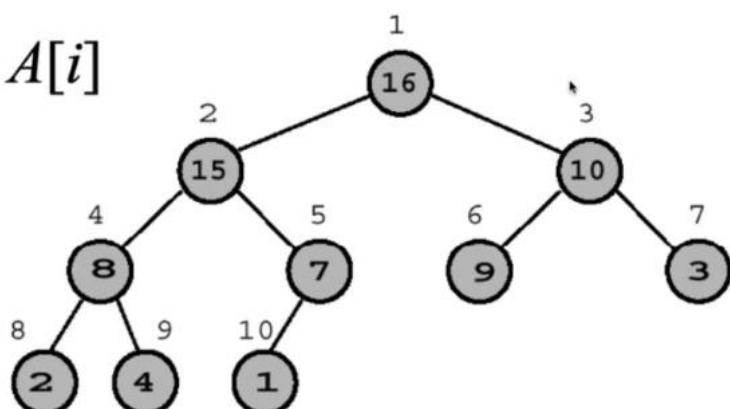
proprietà: Se un albero binario completo è rappresentato sequenzialmente in un array allora per ogni nodo con indice i , $1 \leq i \leq n$ abbiamo

- i. $\text{PARENT}(i)$ è a $\lfloor i/2 \rfloor$ se $i \neq 1$. Per $i=1$, i è la radice e non ha padre.
- ii. $\text{LCHILD}(i)$ è a $2i$ se $2i \leq n$. Se $2i > n$, allora i non ha figlio
- iii. $\text{RCHILD}(i)$ è a $2i+1$ se $2i+1 \leq n$. Se $2i+1 > n$, allora i non ha figlio destro

Ora definiamo cosa è un HEAP:

- Un **heap** è un albero binario completo con la proprietà che il valore in ciascun nodo è maggiore o uguale a quello dei suoi figli.

$$A[\text{PARENT}(i)] \geq A[i]$$



Nota: il massimo è nella radice

Quindi in un *heap* il valore di un nodo è \geq del valore dei suoi figli. Per il resto l'heap è semplicemente un albero binario completo. Ovviamente questo fa sì che il nodo alla radice dell'albero abbia il valore massimo.

Con l'heap cercheremo di trovare il massimo in tempo $O(\log(n))$ invece che $O(n\log(n))$. Questa struttura dati è la più furba per dare supporto alla funzione di ricerca del massimo.

Un heap lo si costruisce con una funzione detta **HEAPIFY** che assume di avere già un heap nel sottoalbero di sinistra ed in quello di destra e costruisce un heap unendo 2 sottoalberi.

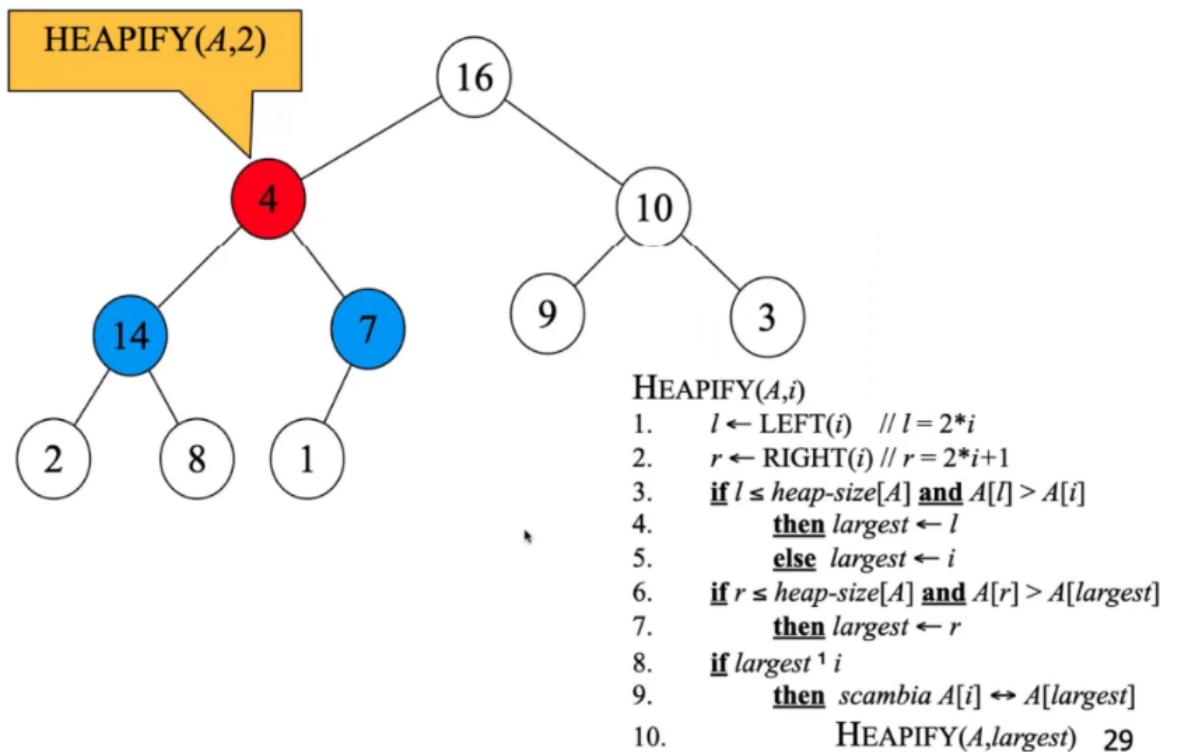
- i è un indice dell'array A
- Gli alberi binari radicati in $\text{LEFT}(i)$ e $\text{RIGHT}(i)$ sono heap
- Ma $A[i]$ potrebbe essere minore dei suoi figli, violando la proprietà heap
- La funzione **Heapify** rende A un heap muovendo $A[i]$ verso il basso finché la proprietà heap non è di nuovo soddisfatta.

Vediamo uno pseudocodice:

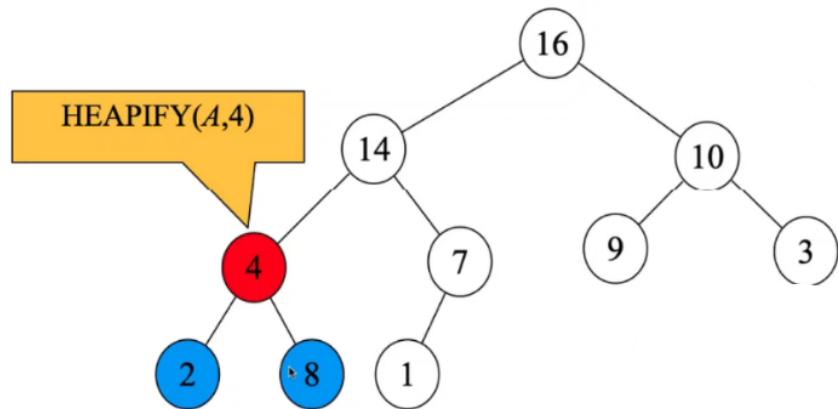
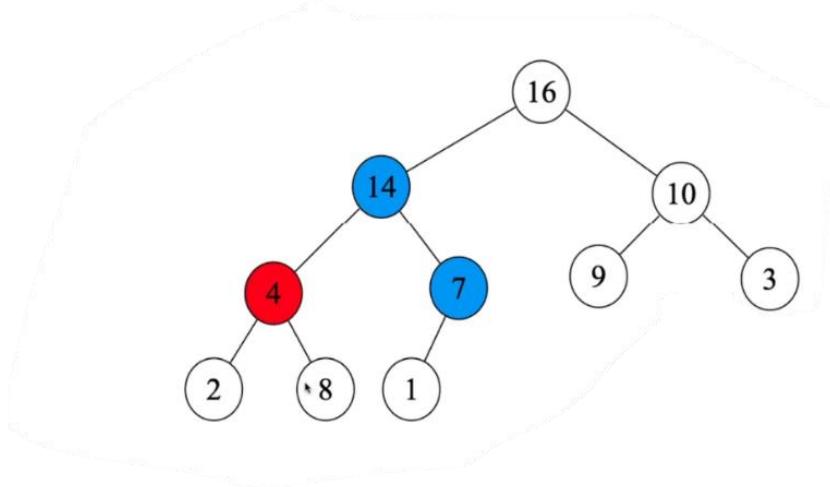
```
HEAPIFY( $A,i$ )
1.  $l \leftarrow \text{LEFT}(i)$  //  $l = 2*i$ 
2.  $r \leftarrow \text{RIGHT}(i)$  //  $r = 2*i+1$ 
3. if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$ 
4.   then  $\text{largest} \leftarrow l$ 
5.   else  $\text{largest} \leftarrow i$ 
6. if  $r \leq \text{heap-size}[A]$  and  $A[r] > A[\text{largest}]$ 
7.   then  $\text{largest} \leftarrow r$ 
8. if  $\text{largest} \neq i$ 
9.   then scambia  $A[i] \leftrightarrow A[\text{largest}]$ 
10.      HEAPIFY( $A,\text{largest}$ )
```

l è il figlio sinistro che sta in posizione $2i$, dove i è la posizione del nodo da “inserire” ordinatamente, ovvero il padre, ed r è la posizione del figlio destro, ovvero $2i+1$. Heapify prende in ingresso l’array e l’indice del nodo da inserire. In questa funzione viene calcolato il massimo tra nodo da inserire e figlio sinistro e si salva l’indice come indice massimo; si fa lo stesso con il figlio destro e l’indice massimo calcolato. A questo punto se il nodo inserito non è il più grande tra quello di sinistra e quello di destra, allora viene scambiato con il massimo appena calcolato e solo in questo caso si invoca ricorsivamente HEAPIFY sull’elemento in posizione *largest* perché in realtà dopo che si effettua lo scambio $A[i] <-> A[\text{largest}]$ nell’indice *largest* ci sarà proprio l’elemento da inserire in maniera ordinata, che in questa maniera viene fatto scendere sempre più sotto fino a quando non si finisce di invocare questa funzione.

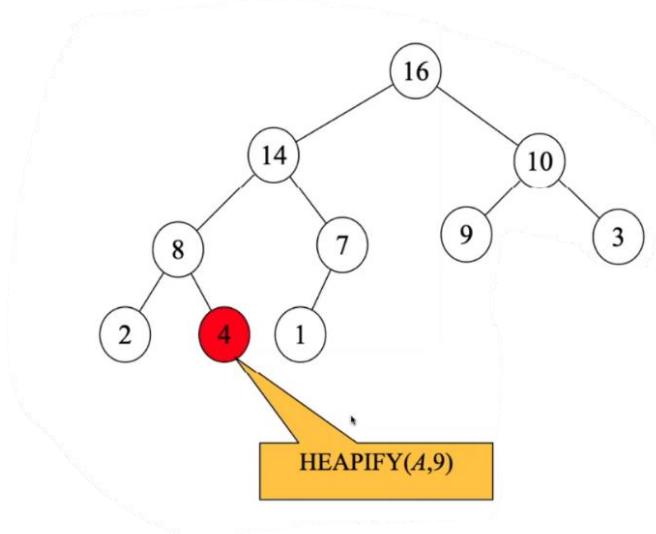
Vediamo un esempio.



Visto che $14 > 4$ e $14 > 7$ avviene uno scambio tra il nodo in posizione 2 e quello in posizione 4 che passa sopra. Il 4 scende e viene applicato nuovamente su di esso HEAPIFY.



8 e 2 sono *heap* ed il massimo è 8 e quindi si inverte 8 con 4 visto che 8 è anche maggiore di 4.



Ovviamente invocata heapify in posizione 9, questa non fa niente perché non c'è né figlio sinistro né quello destro. E quindi abbiamo ottenuto un heap a partire da 2 heap a cui abbiamo aggiunto l'elemento 4.

HEAPIFY quanto costa? Al più quanto l'altezza dell'albero, logaritmica

$$\rightarrow O(\log(n)).$$

HEAPIFY ci fa ottenere un heap a partire da 2 heap; come facciamo a costruire un heap? Lo facciamo con una funzione BUILD-HEAP:

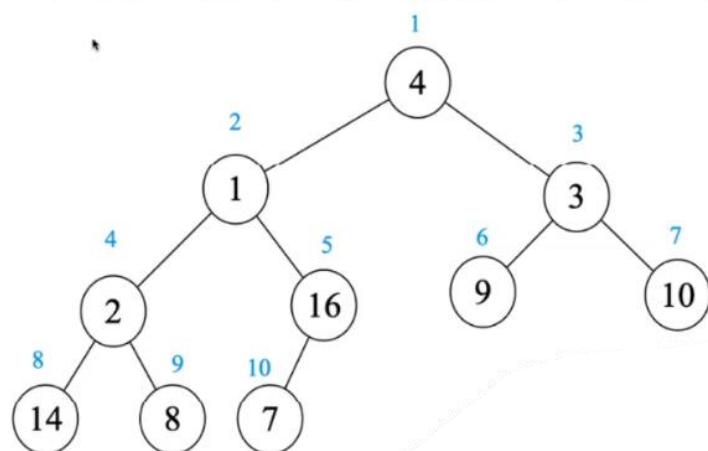
BUILD-HEAP(A)

1. $\text{heap-size}[A] \leftarrow \text{length}[A]$
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1
3. **do** HEAPIFY(A, i)

Partiamo da $n/2$ poiché dall'elemento successivo tutti gli elementi sono già heap (di un solo elemento). E quindi non si fa altro che applicare HEAPIFY sugli elementi che vanno da $n/2$ ad 1 (a ritroso).

Es: partiamo da un array completamente disordinato per rappresentare un heap.

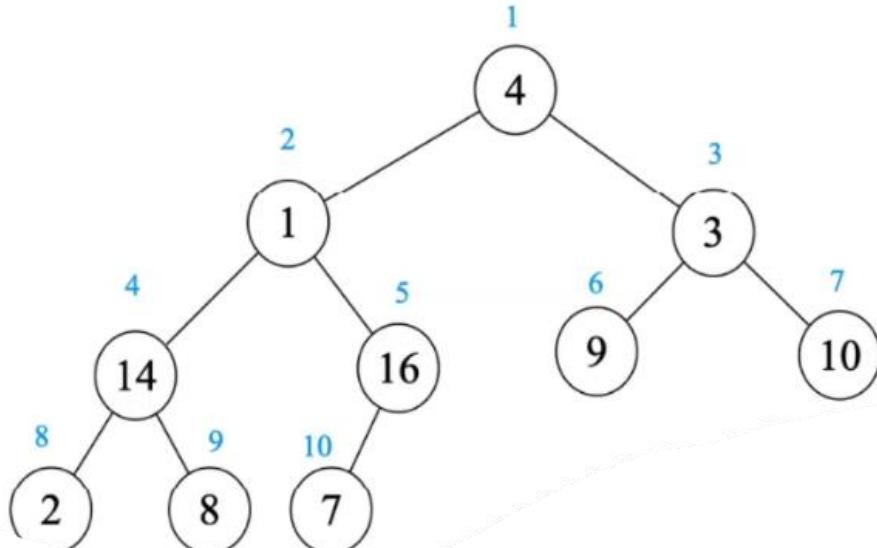
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



Heapify(A,5) non fa niente perché 16 è già > 7 .

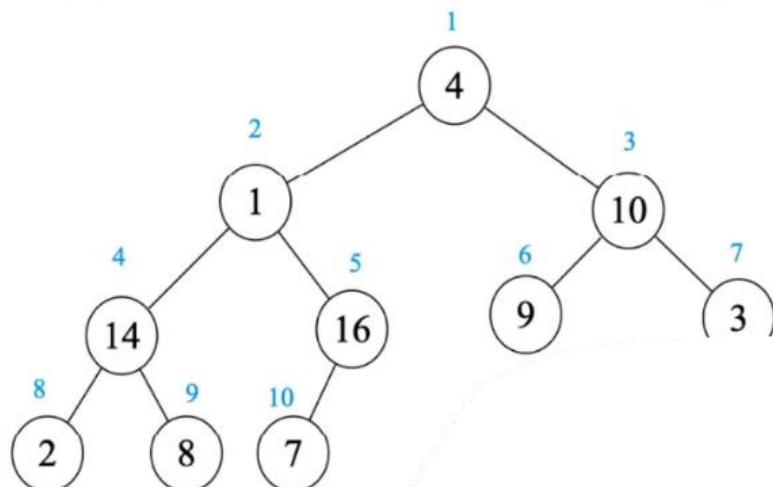
A questo punto viene richiamata Heapify(A,4) : viene scambiato 2 con 14. Questo fa sì che l'elemento in posizione 4 dell'array venga scambiato con quello in posizione 8.

1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7



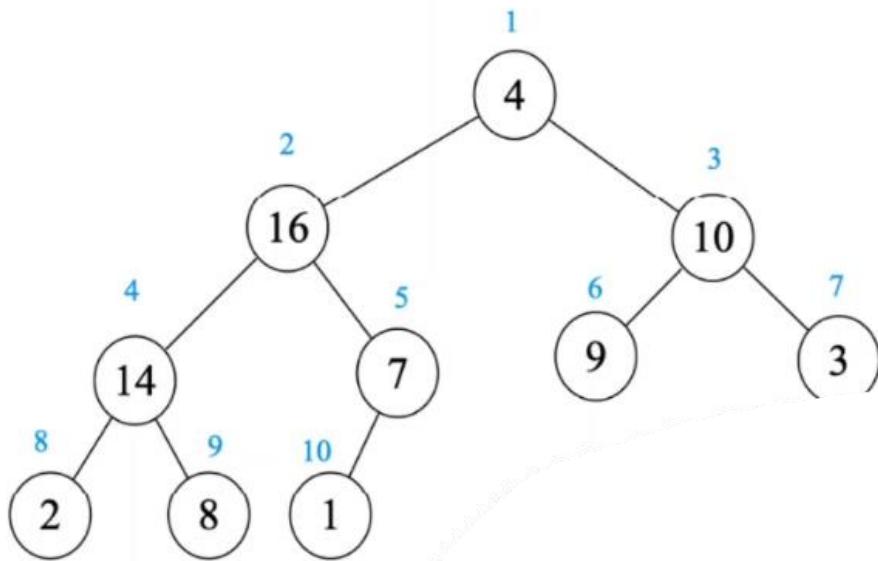
Si invoca poi Heapify sul nodo 3, contenente 3: $3 < 9$ e $3 < 10$, quindi viene scambiato il nodo 3 con il nodo 7 e il valore 10 passa sopra in posizione 3. Allora nell'array in posizione 3 ci va 10, in posizione 7 ci va 3.

1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7



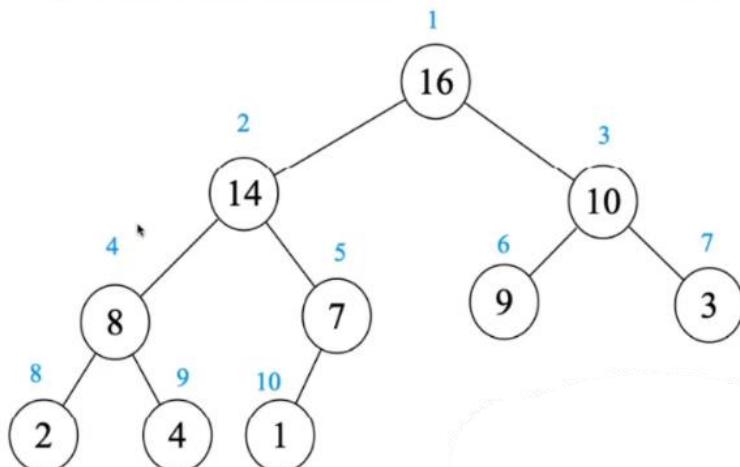
Infine si fa heapify sul nodo 2: $7 < 14 < 16$ e quindi si scambia 7 con 16 → elemento in posizione 2 passa in posizione 5 e viceversa. Ma a questo punto viene reinvocata heapify dalla stessa heapify sull'elemento 5, contenente ora 1, che viene scambiato con quello 10, contenente 7. Quindi alla fine 1 va in posizione 10, 16 va in posizione 2, 7 va in posizione 5.

1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1



Infine si fa Heapify sul nodo 1: 4 viene scambiato con 16, viene poi confrontato con 14 e 7 e quindi scende ancora, scambiato con 14, e a questo punto viene scambiato con 8 scendendo in fondo.

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Ad una prima analisi sembrerebbe che il tutto costi

$$n \log(n),$$

poiché si hanno n chiamate ad heapify che ha costo $\log(n) \rightarrow O(n \log(n))$.

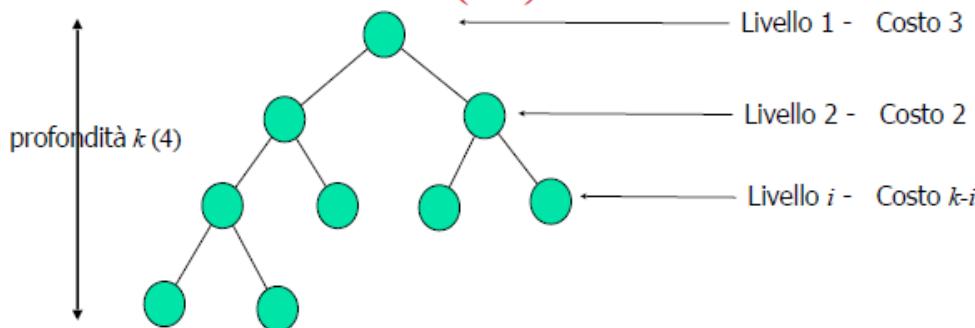
Tuttavia gran parte delle volte viene un costo minore, poiché gran parte delle volte Heapify è invocata su nodi che stanno in basso nell'albero e quindi da un'analisi più attenta il tempo di Build-Heap è $O(n)$.

$O(n)$

Infatti: (*analisi del build-heap*)

Tempo di esecuzione

- Numeriamo da 1 a $k = \lceil \log(n+1) \rceil$ i livelli dell'albero binario completo
- Al livello i ci sono al più 2^{i-1} nodi
- Il tempo di esecuzione di HEAPIFY per i nodi a livello i è $O(k-i)$



47

$$\sum_{1 \leq i \leq k} 2^{i-1}(k-i) = \sum_{1 \leq i \leq k-1} i2^{k-i-1}$$

Con la sostituzione $k-i \rightarrow i$

$$\sum_{1 \leq i \leq k} 2^{i-1}(k-i) = \sum_{1 \leq i \leq k-1} i2^{k-i-1} \leq n \sum_{1 \leq i \leq k-1} i/2^i$$

Perché $n \geq 2^{k-1}$

$$\sum_{1 \leq i \leq k} 2^{i-1}(k-i) = \sum_{1 \leq i \leq k-1} i2^{k-i-1} \leq n \sum_{1 \leq i \leq k-1} i/2^i < 2n$$

Perché (esercizio) $\sum_i i/2^i < 2$

Dove l'ultimo passaggio è dovuto ad un risultato noto sulle serie.

Pensiamo ad una cosa: trovare il massimo in un insieme costa almeno $O(n)$ perché vanno visitati almeno una volta tutti gli elementi; tuttavia ora non solo abbiamo messo il massimo in prima posizione, ma abbiamo anche costruito un albero con proprietà *heap* in tempi $O(n)$ e quindi il nostro array soddisfa la proprietà heap. Avendo un heap, inserire o estrarre un elemento dall'heap è semplice, poiché dobbiamo attraversare l'heap, ma attraversare l'heap vuol dire passare al più dalla radice fino ad una foglia e questo ci costa ovviamente $\log(n)$.

Per questo allora la priority queue nel C++ è basata su una struttura heap: il top ovviamente è $O(1)$ perché so dove si trova, e sia push che pop, mantenendo la priority queue come un heap, richiedono al più $\log(n)$.

HEAP SORT

Sapendo costruire un heap possiamo implementare l'**heap sort**, algoritmo di ordinamento. Questo algoritmo ha come proprietà il fatto che sarà $O(n\log(n))$, ma ha sia tempo medio che tempo peggiore $O(n\log(n))$, mentre per il quick sort invece il caso peggiore era quello di un array già ordinato ed in quel caso la complessità era $O(n^2)$, e questo è il tallone d'achille del quick sort. Quindi l'heap sort:

- non richiede memoria aggiuntiva (a differenza del *merge sort*);
- ha complessità $O(n\log(n))$;
- non ha un tempo peggiore $O(n^2)$ (a differenza del quick sort). Sia worst che average che best case sono tutti $O(n\log(n))$.

Allora l'*heap sort*, algoritmo di ordinamento di vettori, è fatto così:

Heap sort si basa sull'iterazione dei seguenti passi:

- 1. Trasforma il vettore in uno heap
- 2. Scambia il primo elemento (che è sicuramente il più grande) con l'ultimo
- 3. Riduci la dimensione dello heap di 1
- 4. Ripeti da 1.

Allora l'idea è semplicemente questa:

una volta che il vettore è trasformato in heap, avrà all'inizio l'elemento massimo e quindi lo si mette alla fine scambiandolo con l'ultimo elemento. A questo punto si riduce la dimensione dell'heap di 1, lasciando l'ultimo elemento, il massimo, da parte e si ripete il procedimento. Per trasformare il vettore in un heap invochiamo la funzione *heapify*. *Heapify* ci costa $\log(n)$ e quindi questo algoritmo ci viene al più $O(n \log(n))$.

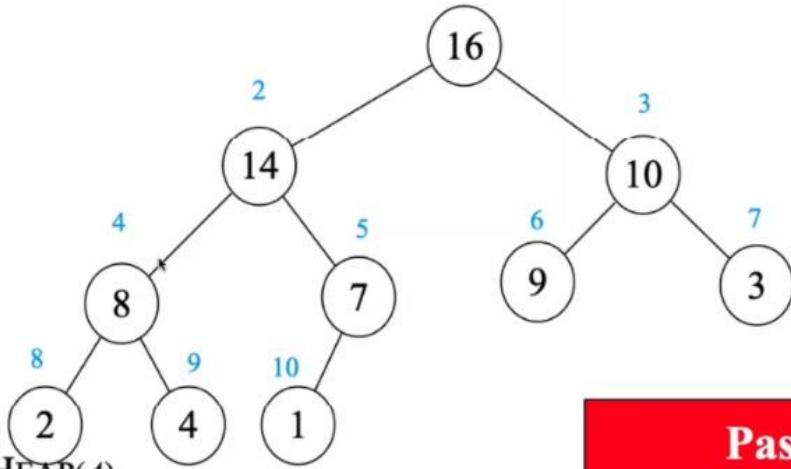
Vediamo uno pseudo-codice:

HEAPSORT(A)	Analisi
1. BUILD-HEAP(A)	$O(n)$
2. <u>for</u> $i \leftarrow \text{length}[A]$ <u>downto</u> 2	n volte
3. <u>do</u> scambia $A[1] \leftrightarrow A[i]$	$O(1)$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$	$O(1)$
5. HEAPIFY($A, 1$)	$O(\log n)$

Il tempo totale di HEAPSORT è
 $O(n \log n)$

Vediamo un ESEMPIO:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

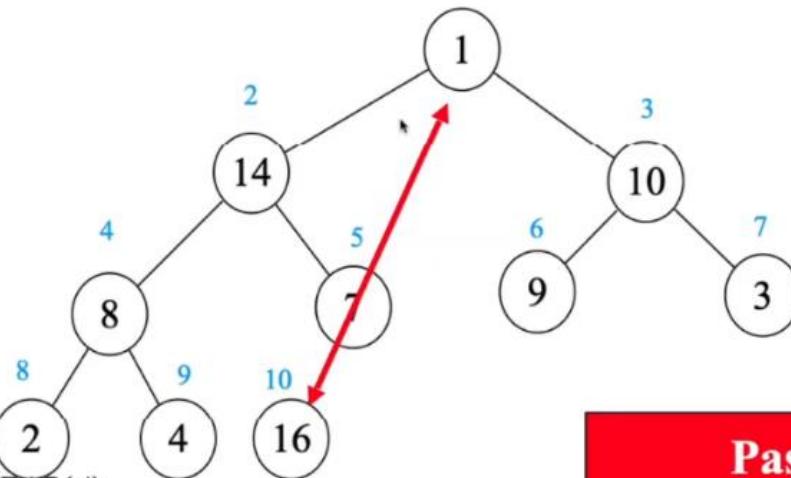


HEAPSORT(A)

1. BUILD-HEAP(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
3. **do** scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 0
si costruisce
l'heap

1	2	3	4	5	6	7	8	9	10
1	14	10	8	7	9	3	2	4	16



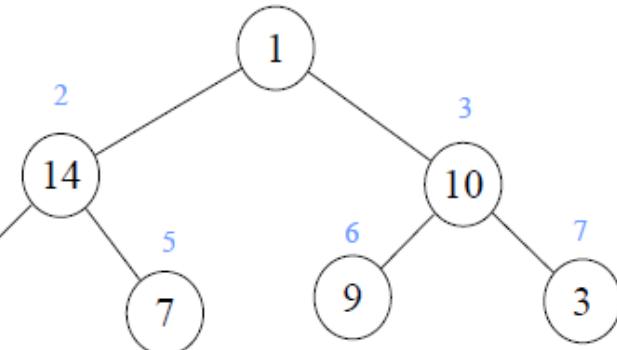
HEAPSORT(A)

1. BUILD-HEAP(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
3. **do** scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 1
si scambiano $A[1]$
Con $A[10]$

—

1	2	3	4	5	6	/	8	9	10
1	14	10	8	7	9	3	2	4	16



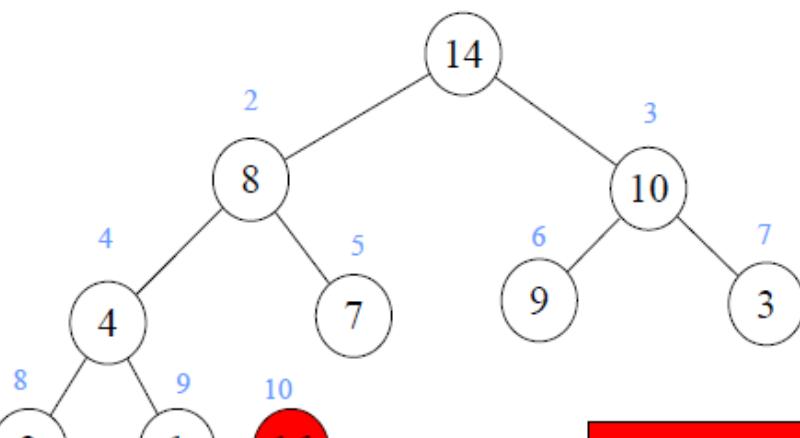
HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 2
la parte da ordinare
si riduce di uno

—

1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16

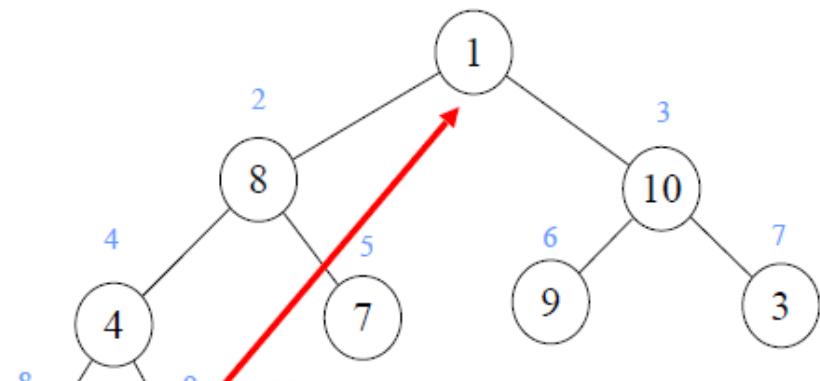


HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 3
Si rispristina la
proprietà heap

1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	3	2	14	16

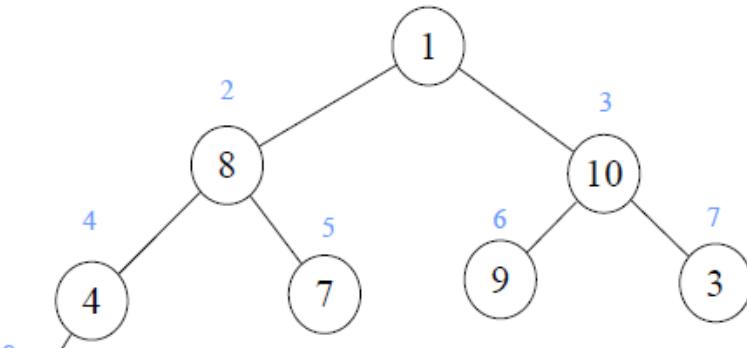


HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 4
si cambiano $A[1]$
Con $A[9]$

1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	3	2	14	16

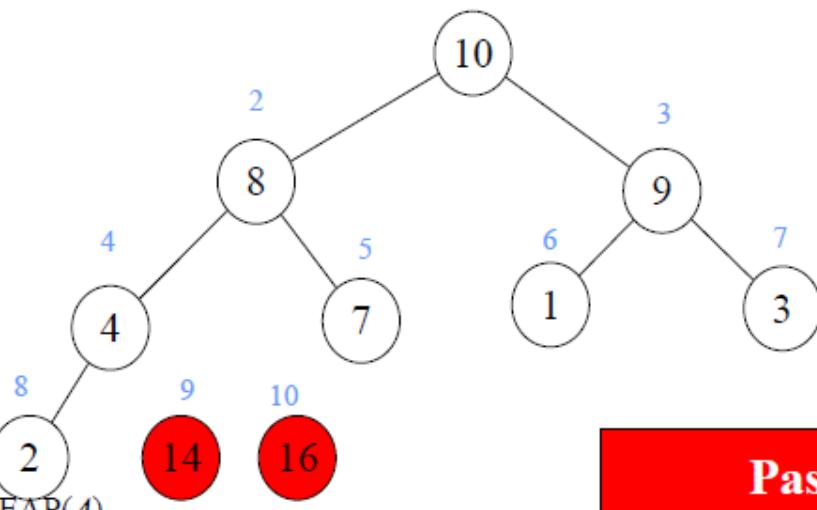


HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 5
la parte da ordinare
si riduce di uno

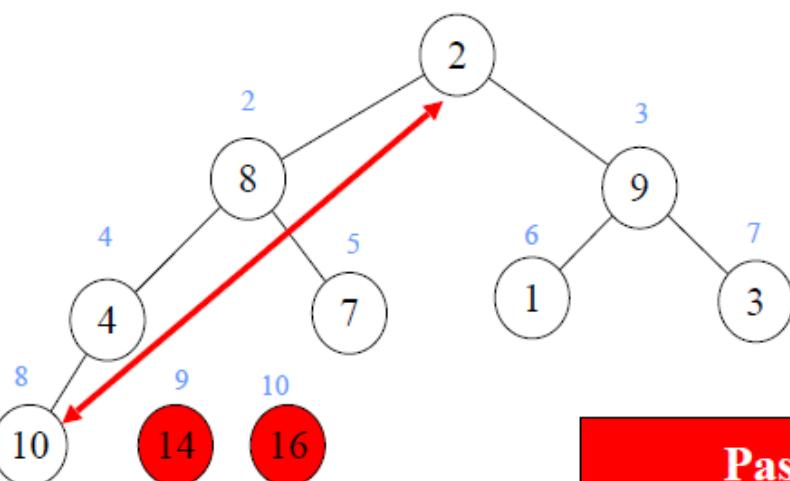
1	2	3	4	5	6	/	8	9	10
10	8	9	4	7	1	3	2	14	16



HEAPSORT(A)
1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 6
si risprista la proprietà heap

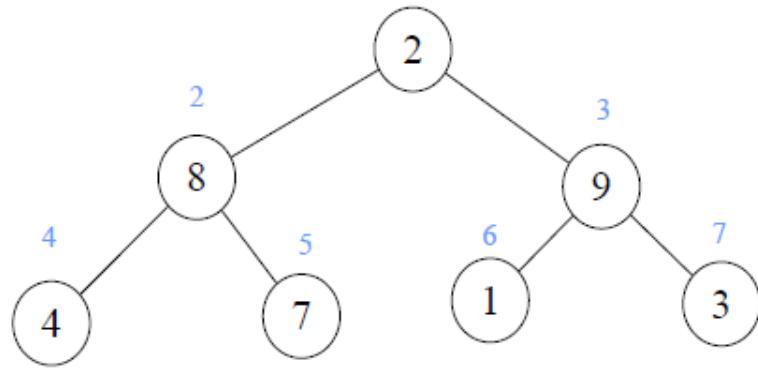
1	2	3	4	5	6	7	8	9	10
2	8	9	4	7	1	3	10	14	16



HEAPSORT(A)
1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 7
si scambiano $A[1]$ con $A[8]$

1	2	3	4	5	6	7	8	9	10
2	8	9	4	7	1	3	10	14	16

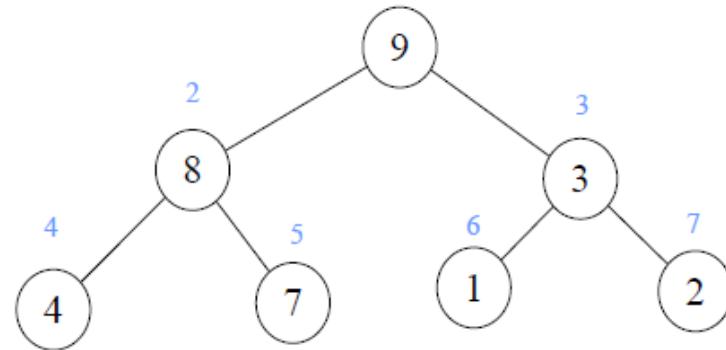


HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 8
la parte da ordinare
si riduce di uno

1	2	3	4	5	6	/	8	9	10
9	8	3	4	7	1	2	10	14	16



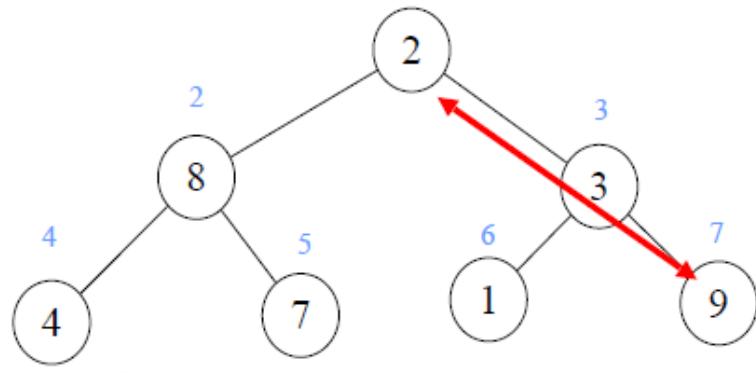
HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 9
si ripristina la
proprietà heap

■ ■ ■

1	2	3	4	5	6	7	8	9	10
2	8	3	4	7	1	9	10	14	16



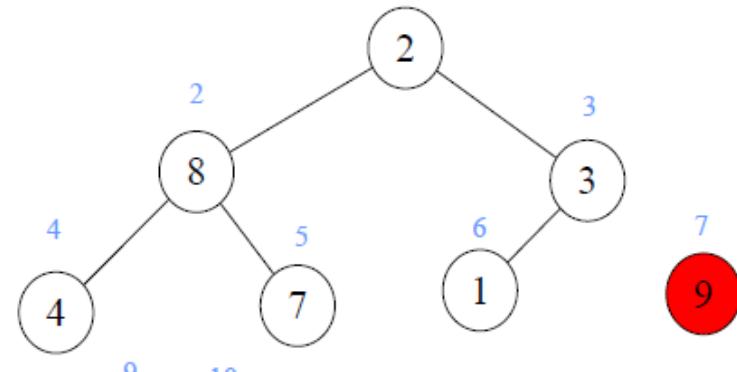
HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

**Passo 10
si scambiano $A[1]$
con $A[7]$**

■ ■ ■

1	2	3	4	5	6	7	8	9	10
2	8	3	4	7	1	9	10	14	16

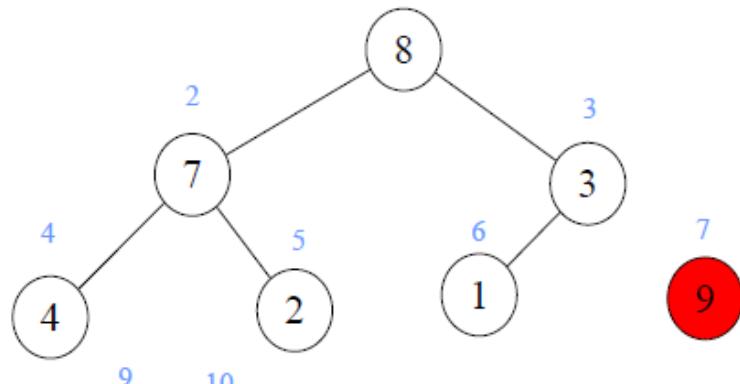


HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

**Passo 11
la parte da ordinare
si riduce di uno**

1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16

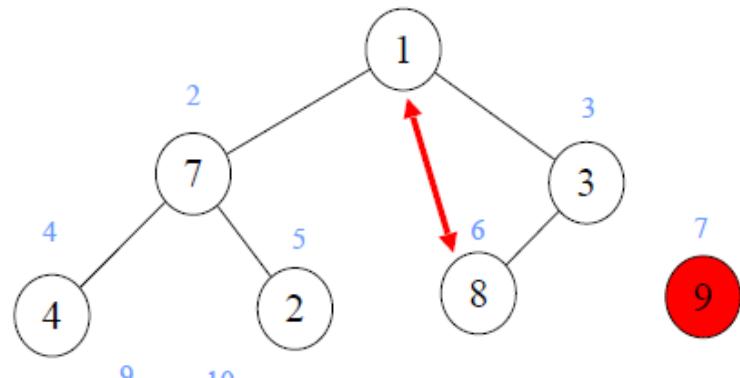


HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 12
si rispristina la
proprietà heap

1	2	3	4	5	6	7	8	9	10
1	7	3	4	2	8	9	10	14	16

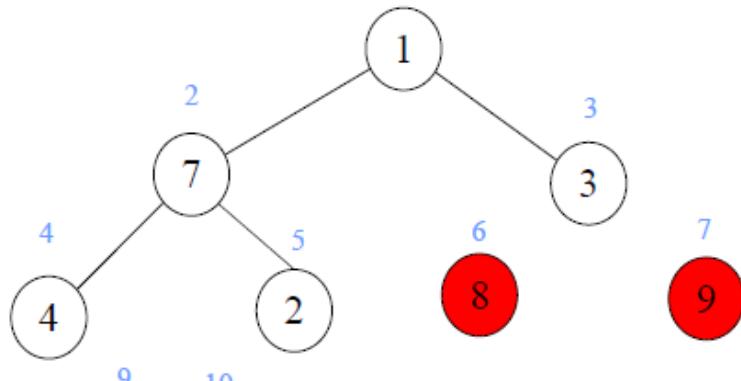


HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 13
si scambiano $A[1]$
con $A[6]$

1	2	3	4	5	6	7	8	9	10
1	7	3	4	2	8	9	10	14	16

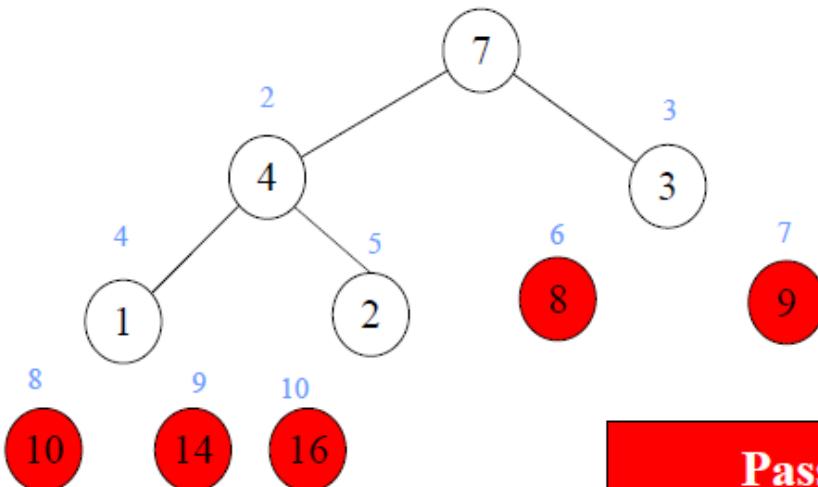


HEAPSORT(A)

1. BUILD-HEAP(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
3. do scambia $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
5. HEAPIFY($A, 1$)

Passo 14
la parte da ordinare
si riduce di uno

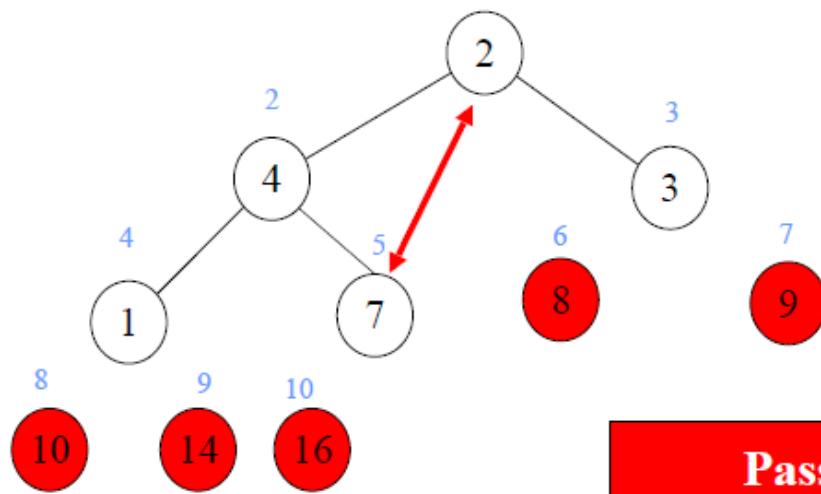
1	2	3	4	5	6	/	8	9	10
7	4	3	1	2	8	9	10	14	16



Passo 15
si ripristina la
proprietà heap

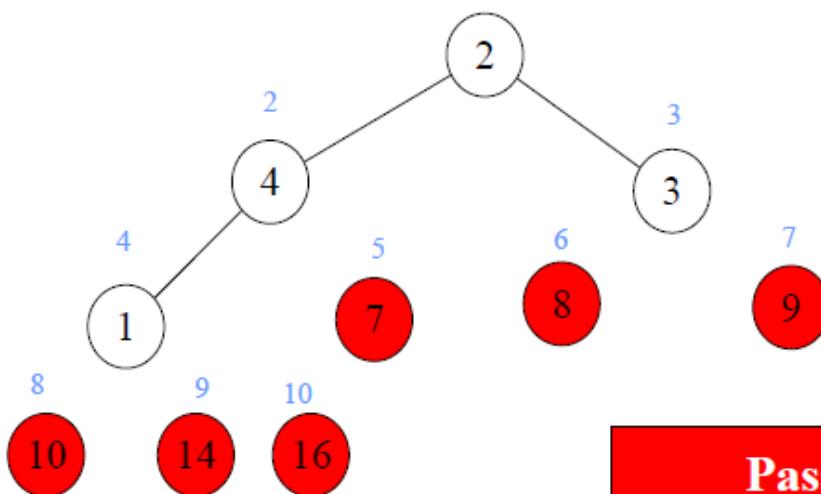
■ ■ ■

1	2	3	4	5	6	7	8	9	10
2	4	3	1	7	8	9	10	14	16

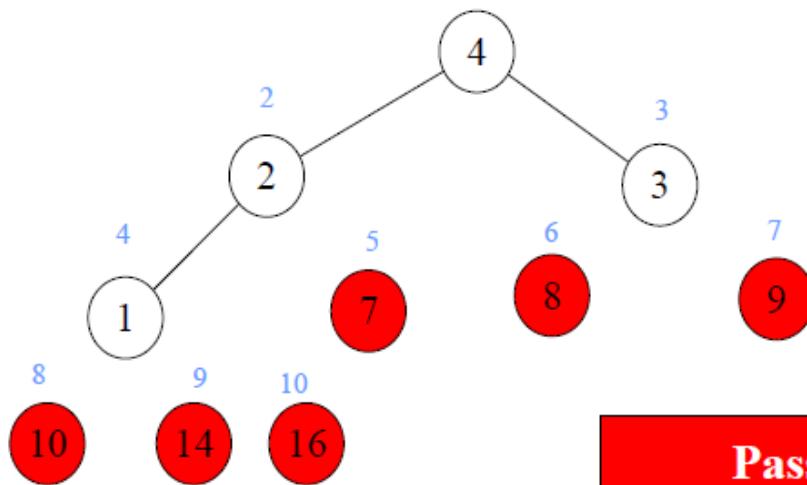


■ ■ ■

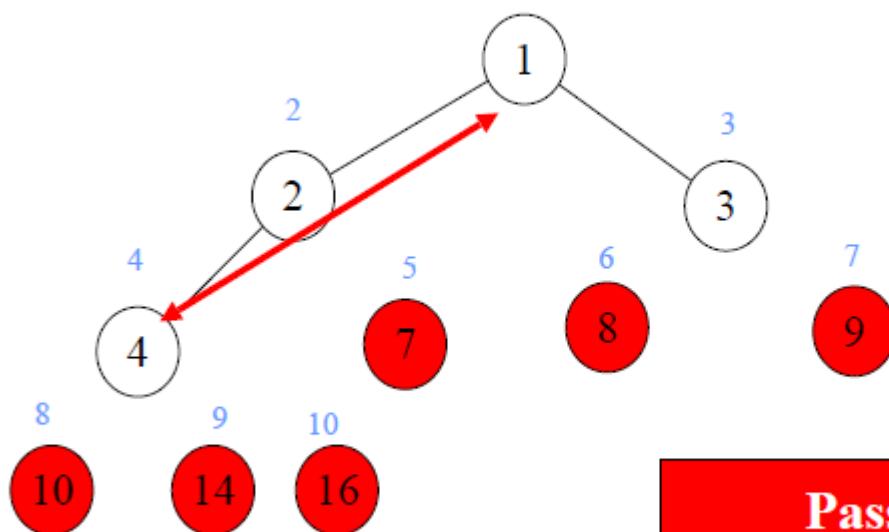
1	2	3	4	5	6	7	8	9	10
2	4	3	1	7	8	9	10	14	16



—



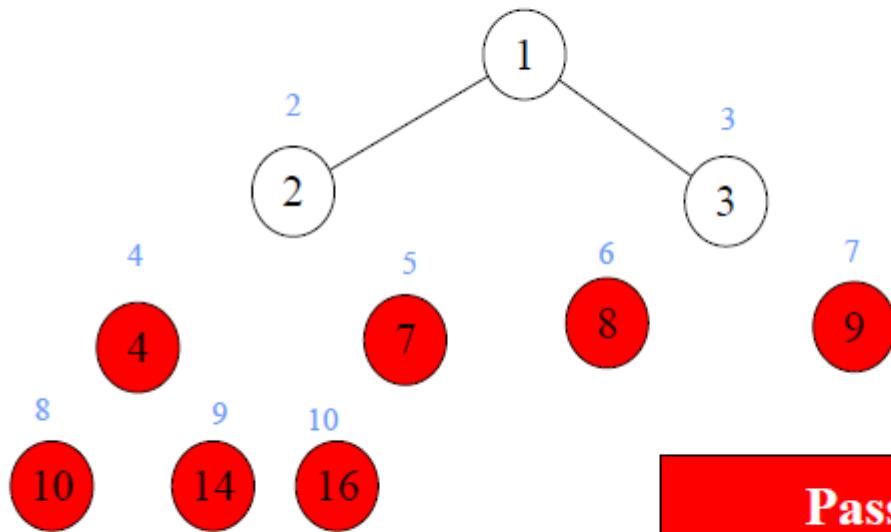
Passo 18
si ripristina la
proprietà heap



Passo 19
si scambia A[1]
con A[4]

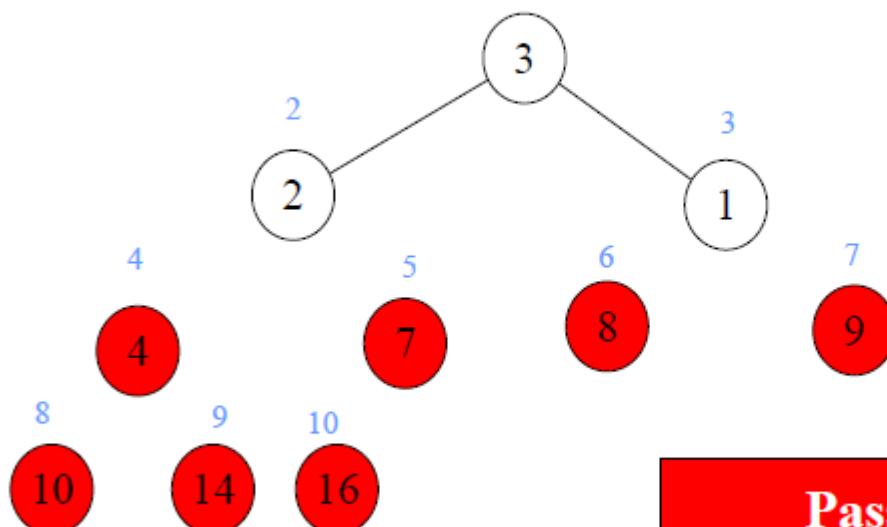
■

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



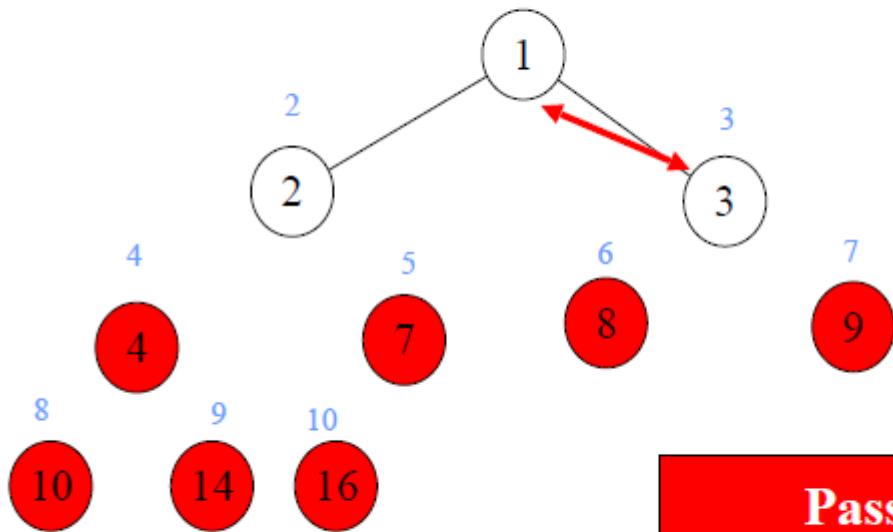
Passo 20
la parte da ordinare
si riduce di uno

1	2	3	4	5	6	7	8	9	10
3	2	1	4	7	8	9	10	14	16



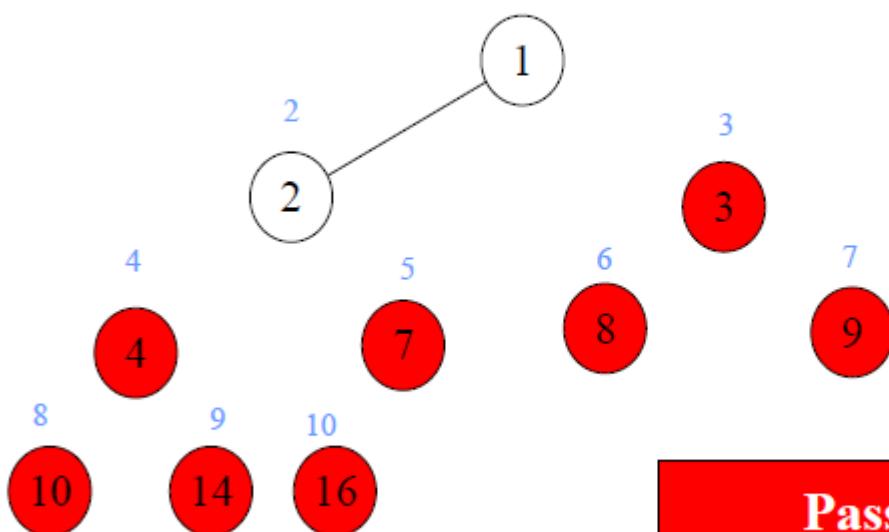
Passo 21
si ripristina
la proprietà heap

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



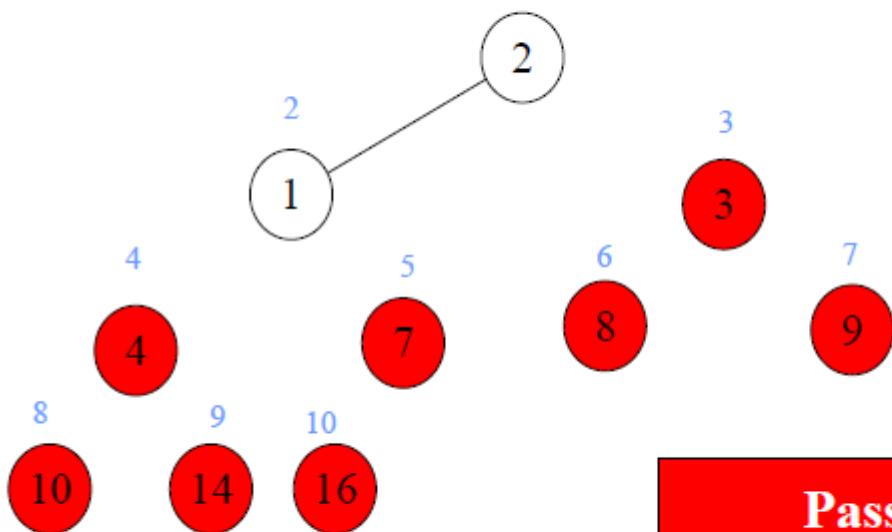
Passo 22
si scambiano A[1]
con A[3]

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



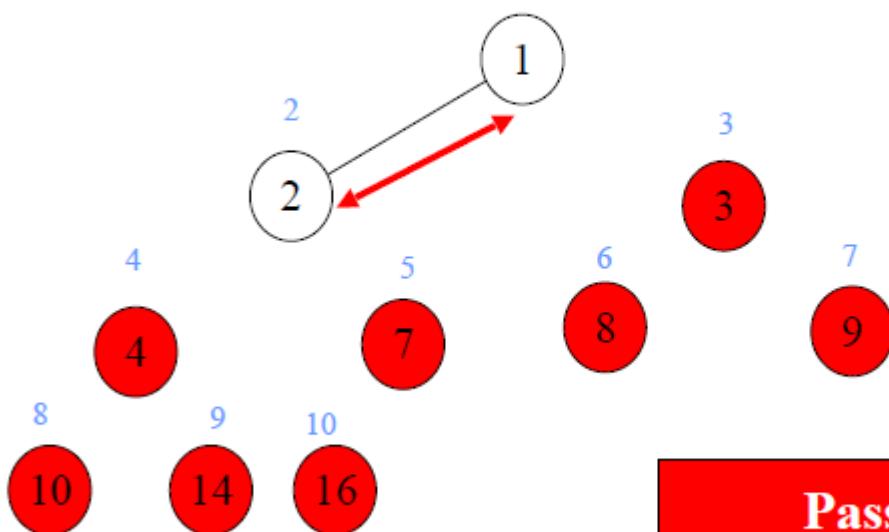
Passo 23
la parte da ordinare
si riduce di uno

1	2	3	4	5	6	7	8	9	10
2	1	3	4	7	8	9	10	14	16



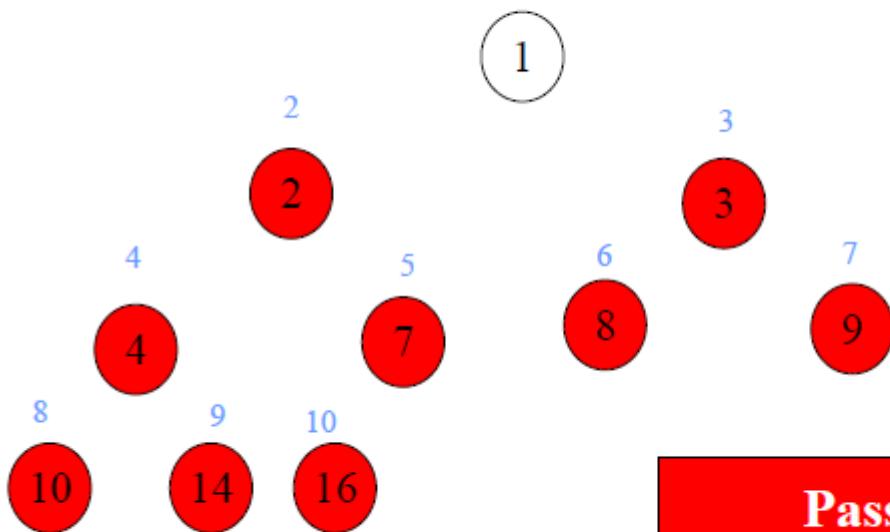
Passo 24
si ripristina la proprietà heap

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



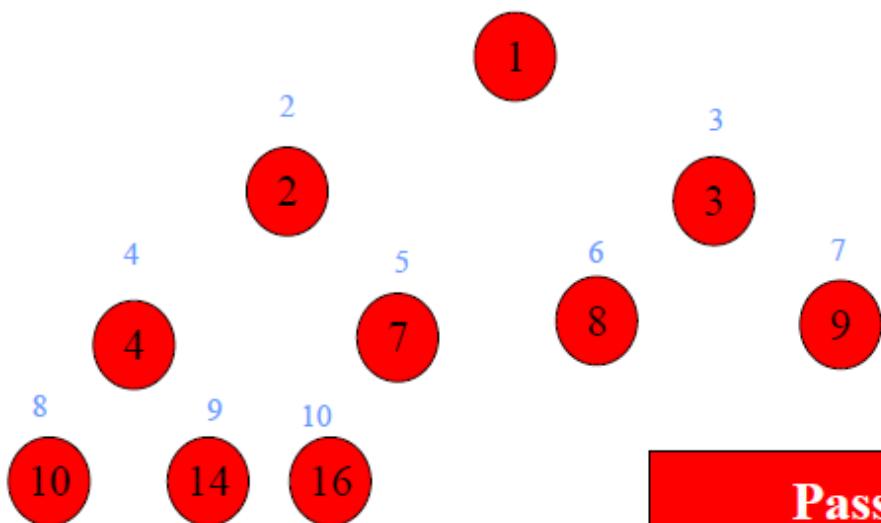
Passo 25
si scambiano A[1]
con A[2]

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



Passo 26
la parte da ordinare
si riduce di uno

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



Passo 27
L'array è ora
ordinato

Allora, vista la struttura dati heap e visto l'ordinamento basato su di essa, possiamo costruirci la nostra *priority queue* definendoci le operazioni. In cosa consiste effettuare il push?

PUSH

HEAP-INSERT(A , key)

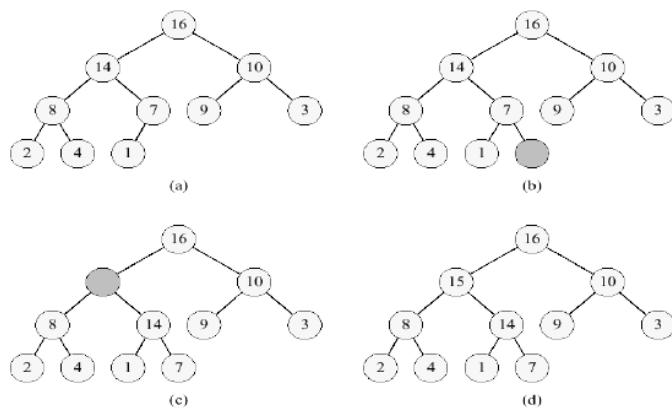
1. $heap\text{-size}[A] \leftarrow heap\text{-size}[A] + 1$
2. $i \leftarrow heap\text{-size}[A]$
3. **while** $i > 1$ **and** $A[\text{PARENT}(i)] < key$
4. **do** $A[i] \leftarrow A[\text{PARENT}(i)]$
5. $i \leftarrow \text{PARENT}(i)$
6. $A[i] \leftarrow key$

Innanzitutto si aumenta la dimensione dell'heap di 1 per fare spazio all'elemento. Poi si inserisce l'elemento nell'ultima posizione e a questo punto con un while si sposta l'elemento sempre più indietro per inserirlo in ordine, come nell'insertion sort, ma non fino all'elemento $i-1$ bensì al $\text{PARENT}(i)$. Quindi il nuovo elemento lo confronto con il suo parent e se il genitore è più piccolo di questo elemento allora effettuo lo scambio.

Heap-Insert sale su mentre *Heapify* scende giù attraverso l'albero, però anche il primo come il secondo al massimo si può muovere di $\log(n)$

→ inserimento con tempi $O(\log(n))$.

ESEMPIO: supponiamo di inserire un elemento con chiave 15.



Ora vediamo il pop.

POP

HEAP-EXTRACT-MAX(A)

1. **if** $heap\text{-size}[A] < 1$
2. **then error** “heap underflow”
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[heap\text{-size}[A]]$
5. $heap\text{-size}[A] \leftarrow heap\text{-size}[A] - 1$
6. HEAPIFY($A, 1$)
7. **return** max

Il pop estrae il massimo. L'estrazione del massimo è $O(1)$, però va ricostruita la proprietà heap. Quindi si prende il massimo, che sta in prima posizione, e lo si mette da qualche parte (in max). A questo punto si sposta in prima posizione l'oggetto che stava in ultima posizione e si riduce la dimensione dell'heap da n ad $n-1$. Infine si invoca *heapify* per ricostruire l'heap a partire dal primo elemento.

Ovviamente il TOP costa $O(1)$.



Esercizio (da Fare!)

- Implementare in C++ i template delle funzioni:
 - Heapify
 - Build-Heap
 - HeapSort

LEZIONE 28

Vediamo un'altra struttura dati.

BINARY SEARCH TREE (BST opp. ABR)

una struttura dati concreta che ci serve per mantenere dei dizionari ordinari. Un dizionario non è altro che una struttura dati con il supporto all'inserimento (push), alla rimozione (pop) ed alla ricerca.

- **Dizionario ADT** – un insieme dinamico con i metodi:
 - **SEARCH(S, k)** – un metodo di query che restituisce un riferimento x ad un elemento tale che $x.key = k$
 - **INSERT(S, x)** – un metodo modificatore che aggiunge l'elemento puntato da x ad S
 - **DELETE(S, x)** – un metodo modificatore che elimina da S l'elemento puntato da x
- Ogni elemento ha una parte **un'informazione chiave** ed una parte di **informazioni satellite**

Quindi ogni elemento possiede un'informazione chiave ed una satellite e vogliamo effettuare la ricerca in base la chiave.

Allora ad esempio per la classe Employee possiamo mantenere un dizionario ordinato degli impiegati effettuando una ricerca in base al codice dell'impiegato, inserendo quindi come altra informazione un intero per il codice.

In C++ il dizionario della STL è implementato con delle mappe, ovvero con la struttura dati *map* che può essere ordinata o meno. Per default nel C++ le mappe sono ordinate e quindi il dizionario ha supporto alla ricerca ma anche all'ordinamento. Il supporto all'ordinamento è dato da **MIN(S)** e **MAX(S)**, operazioni supportate dalle code a priorità, e da **PREDECESSOR(S, k)** e **SUCCESSOR(S, k)**; queste ultime non sono supportate dalle code a priorità, ed è questo il motivo per cui non vengono usate le code a priorità per implementare i dizionari. Capiamo come *predecessor* e *successor* ci permettano, unitamente a *min* e *max*, di effettuare l'ordinamento: possiamo trovare il minimo e poi dal minimo il successore del minimo, poi il successore del successore del minimo e così via, fino ad ottenere la sequenza ordinata.

Come implementiamo queste funzioni? Come vogliamo. Però al solito c'è sempre qualche struttura dati che conviene rispetto alle altre, ed in questo caso il BST si vede che riesce a fornire supporto abbastanza efficiente a queste funzioni. Ad esempio invece l'heap sarebbe un po' scomodo perché non conosciamo predecessore e successore in maniera immediata, sappiamo solo dove sta il massimo; quindi dovremmo comunque scorrere tutto l'heap.

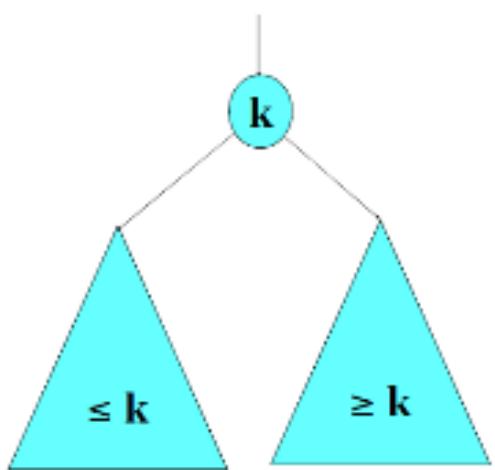
Un albero binario già abbiamo detto cos'è, poiché abbiamo visto l'heap che altro non è che un albero binario particolare.

Cos'è un **binary search tree** invece?

Una albero binario di ricerca è un albero binario T tale che

- Ciascun nodo interno memorizza un elemento (k, e) di un dizionario
- Le chiavi memorizzate nel **sottoalbero di sinistra** di un nodo con chiave k sono **minori o uguali di k**
- Le chiavi memorizzate nei nodi del **sottoalbero di destra** di un nodo con chiave k sono **maggiori o uguali di k**

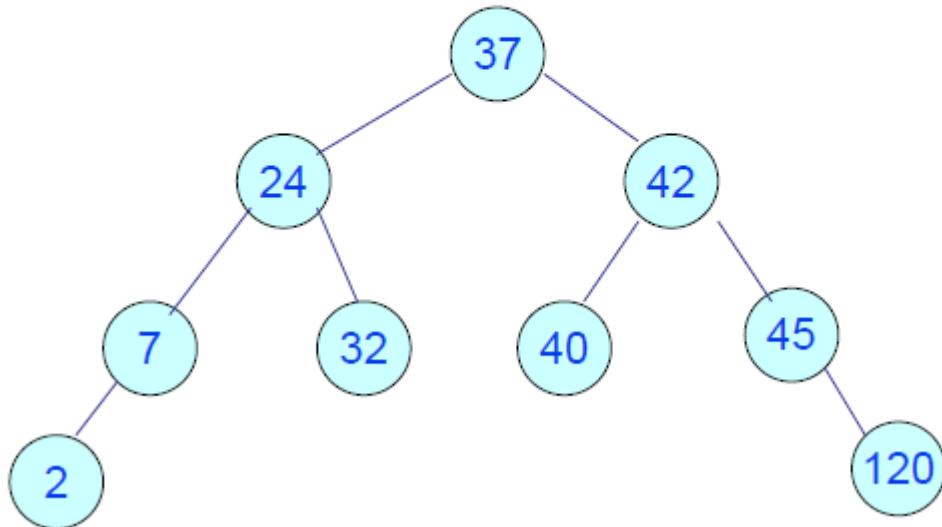
Quindi in un albero binario di ricerca ogni nodo memorizza oltre ad un elemento anche una chiave. I nodi sono disposti in base alle chiavi, come scritto sopra: quelli nel sottoalbero di sinistra di un certo nodo radice hanno una chiave minore o uguale di quella del nodo radice, quelli del sottoalbero di destra hanno una chiave maggiore o uguale di quella del nodo radice.



Perché ciò è efficiente? Basta tenere a mente la ricerca binaria già vista, perché il ragionamento è lo stesso. Se sto cercando una chiave, vado nella radice e se la chiave è nella radice allora l'ho trovata, altrimenti vedo se è minore o maggiore di quella della radice e nel primo caso mi sposto nel sottoalbero di sinistra, nel secondo caso in quello di destra.

Ad esempio, l'albero di sotto è un binary search tree.

Sequenza: 37, 24, 42, 7, 2, 40, 45, 32, 120

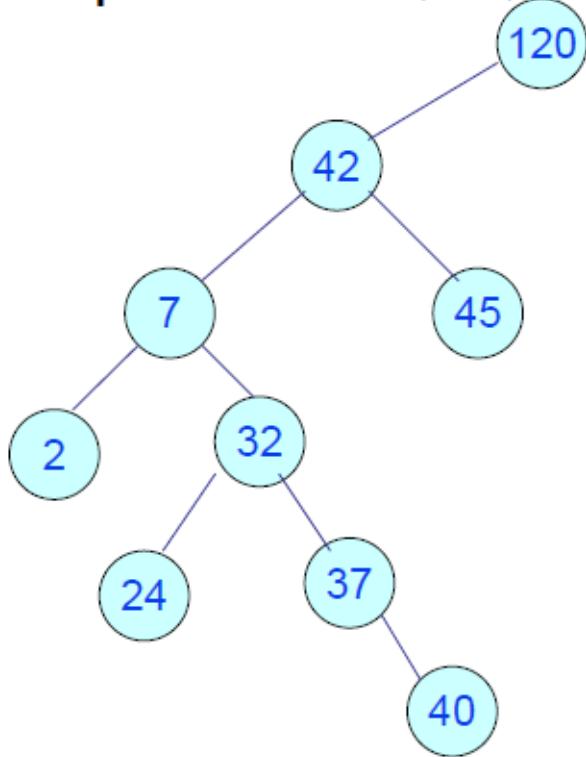


Infatti tutti i nodi a sinistra di 37 sono < 37 , tutti quelli a destra sono > 37 . Ma questo vale anche per le altre radici: tutti i nodi a sinistra di 24 sono minori di 24, tutti quelli a destra sono maggiori di 24, e stesso vale per tutti i nodi. La sequenza scritta poi ci fa capire come costruiremo l'albero. Infatti, supponiamo di non avere alcun albero. Allora dobbiamo memorizzare 37 e lo mettiamo nella radice. Poi facciamo un push di 24, che metteremo a sinistra e poi un push di 42, a destra. Poi inseriamo il 7, che va a sinistra e poi il 2 che va a sinistra del 7. Dopo capiremo bene, però il concetto è che quella sequenza non è a caso ma è la sequenza dei push effettuati per costruire l'albero.

Sostanzialmente il modo in cui costruiamo l'albero è molto semplice: ogni elemento che inseriamo andiamo a vedere se è maggiore, uguale o minore della radice precedente. Quindi il primo nodo lo costruiamo e ci mettiamo 37, poi quando inseriamo 24, essendo $24 < 37$, lo mettiamo a sinistra. Poi inseriamo 42, ed essendo $42 > 37$ lo mettiamo sempre al secondo livello ma a destra. Il 7 invece va a sinistra di 37, ma anche a sinistra di 24. E così via. Quindi basta navigare attraverso l'albero per inserire l'oggetto nella giusta posizione. E di fatto in pratica stiamo facendo l'esercizio che già tempo fa accennammo, ovvero trovare la posizione in cui inserire un nuovo elemento; e questo ci viene quasi gratis perché basta navigare nell'albero.

Anche quello di sotto è un albero di ricerca, anche se siamo stati sfortunati perché abbiamo potuto inserire elementi solo alla sinistra della radice dell'albero.

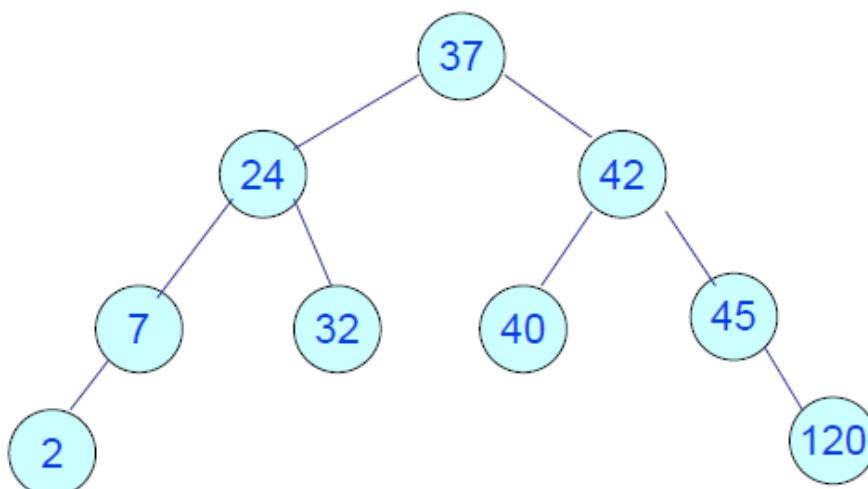
- **Sequenza:** 120, 42, 45, 7, 2, 32, 37, 24, 40



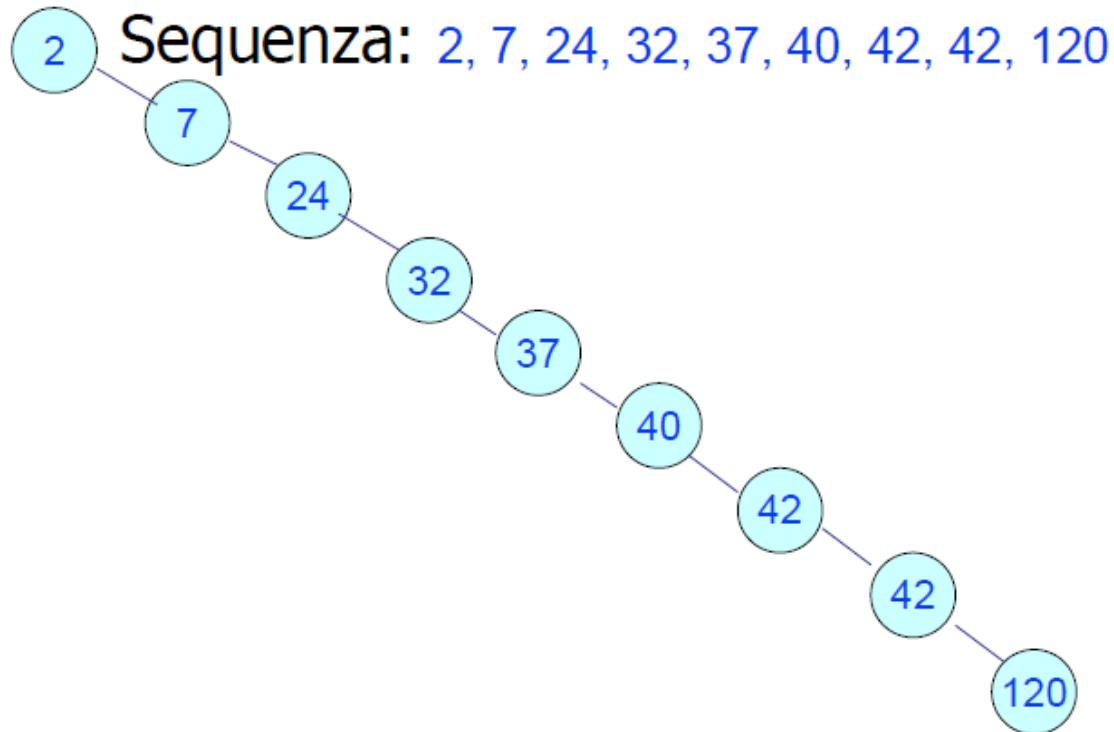
Ci rendiamo conto allora che dobbiamo essere furbi a non costruire l'albero in modo da avere questa situazione.

Un albero che invece ci piace è uno bilanciato, come quello di sotto:

- **Sequenza:** 37, 24, 42, 7, 2, 40, 45, 32, 120



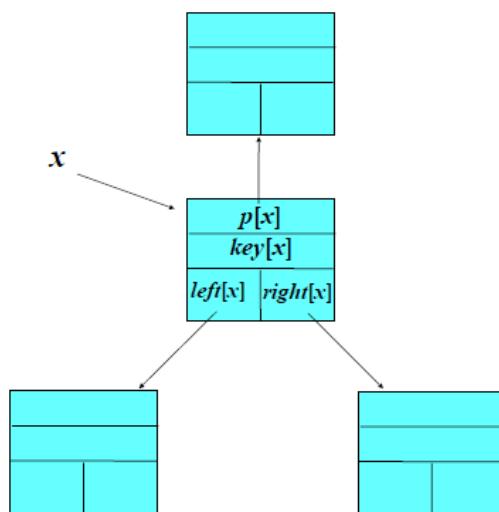
Anche questo è un albero binario di ricerca:



Però ha tutta la parte a sinistra vuota; di fatto è una lista concatenata. Notiamo che questa situazione ce l'abbiamo per una sequenza ordinata. Allora capiamo che quello che dobbiamo evitare è di avere una sequenza di elementi ordinati di cui fare il push; questo ci fa avere un albero completamente sbilanciato.

Come implementiamo l'albero binario di ricerca? Con una lista, dove ogni nodo invece che avere solo un puntatore al nodo successivo ha ben 3 puntatori: questo perché ogni nodo è collegati ad altri 3 nodi: il padre ed i 2 figli. Ovviamente la radice dell'albero non ha un padre, le foglie non hanno figli. In più ogni nodo dovrà contenere la chiave.

- Ciakun n o d o
dell'albero contiene
- *key[x]* – parte chiave
 - *left[x]* – riferimento al figlio sinistro
 - *right[x]* – riferimento al figlio destro
 - *p[x]* – riferimento al nodo padre



Per la *key* deve essere stato implementato l'operatore *<*, per il confronto.

La prima cosa che ci interessa è come sviluppiamo un *iterator* per poter navigare la struttura, ovvero per poter visitare l'albero. Quindi: come numeriamo gli elementi?

L'attraversamento di un albero produce un ordinamento lineare dell'informazione nell'albero. Su ciascun nodo si possono fare 3 azioni:

- **Muoversi a sinistra**
- **Muoversi a destra**
- **Visitare il nodo (es. stamparne il contenuto)**

Per convenzione quando visitiamo un albero (ovvero quando si stampa il contenuto dei suoi nodi) si va sempre prima a sinistra e poi a destra. Stabilita questa convenzione si hanno 3 opzioni:

- **Preorder** : si stampa e poi si va a sinistra prima e a destra poi;
- **Inorder** : si va prima a sinistra, si stampa e poi si va a destra;
- **Postorder**: si va a sinistra, poi a destra ed infine si stampa.

Queste 3 funzioni membro saranno funzioni membro ricorsive, perché le utilizziamo per tutti i nodi dell'albero. (NIL=NULL POINTER)

PREORDER-TREE-WALK(x)

1. **if** $x \neq \text{NIL}$
2. **then** stampa $\text{key}[x]$
3. PREORDER-TREE-WALK($\text{left}[x]$)
4. PREORDER-TREE-WALK($\text{right}[x]$)

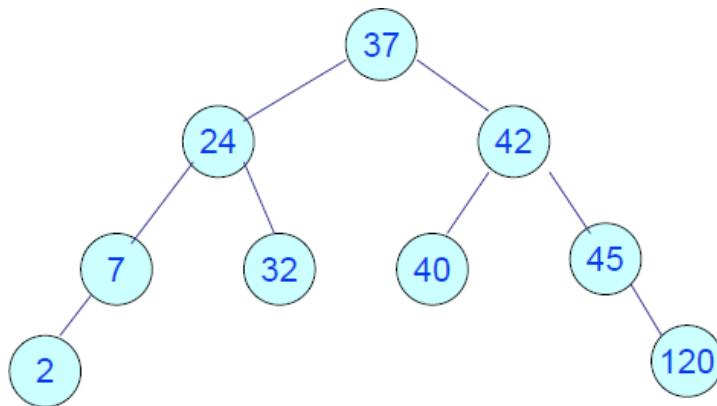
INORDER-TREE-WALK(x)

1. **if** $x \neq \text{NIL}$
2. **then** INORDER-TREE-WALK($\text{left}[x]$)
3. stampare $\text{key}[x]$
4. INORDER-TREE-WALK($\text{right}[x]$)

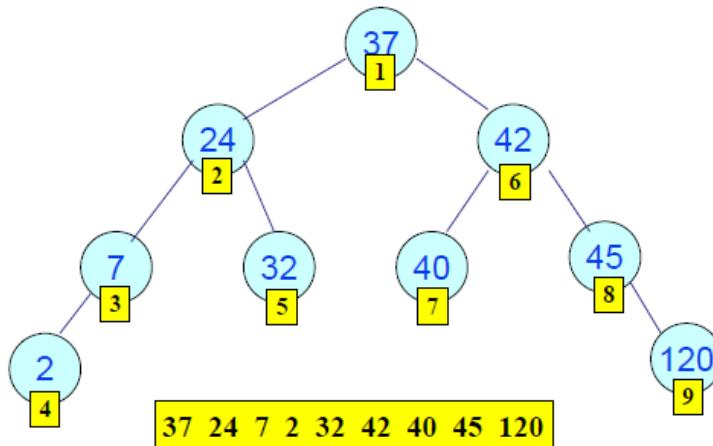
POSTORDER-TREE-WALK(x)

1. **if** $x \neq \text{NIL}$
2. **then** POSTORDER-TREE-WALK($\text{left}[x]$)
3. POSTORDER-TREE-WALK($\text{right}[x]$)
4. stampa $\text{key}[x]$

Vediamo subito un esempio:



Esempio: Visita preorder



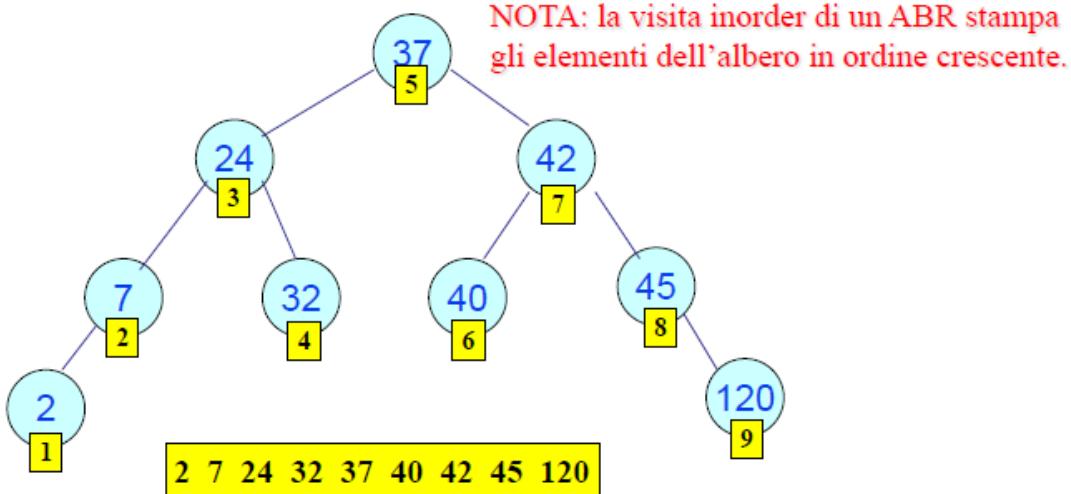
37 24 7 2 32 42 40 45 120

PREORDER-TREE-WALK(x)

- **if** $x \neq \text{NIL}$
- **then** stampa $\text{key}[x]$
- PREORDER-TREE-WALK($\text{left}[x]$)
- PREORDER-TREE-WALK($\text{right}[x]$)



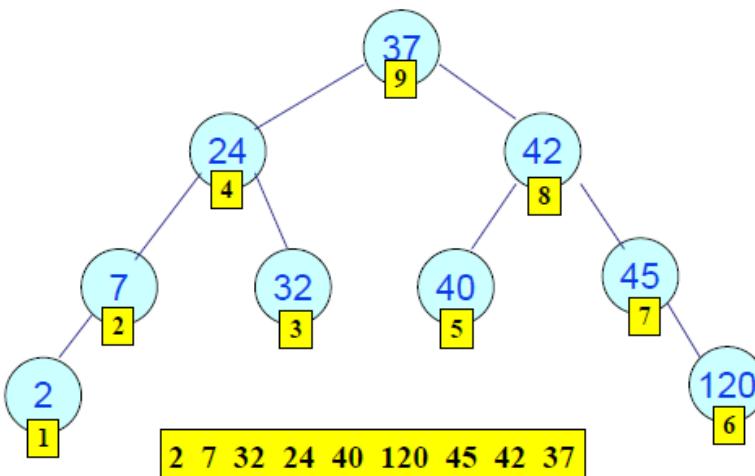
Esempio: Visita inorder



INORDER-TREE-WALK(x)

- if $x \neq \text{NIL}$
- then INORDER-TREE-WALK($\text{left}[x]$)
- stampa $\text{key}[x]$
- INORDER-TREE-WALK($\text{right}[x]$)

Con l'algoritmo *Inorder* vengono stampati gli elementi in ordine di chiave: a sinistra tutti quelli con chiave minore della chiave della radice dell'albero, a destra tutti quelli con chiave maggiore. Quindi la visita in Inorder permette di stampare gli elementi in ordine di chiave crescente.



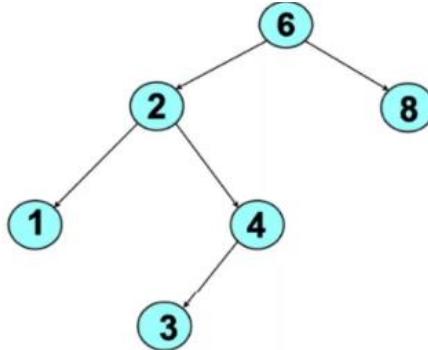
Tutti e tre gli algoritmi di visita hanno complessità $\Theta(n)$

POSTORDER-TREE-WALK(x)

- if $x \neq \text{NIL}$
- then POSTORDER-TREE-WALK($\text{left}[x]$)
- POSTORDER-TREE-WALK($\text{right}[x]$)
- stampa $\text{key}[x]$

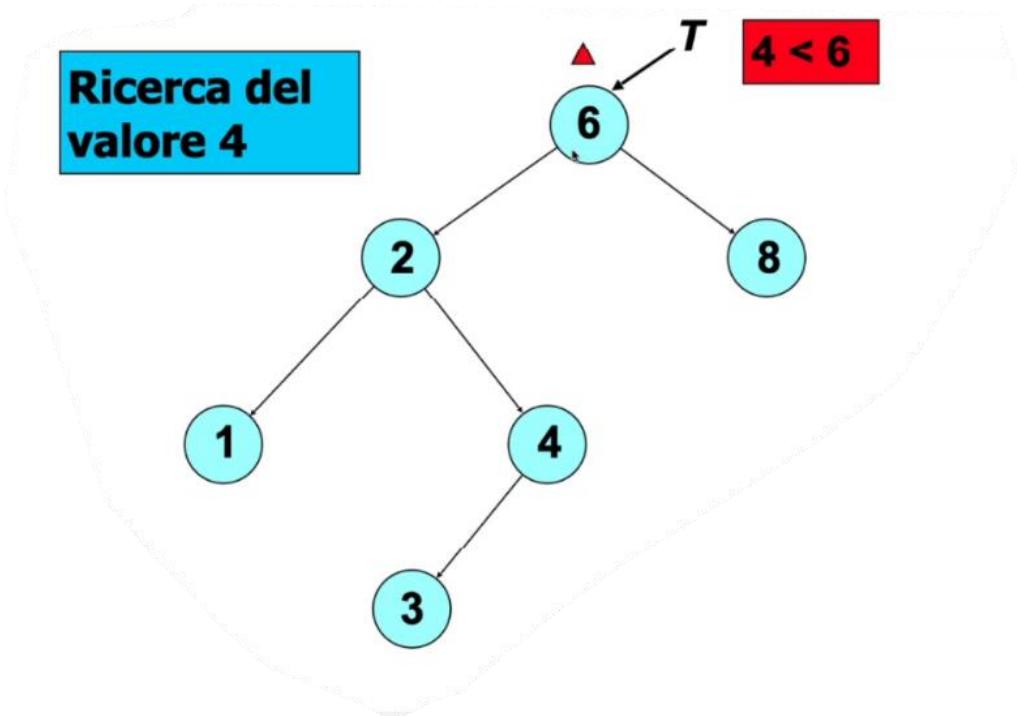
Ci si pone una domanda: è possibile ricostruire la struttura dell'albero a partire dalle liste di visita dell'albero? La risposta è sì, ma con almeno 2 liste (tra quelle preorder-inorder-postorder).

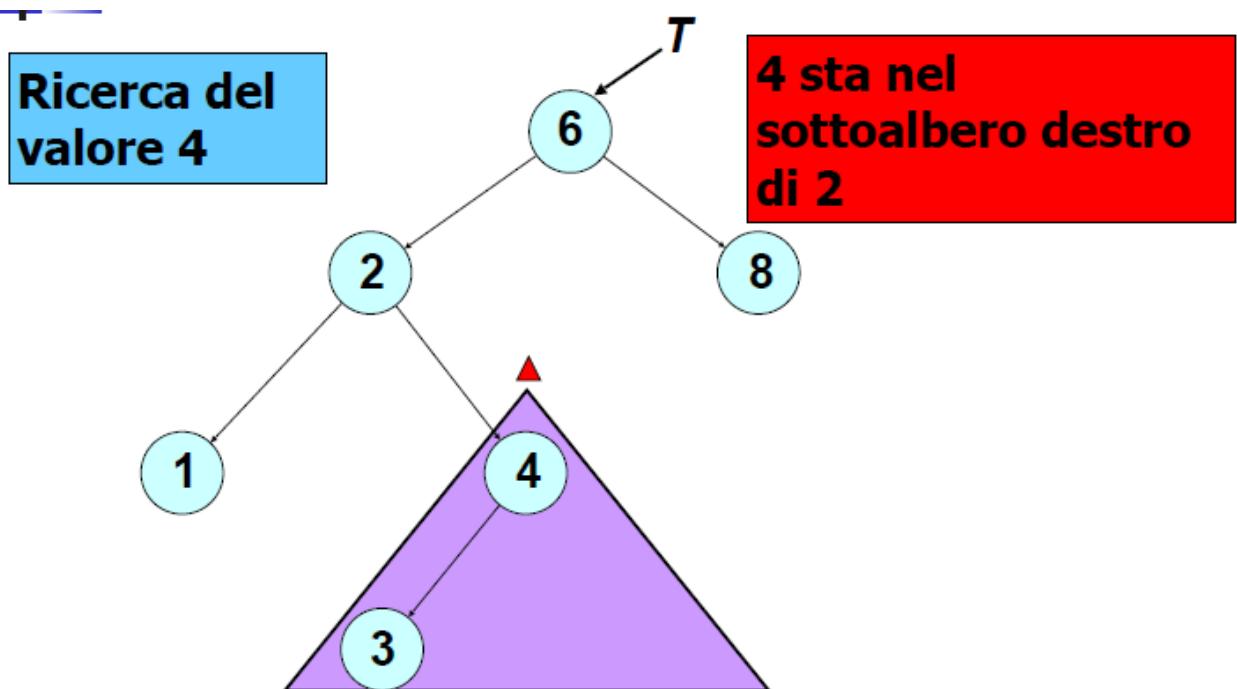
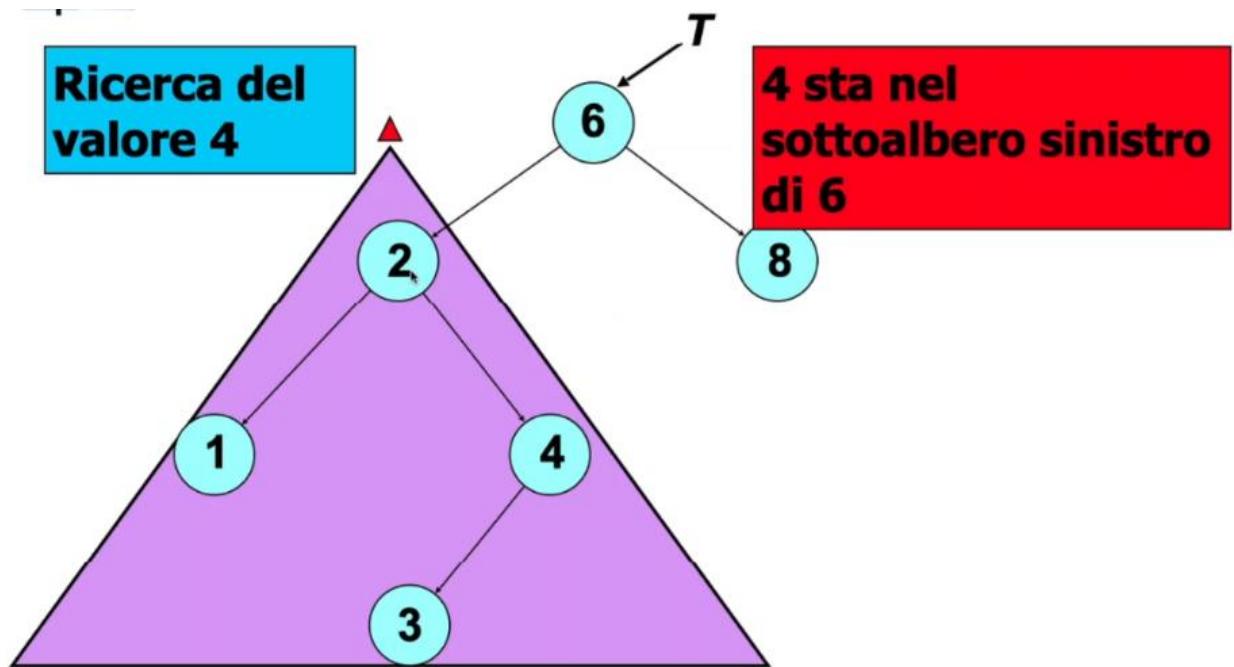
Vediamo come effettuare la ricerca in un ABR.

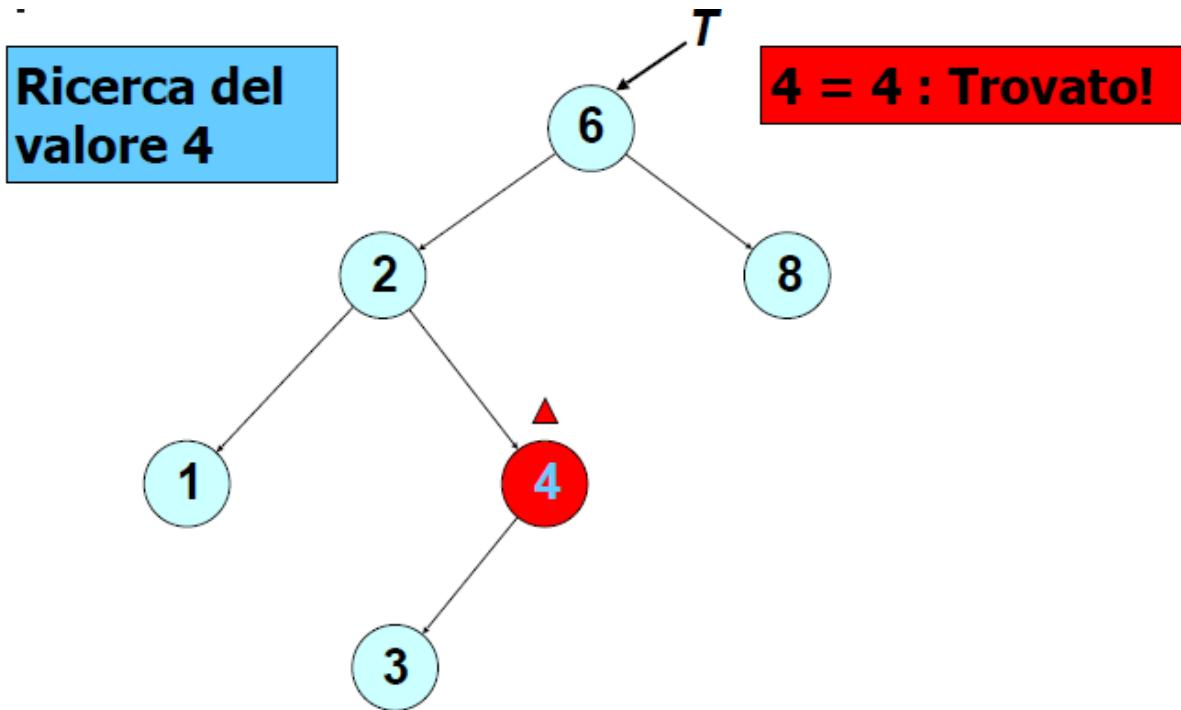


- Per trovare un elemento con chiave k in un albero T
 - confrontiamo k con $\text{key}[\text{root}[T]]$
 - Se $k < \text{key}[\text{root}[T]]$, ricerca k in $\text{left}[\text{root}[T]]$
 - altrimenti, ricerca k in $\text{right}[\text{root}[T]]$

ES: cerchiamo la chiave 4.







La ricerca può essere implementata sia in una versione iterativa che in una ricorsiva.

- **Versione Ricorsiva**

TREE-SEARCH(x, k)

1. **if** $x = \text{NIL}$
2. **then return** NIL
3. **if** $k = \text{key}[x]$
4. **then return** x
5. **if** $k < \text{key}[x]$
6. **then return** TREE-SEARCH($\text{left}[x], k$)
7. **else return** TREE-SEARCH($\text{right}[x], k$)

- **Versione Iterativa**

ITERATIVE-TREE-SEARCH(x, k)

1. **while** $x \neq \text{NIL}$
2. **do if** $k = \text{key}[x]$
3. **then return** x
4. **if** $k < \text{key}[x]$
5. **then** $x \leftarrow \text{left}[x]$
6. **else** $x \leftarrow \text{right}[x]$
7. **return** NIL

La prima chiamata avviene in entrambi i casi al nodo radice.

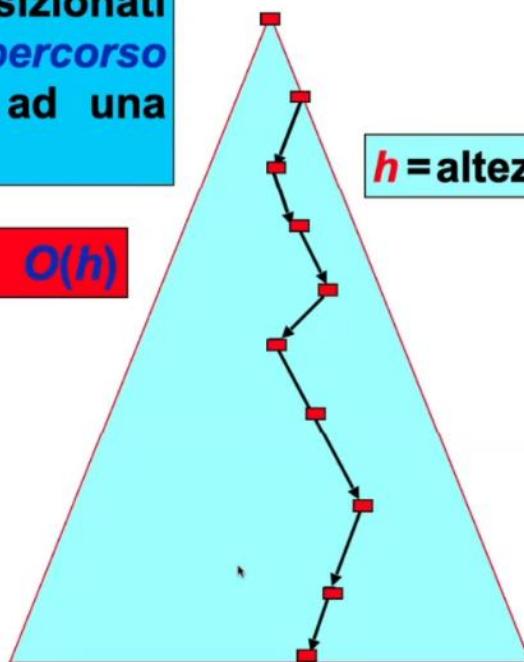
ANALISI DELL'ALGORITMO DI RICERCA

La ricerca ha un tempo che dipende dall'altezza dell'albero, poiché è confinata ai nodi posizionati lungo un singolo percorso dell'albero, dalla radice ad una foglia.

In generale, la *ricerca* è confinata ai *nodi* posizionati lungo un singolo percorso (path) dalla radice ad una foglia

Tempo di ricerca = $O(h)$

h = altezza dell'albero



32

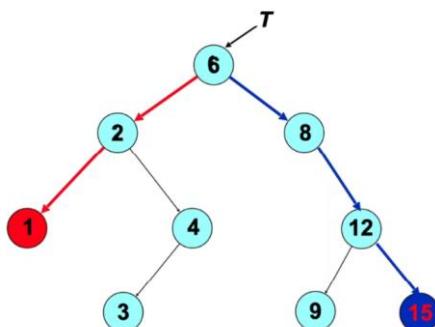
E quindi:

- se l'albero è bilanciato \rightarrow complessità logaritmica $O(\log(n))$.
- nel caso peggiore \rightarrow complessità $O(h)$

Allora ci rendiamo conto che in generale vogliamo alberi corti.

MINIMO E MASSIMO

facile trovare minimo e massimo. Il minimo lo trovo spostandomi sempre verso sinistra, il massimo spostandomi sempre verso destra.



- Ricerca del minimo e del massimo dell'albero con radice x

TREE-MINIMUM(x)

- **while** $left[x] \neq \text{NIL}$
- **do** $x \leftarrow left[x]$
- **return** x

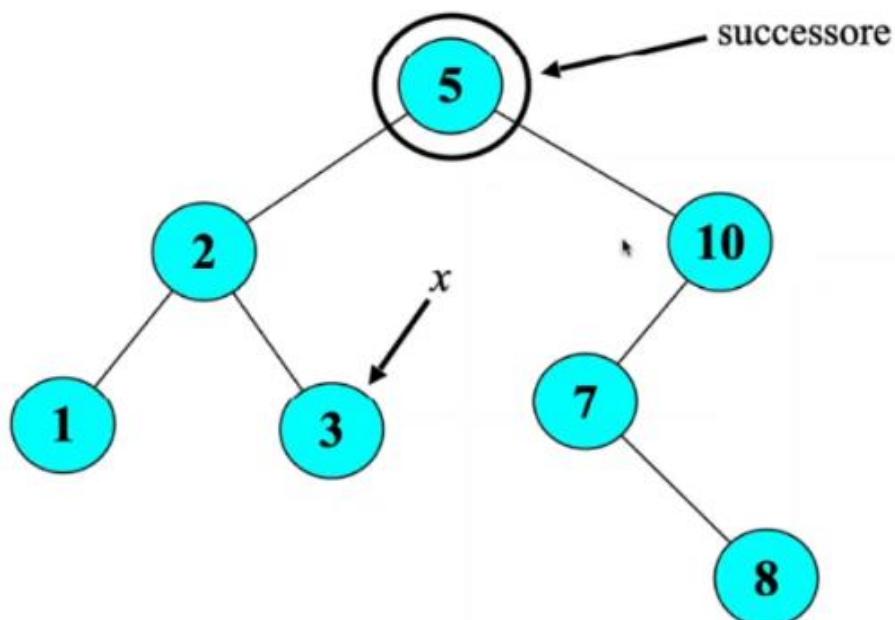
TREE-MAXIMUM(x)

- **while** $right[x] \neq \text{NIL}$
- **do** $x \leftarrow right[x]$
- **return** x

- Tempo di esecuzione $O(h)$, proporzionale all'altezza dell'albero

SUCCESSORE

Per trovare un successore di un nodo x bisogna andare a destra di x e trovare il minimo del sottoalbero di destra, perché quel valore sarà il successivo del nodo. Questo quando il nodo ha un figlio destro. Se però il nodo non ha un figlio destro ciò non vuol dire che non ha un successore, semplicemente il discorso è leggermente più complicato. Bisogna risalire l'albero; come? Dobbiamo pensare che se un nodo x non ha un figlio destro, il suo successore sarà sicuramente un nodo che ha x nel suo sottoalbero di sinistra, perché sarà più grande di x . Quindi bisogna tornare indietro fino a trovare il primo nodo che ha x nel suo sottoalbero di sinistra. Allora sono questi 2 i procedimenti per trovare il successore di un nodo x : uno per quando x ha un figlio destro ed uno quando x non ha alcun figlio destro. Sotto una situazione del 2° tipo:



In generale il procedimento per trovare il successore può essere riassunto dal seguente algoritmo:

```
TREE-SUCCESSOR( $x$ )
1. if  $right[x] \neq \text{NIL}$ 
2.   then return TREE-MINIMUM( $right[x]$ )
3.    $y \leftarrow p[x]$ 
4.   while  $y \neq \text{NIL}$  and  $x = right[y]$ 
5.     do  $x \leftarrow y$ 
6.      $y \leftarrow p[y]$ 
7.   return  $y$ 
```

Per un albero di altezza h , la ricerca del successore può essere eseguita in tempo $O(h)$

Nell'algoritmo mi sposto indietro con 2 puntatori in maniera che x sia il parent ed y sia il figlio.

PREDECESSORE

Il discorso è esattamente lo stesso, si procede in maniera duale.

Basta scambiare $left$ con $right$ e TREE-MAXIMUM con TREE-MINIMUM

INSERIMENTO E CANCELLAZIONE

Entrambe le operazioni vengono effettuate mantenendo le proprietà di ordinamento dell'ABR.

NOTIAMO CHE STIAMO FACENDO UN'ASSUNZIONE: LE CHIAVI SONO UNICHE. Se non fosse così il discorso sarebbe un pochettino più complicato.

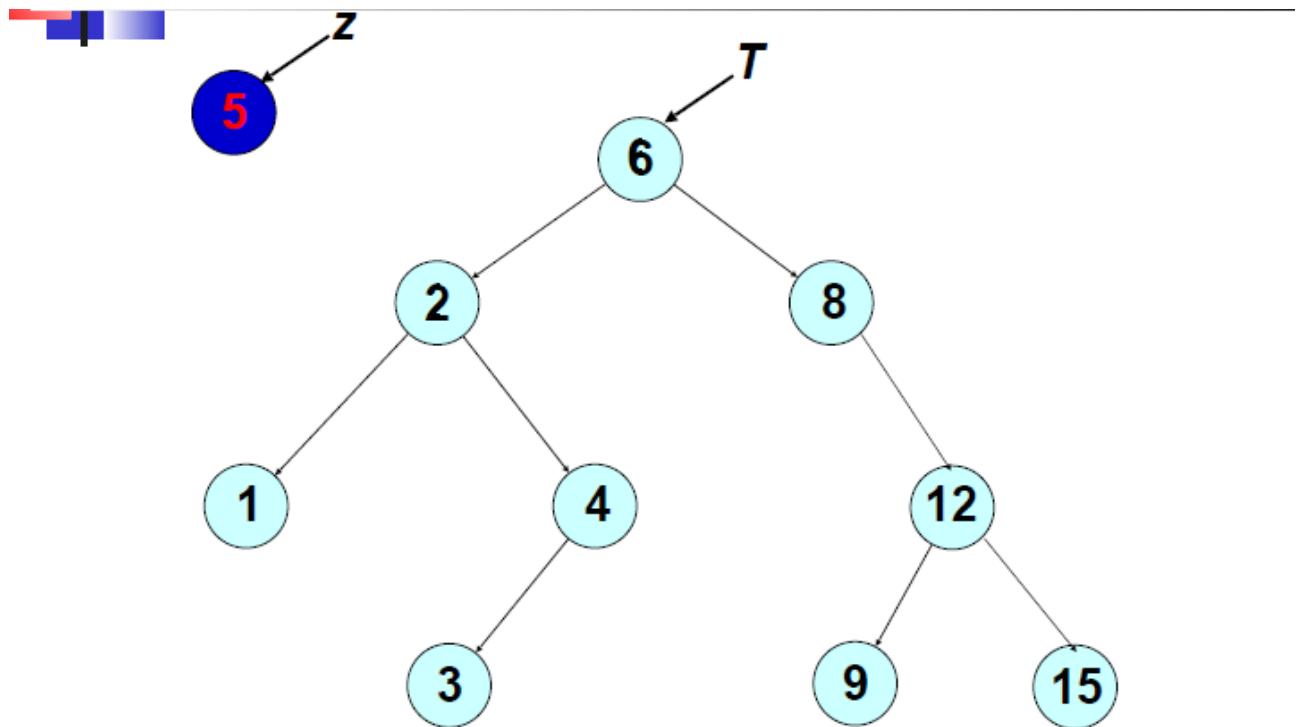
INSERIMENTO

L'inserimento è uguale alla ricerca, solo che:

- Si crea un nodo z con $left[z]=right[z]=NIL$
- Trova la posizione dell'albero che conterrebbe $key[z]$ (simile alla ricerca...),
- Si inserisce z in questa posizione

IL NUOVO NODO LO SI INSERISCE SEMPRE COME FOGLIA.

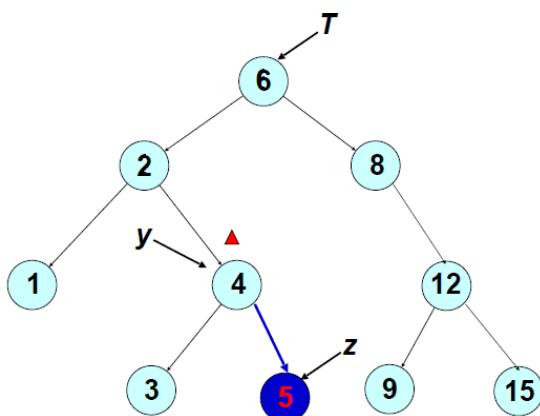
Es:



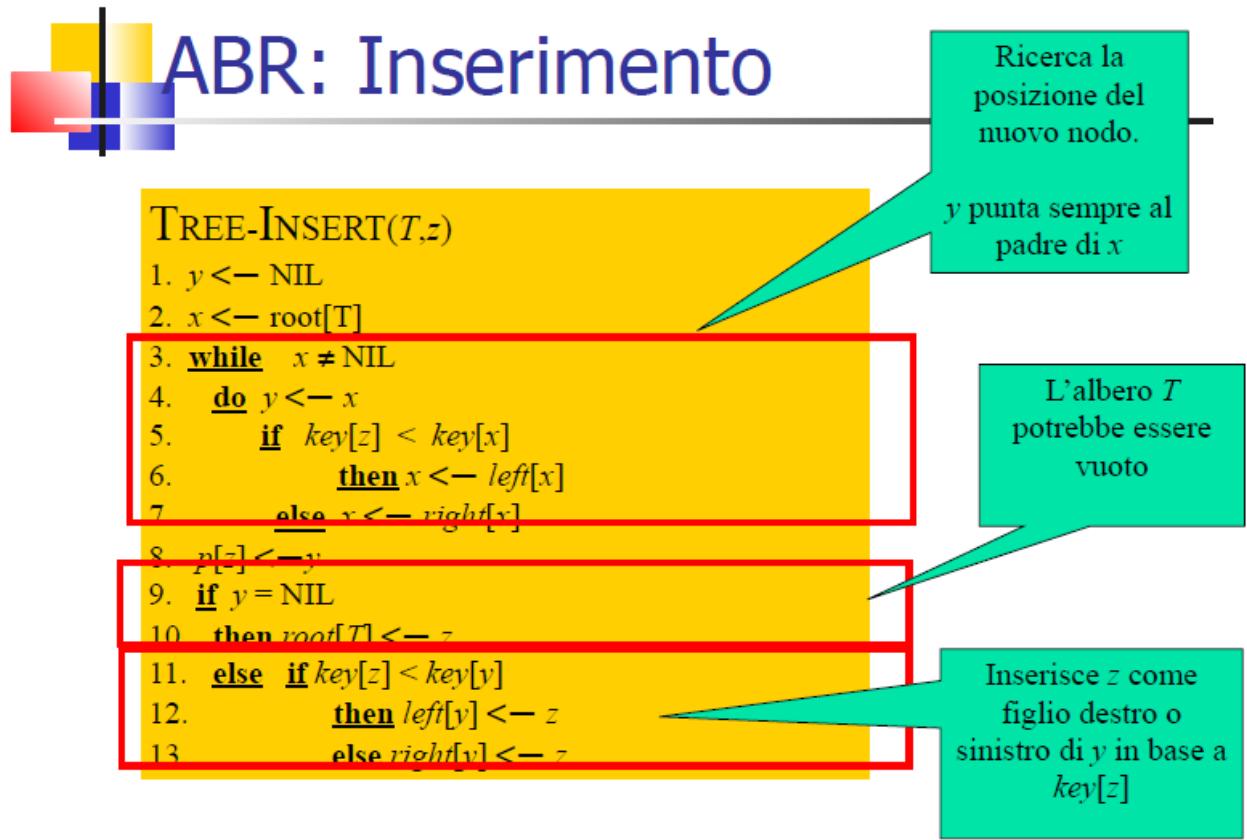
Dove lo si inserisce il 5?

Partiamo dalla radice. $5 < 6 \rightarrow$ scendo a sinistra; $5 > 2 \rightarrow$ scendo a destra; $5 > 4 \rightarrow$ scendo a destra.

Quindi il ragionamento è semplicemente questo.



Vediamo allora l'algoritmo in pseudo-codice:



Si usano sempre 2 puntatori: uno per il nodo attuale (x) ed uno per il padre (y).

Quando il while termina si stabilisce che y è il padre del nodo da inserire, e quindi si inserisce il nodo o come figlio sinistro di y o come figlio destro. Subito prima invece è verificato che l'albero non sia vuoto: se è vuoto allora y punterà a NIL e suo figlio sarà la radice nonché l'elemento da inserire.

Quanto costa l'inserimento?

Costa quanto l'altezza dell'albero perché bisogna sempre arrivare fino in fondo.

Il caso peggiore è sempre quello in cui l'albero è completamente sbilanciato → le chiavi sono inserite in ordine crescente o decrescente.

Come facciamo ad evitare che un avversario malizioso ci dia una lista di chiavi ordinate? Qui non possiamo randomizzare come fatto nel caso del quick sort e sarebbe un discorso lungo e complicato. In generale l'inserimento viene accompagnato da una procedura di bilanciamento. Questo lo si fa su una struttura dati più complessa che è quella degli alberi bilanciati.

La cancellazione non la vediamo.

STL MAP

La STL mette a disposizione una *map* basata sul BST. Il template ha 2 tipi: il tipo della chiave ed il tipo del value.

```
std::map  
template < class Key,  
          class T,  
          class Compare = less<Key>,  
          class Alloc = allocator<pair<const Key,T> >    // map::key_type  
          > class map;  
  
Map
```

Ovviamente assume che le chiavi possano essere confrontate con “<”.

Es:

```
1 #include <iostream>  
2 #include <map>  
3 using std::map;  
4 using std::string;  
5  
6  
7 int main() {  
8  
9     map<string,double> myConst;  
10    //use of [] operatore  
11    myConst[ "PI" ] = 3.14;  
12    myConst[ "e" ] = 2.718;  
13    myConst[ "Erdos" ] = 1.6066;  
14    //use of insert  
15    myConst.insert(std::pair<string,double>( "Gauss" , 0.83462));  
16    return 0;  
17 }
```

La mappa crea una corrispondenza tra le chiavi ed i valori in maniera da effettuare non solo il search in tempi logaritmici ma anche l'ordinamento.

La mappa ha inoltre un operatore [] che permette di indirizzare gli elementi della mappa mediante la chiave. Ad esempio nella riga 9 definiamo il container (non è myConst ma myCont, errore di battitura) come una mappa di coppie (stringa, double) e quindi la chiave è una stringa, il valore è un double. Quindi il secondo oggetto della mappa può essere qualunque, basta che il primo abbia un operatore “<” per il confronto. Alla riga 11 inseriamo nel container un oggetto con chiave PI (pi greco) ed il cui contenuto è 3.14; in pratica questa non è altro che una insert nel container e stiamo indirizzando un elemento del container utilizzando la chiave. Se non si vuole usare [], si può usare la funzione *pair* che permette di costruire una coppia (chiave, valore), invocando *insert*.

Esiste poi la ordered map, che non vedremo, basata ancora su un'altra struttura dati che è la hash table e che viene sempre messa a disposizione dalla STL.

Vediamo la nostra implementazione della mappa, attraverso il BST.

MAP CON BST

```
1 //bst.h
2 #ifndef BST_H
3 #define BST_H
4 #include <iostream>
5 using std::cout;
6 template <typename Key> class BST;
7
8
9 template<typename Key>
10 class BSTNode {
11 public:
12     BSTNode(Key k, BSTNode<Key> *p = nullptr, BSTNode<Key> *l = nullptr, BSTNode<Key> *r = nullptr)
13         : key(k), left(l), right(r), parent(p) {}
14     friend class BST<Key>;
15 private:
16     Key key;
17     BSTNode<Key> *left;
18     BSTNode<Key> *right;
19     BSTNode<Key> *parent;
20 };
```

Il generico nodo avrà quindi 3 puntatori ad altri nodi ed un valore che è la chiave. Dichiariamo poi BST come classe amica perché deve poter accedere ai membri di BSTNode. Andrebbero implementati anche set e get di BSTNode.

```
24 class BST {
25 protected:
26     BSTNode<Key> *root;
27 public:
28     // Constructors
29     BST(BSTNode<Key>* r = NULL): root(r) {}
30     ~BST() {release(root);}
31
32     // traverse
33     void inorderTreeWalk(BSTNode<Key>* x);
34     void inorderTreeWalk() {inorderTreeWalk(root);}
35
36     // Accessors
37     BSTNode<Key>* getRoot() {return root;}
38     BSTNode<Key>* search(Key key);
39     BSTNode<Key>* search(BSTNode<Key> **x, Key key);
40     BSTNode<Key>* minimum() {return minimum(root);}
41     BSTNode<Key>* minimum(BSTNode<Key> *subroot);
42     BSTNode<Key>* maximum() {return maximum(root);}
43     BSTNode<Key>* maximum(BSTNode<Key> *subroot);
44     BSTNode<Key>* predecessor(BSTNode<Key>* x);
45     BSTNode<Key>* successor(BSTNode<Key>* x);
46     // Modifiers
47     BSTNode<Key>* insert(Key key);    // if return NULL, the key already exists.
48     void release(BSTNode<Key>* x);
49     //BSTNode<Key>* deleteNode(Key key); // if return NULL, the key does not exists.
50 };
```

Le funzioni di BST sono predecessor, successor, minimum e maximum; poi ci sta la ricerca, implementata in 2 modi: nel primo si inserisce la chiave dell'elemento che si

ricerca, nel secondo si inserisce anche il nodo da cui partire per la ricerca e quindi la radice (*root*) in sostanza. Questo secondo modo potremmo anche non esporlo e dichiararlo come privato. C'è poi anche la funzione *inorderTreeWalk* per esplorare l'albero in questa modalità.

```
52 template<typename Key>
53 void BST<Key>::inorderTreeWalk(BSTNode<Key>*>x) {
54     if (x) {
55         inorderTreeWalk(x->left);
56         //visit the node: here you
57         //can substitute your visitor function
58         cout << x->key << "\n";
59         inorderTreeWalk(x->right);
60     }
61 }
62
63
64 template<typename Key>
65 BSTNode<Key>* BST<Key>::search(BSTNode<Key> *x, Key key) {
66     while (x) {
67         if (x->key == key) break;
68         else if (x->key < key) x = x->right;
69         else x = x->left;
70     }
71     return x;
72 }
73
74 template<typename Key>
75 BSTNode<Key>* BST<Key>::search(Key key) {
76     return search(root, key);
77 }

78
79
80 template<typename Key>
81 BSTNode<Key>* BST<Key>::minimum(BSTNode<Key> *subroot) {
82     BSTNode<Key>* x = subroot;
83     if (x) {
84         while (x->left) x = x->left;
85     }
86     return x;
87 }
88
89 template<typename Key>
90 BSTNode<Key>* BST<Key>::maximum(BSTNode<Key> *subroot) {
91     BSTNode<Key>* x = subroot;
92     if (x) {
93         while (x->right) x = x->right;
94     }
95     return x;
96 }
97 }
```

```

98 template<typename Key>
99 BSTNode<Key>* BST<Key>::predecessor(BSTNode<Key>* x) {
100     BSTNode<Key>* prev = NULL;
101     if (x->left) prev = maximum(x->left);
102     else {
103         prev = x->parent;
104         while (prev && prev->left == x) {
105             x = prev;
106             prev = x->parent;
107         }
108     }
109     return prev;
110 }
111
112 template<typename Key>
113 BSTNode<Key>* BST<Key>::successor(BSTNode<Key>* x) {
114     BSTNode<Key>* succ = nullptr;
115     if (x->right) succ = minimum(x->right);
116     else {
117         succ = x->parent;
118         while (succ && succ->right == x) {
119             x = succ;
120             succ = x->parent;
121         }
122     }
123     return succ;
124 }

127 template<typename Key>
128 BSTNode<Key>* BST<Key>::insert(Key key) {
129     if(search(key)) return nullptr;
130     BSTNode<Key>* x = root;
131     BSTNode<Key>* y = nullptr;
132     BSTNode<Key>* z = new BSTNode<Key>(key);
133     while(x) {
134         y = x;
135         if(z->key < x->key) x = x->left;
136         else x = x->right;
137     }
138     z->parent = y;
139     if(y==nullptr)
140         root = z;
141     else {
142         if (z->key < y->key)
143             y->left = z;
144         else
145             y->right = z;
146     }
147     return z;
148 }
149

```

Test:

```
1 #include "bst.h"
2
3 #include <iostream>
4
5
6 int main(){
7     BST<int> T;
8     T.insert(10);
9     T.insert(12);
10    T.insert(4);
11    T.inorderTreeWalk();
12 }
```