By Giuseppe Casizzone

Domanda: Cos'è una **ECCEZIONE**?
Risposta: Una **ECCEZIONE** rappresenta un'alternativa alle tecniche tradizionali quando queste sono insufficienti a gestire delle situazioni eccezionali. L'idea è che una funzione che trova un errore che non sa gestire lancia un'eccezione, nella speranza che possa gestire il problema. Il codice della gestione di un'eccezione è separato da quello ordinario. Se ripercorrendo a ritroso la catena di chiamanti non si incontra nessuna funzione che cattura l'eccezione, il programma viene terminato. Il costrutto **catch()** può essere usato solo dopo un blocco preceduto dalla keyword try o dopo un altro blocco **catch(). Catch()** ha tra parentesi una dichiarazione simile a quella degli argomenti di una funzione. Se una funzione nel blocco **try** lancia un'eccezione:
- le istruzioni nel blocco **try** seguenti tale funzione **non vengono eseguite**
- viene eseguito solo il **primo** handler trovato per il tipo di eccezione lanciata
try { codice che lancia un'eccezione di tipo E; } catch(H) { quando arriviamo qui? } :
1. H è dello **stesso tipo** di E (o una sua classe base)
2. H ed E sono di tipo **puntatore** ed 1 vale per i **tipi puntati**
3. H è un **riferimento** ed 1 vale per il **tipo a cui H si riferisce**

Catch() può essere usato per catturare qualunque tipo di eccezione e, se posto in testa ad una lista di handler, fa sì che gli altri handler **non** vengano mai considerati. Se un handler non è in grado di gestire completamente un'eccezione, può **rilanciarla** assumendo che un altro hanlder possa completarne la gestione. Il meccanismo delle eccezioni consente di specificare il tipo di una eccezione (tramite catch()) e di fornire informazioni sull'errore che ha causato l'eccezione. Se una eccezione non viene catturata all'interno di uno specifico ambiente di visibilità, viene fatto un tentativo di catturare l'eccezione in un blocco esterno. Scaricare lo STACK (stack unwinding) significa che la funzione nella quale l'eccezione non viene catturata viene terminata ed il suo record di attivazione deallocato. Il controllo ritorna al punto in cui la funzione è stata invocata e, se l'eccezione non viene catturata in un altro blocco esterno o nel resto del programma, quest'ultimo viene terminato.

Domanda: Cos'è il **POLIMORFISMO**?
Risposta: Il **POLIMORFISMO** è la proprietà di una entità di assumere forme diverse nel tempo. Esso consente agli oggetti di classi diverse all'interno di una gerarchia di esibire un comportamento diverso a tempo di esecuzione in risposta ad uno stesso messaggio (= invocazione di una funzione) che assume forma diversa a seconda del tipo di oggetto. Il **POLIMORFISMO** supporta dunque la proprietà di **estensibilità** di un sistema, nel senso che minimizza la quantità di codice che occorre modificare quando si introducono nuove classi e nuove funzionalità. Il **POLIMORFISMO** viene realizzato tramite il **BINDING DINAMICO** (**late binding**): esso consiste nel determinare a **tempo di esecuzione** (anziché a **tempo di compilazione**) il corpo del metodo da invocare su un dato oggetto. Il **POLIMORFISMO** è implementato sfruttando la **compatibilità** tra puntatore a **classe base** e puntatore a **classe derivata** mediante l'uso di funzioni membro **VIRTUAL**. Una funzione virtual viene **definita** nella classe base e **ridefinita** nelle classi derivate. Quando una funzione virtual è invocata attraverso un puntatore alla classe base viene eseguita la versione ridefinita corretta con **riferimento alla classe dell'oggetto puntato**. Se una funzione è dichiarata virtual nella specifica della classe base **anche le sue versioni ridefinite nelle classi derivate** sono automaticamente virtual. Il polimorfismo può essere attivato sia mediante l'uso di un **puntatore** a classe base sia con l'uso di un **riferimento** a classe base.

Domanda: Cos'è **l'EREDITARIETA'**? (vuole sapere anche i metodi di accesso public, protected e private, in pratica lo schema che c'è sulle slides).
Risposta: **l'EREDITARIETA'** è quel meccanismo che comporta una strutturazione gerarchica nel sistema software da costruire e che promuove il riuso del software. L'**EREDITARIETA'** consente di realizzare relazioni tra classi di tipo generalizzazione – specializzazione. Una classe **base** (classe madre) realizza un comportamento generale comune ad un insieme di entità; mentre le classi **derivate** (classi figlie) realizzano comportamenti specializzati rispetto a quelli della classe base. Si creano quindi delle relazioni tra classi e, proprio per questo, un oggetto di una classe derivata può essere trattato come un oggetto della classe base. E' possibile che alcuni dei comportamenti definiti dalla classe base valgano per la classe derivata ma debbano essere **ridefiniti**, possibilmente **riutilizzando** il codice esistente. La classe derivata **NON** ha accesso alla parte **privata** della classe base. Al fine di consentire l'accesso alle classi derivate è introdotta la sezione **protected** nella specifica della classe. Variabili e funzioni membro **protected** restano **NON** accessibili ad un utente esterno. Ereditare dati **protected** può avere effetti positivi sull'efficienza del programma dato che permette alle derivate di accedere a tali dati senza necessariamente invocare le funzioni membro della classe base. D'altro canto vi sono almeno due svantaggi:
- le classi derivate possono effettuare **operazioni scorrette**
- se l'implementazione della classe base cambia, **la modifica può ripercuotersi** anche sulle derivate
In una gerarchia di derivazione **i costruttori sono invocati sempre nell'ordine di derivazione** (dal costruttore della classe meno derivata al costruttore della classe più derivata). Viceversa **i distruttori sono invocati nell'ordine inverso**. Inoltre le seguenti funzioni membro **non** vengono ereditate e devono essere **ridefinite** nelle classi derivate: **costruttori (compreso quello di copia), distruttori, operatore di assegnazione, funzioni membro private.**

Domanda: Cos'è un **COSTRUTTORE DI COPIA**?
Risposta: Un **COSTRUTTORE DI COPIA** è un costruttore che crea un oggetto a partire da un altro oggetto della classe (i valori delle variabili membro vengono copiati). Esso è **implicitamente** richiamato:
- all'atto della definizione di un oggetto per inizializzarlo con il valore di un altro oggetto.
- all'atto della chiamata di una funzione per inizializzare un oggetto passato per valore.
- all'atto del ritorno da una funzione, per restituire l'oggetto proprio per valore.
In mancanza della definizione del costruttore di copia, il compilatore richiama un **costruttore di copia di default**. Esso esegue una copia superficiale dell'oggetto limitata alla sola parte base. Nel caso di una estensione dinamica, il costruttore di copia deve essere esplicitamente definito dal progettista della classe. Il **costruttore di copia** non può essere ereditato e possono verificarsi i seguenti casi (relativamente alla costruzione di un oggetto derivato):
- **né la classe base né la derivata implementano CC:** il compilatore utilizza il CC di default ricorsivamente effettuando l'assegnazione membro a membro
- **solo la classe base implementa CC:** il compilatore utilizza il CC della classe base per la parte base e l'operatore di default per la parte propria
- **sia la classe base che la derivata implementano CC:** il CC della classe derivata deve richiamare mediante lista di inizializzazione il CC della classe base
- **solo la classe derivata implementa CC:** il compilatore non è in grado di richiamare il CC di default della classe base! La classe derivata diviene responsabile della corretta inizializzazione del sotto oggetto-base.

Domanda: Cos'è e come funziona l'**OPERATORE DI ASSEGNAZIONE** ? (inclusa implementazione)
Risposta: L'**ASSEGNAZIONE** è un'operazione **distruttiva**. L'oggetto deve in generale essere distrutto e poi ricostruito a partire dal nuovo valore. L'operatore di assegnazione **NON** può essere ereditato e deve essere

una funzione **membro**. Ha la forma "**const C& operator=(const C&)"** perché deve consentire assegnazioni multiple del tipo a=b=c (quindi l'oggetto deve essere ritornato dalla funzione) e deve impedire situazioni del tipo (a=b)=c (quindi l'oggetto viene ritornato const). L'**OPERATORE DI ASSEGNAZIONE** deve dunque ritornare l'oggetto su cui è applicato e quindi è necessario utilizzare il puntatore **THIS**. Il puntatore deve ovviamente essere **dereferenziato**, in quanto il tipo di ritorno è un riferimento all'oggetto (e non il suo indirizzo): **return *this**. L'**operatore di assegnazione non viene ereditato** e possono verificarsi i seguenti casi (relativamente all'assegnazione tra due oggetti derivati):
- **né la classe base né la derivata implementano " = ":** il compilatore utilizza l'op. " = " di default ricorsivamente effettuando l'assegnazione membro a membro
- **solo la classe base implementa " = ":** il compilatore utilizza l'operatore della classe base per la parte base e l'operatore di default per la parte propria
- **sia la classe base che la derivata implementano " = ":** l'operatore " = " della classe derivata deve esplicitamente richiamare l'operatore di assegnazione della classe base
- **solo la classe derivata implementa " = ":** il compilatore non è in grado di richiamare l'operatore " = " di default della classe base! La classe derivata diviene responsabile della corretta assegnazione del sotto oggetto-base.

Domanda: Cosa sono le **FUNZIONI INLINE** ?
Risposta: Le **FUNZIONI INLINE** sono funzioni le cui chiamate, per motivi di efficienza, **non** sono tradotte in linguaggio macchina dal compilatore tramite salto a sottoprogramma (con relativo scambio di parametri). Il codice della funzione viene inserito **in linea**, cioè nel punto di chiamata. L'uso di tali funzioni deve essere limitato a funzioni dal corpo molto breve, altrimenti i vantaggi sono irrilevanti. Le **FUNZIONI INLINE** devono essere definite nello stesso file in cui sono invocate, ovvero in un file di intestazione (*header file*) da importare, perché il compilatore deve conoscerne il corpo per sostituirlo ad ogni chiamata. Tali funzioni non sempre sono sviluppate in linea dal compilatore, non possono essere esportate e possono specificare argomenti di default.

Domanda: Cos'è la parola chiave **EXPLICIT** ?
Risposta: Il C++ fornisce la parola chiave **EXPLICIT** per inibire l'utilizzo dei costruttori con un solo argomento nella conversione implicita di tipo (per es. explicit Vettore(const int = 10);).

Domanda: In cosa consiste l'**OVERLOADING DEGLI OPERATORI**? (i più chiesti sono gli operatori di shift e l'operatore di assegnazione, inclusa l'implementazione)
Risposta: L'**overloading degli operatori** è una caratteristica importante che consente di attribuire ulteriori significati agli operatori del linguaggio. Un **operatore unario** può essere ridefinito:
- come **funzione membro senza argomenti**
- come **funzione NON membro che riceve un solo argomento**
Un **operatore binario** può essere ridefinito:
- come **funzione membro che riceve un solo argomento**
- come **funzione NON membro che riceve due argomenti**.

Domanda: Cos'è la **DERIVAZIONE MULTIPLA** e cosa si intende per **DIAMOND PROBLEM** ?
Risposta: La **DERIVAZIONE MULTIPLA** si divide in:
- Derivazione multipla da **classi tra loro NON collegate**: in questo caso, ad esempio, la classe D, figlia delle classi B e C tra loro **NON** collegate, eredita **TUTTE** le variabili e funzioni membro di B e C.
- Derivazione multipla da **classi tra loro collegate**: in questo caso, ad esempio, le classi B e C, che a loro volta sono figlie di A, contengono le variabili membro di A. La classe D, figlia di B e C, contiene le variabili

membro di B e C e quindi 2 volte le variabili membro di A ("**DIAMOND PROBLEM**"). Occorre utilizzare il meccanismo di risoluzione di visibilità per distinguere tra le due distinte copie delle variabili membro definite in A. Un'ambiguità si verifica se si tenta di eseguire un up-casting per assegnare "d" ad un oggetto "a" di A: quale dei due sotto-oggetti di A assegnare ad "a" ? La chiamata dei costruttori avviene **secondo l'ordine di derivazione**, a partire dalla classe base e tenendo conto dell'**ordine di elencazione** nella dichiarazione di derivazione. In alcuni casi però sarebbe meglio **non** avere una duplicazione dei sotto-oggetti della classe base A…

- Derivazione **VIRTUALE**: Per evitare la duplicazione dei dati si ricorre alla **DERIVAZIONE VIRTUALE**. Rispetto al caso precedente, la classe D contiene **SOLO UNA COPIA** delle variabili membro di A. Abbiamo quindi la classe A ereditata in maniera **VIRTUALE** (classe base virtuale), le classi B e C che **non** contengono le variabili membro di A ma **un puntatore ad A**, la classe D che contiene le variabili membro di B e C ed un doppio puntatore ad A (ma **una sola volta** un sotto-oggetto di tipo A). Non essendoci ambiguità sul sotto-oggetto A, l'up-casting è lecito. Dato che D contiene un solo sotto-oggetto di A, bisogna impedire che entrambe le classi ereditate da D (B e C) invochino il costruttore di A. Nel caso di derivazione virtuale, **la sola classe maggiormente derivata** (la D) può invocare il costruttore della classe base A. Le eventuali chiamate al costruttore di A nei costruttori B e C (quando invocati dal costruttore di D) vengono **ignorate**.

(NOTA: in assenza di derivazione virtuale, il costruttore di una classe non può invocare il costruttore di una classe da cui non discende direttamente.)

Domanda: Scrivi una funzione che effettui la **SCRITTURA SU FILE**.

Risposta:

```
void scriviSuFile (T& t)
{
ofstream outfile("prova.txt", ios:out);

if(!outfile)
{
  cout << "Impossibile aprire il file!" << endl;
  exit(1);
}

else
  file << t.getA() << " " << t.getB() << " " << t.getC() << " " << endl;

outfile.close();
}

void letturasufile(T & t){

ifstream infile("prova.txt");

if(!infile)cout<<"File non trovato";

else  infile <<t.getA()<<""<<t.getB()<<endl;
infile.close();

)
```

Domanda: scrivere una funzione che effettui una **RICERCA BINARIA RICORSIVA**.

Risposta:

```
int ricercaBinRic (const Vettore V, const int primo, const int ultimo, const T x, int & pos)
{
int med, trovato=0;
if((ultimo-primo>=0 && !trovato) {
med=(ultimo+primo)/2;
if(V[med]==x) {
```

```
pos=med;
trovato=1;
}
else if (V[med]<x) trovato=ricercaBinRic(V, med+1, ultimo, x, pos);
else trovato=ricercaBinRic(V, primo, med-1, x, pos);
}
return trovato;
}
```

Domanda: cosa sono e a cosa servono i **NAMESPACE** ?
Risposta: In progetti di dimensioni medio-grandi è possibile che ci siano **conflitti tra i nomi** (ad esempio l'omonimia) esportati tra due differenti librerie. Questo fenomeno è noto come "**inquinamento dello spazio dei nomi**". Il C tenta di risolvere il problema attraverso l'uso dei simboli. Il C++ offre un metodo che risolve il problema: il meccanismo dei **NAMESPACE**. I **NAMESPACES** permettono di raggruppare classi, oggetti, funzioni, variabili definendo in pratica un ambiente di visibilità per **identificatori globali** e **variabili globali**. A livello di modulo è possibile dichiarare un namespace nel quale vengono definiti tutti i simboli del modulo.
I nomi raggruppati dal namespace hanno **visibilità locale al namespace**, quindi sono incapsulati nel namespace e **non** sono visibili all'esterno, evitando conflitti tra nomi che si ripetono nel programma. Il nome del namespace è invece un nome **globale** e può essere usato **al di fuori** del namespace. Per poter utilizzare i nomi raggruppati da un namespace, bisogna **esplicitamente aprire** lo spazio dei nomi, specificando il namespace di cui fanno parte. Tale apertura può essere fatta in due modi:
- utilizzando l'**operatore binario ::** (operatore di risoluzione di visibilità) specificando come operandi il **namespace** e il **nome** cui si vuole accedere: **namespace::nome**
- utilizzando l'istruzione: **using namespace nome_namspace;** (nel caso i nomi del namespace siano usati di frequente nel programma).
Un namespace può contenere sia dichiarazioni che definizioni, ma si usa tenerle **separate**. In particolare le funzioni vengono dichiarate nel namespace e definite fuori di esso (ecco perché si usa l'operatore di risoluzione di visibilità, per qualificare i membri definiti altrove). Possono comunque verificarsi conflitti se:
1. Si usano librerie o namespace che raggruppano **nomi uguali**
2. Si usano **contemporaneamente** due namespace con lo **stesso nome** e in cui vi sono **nomi uguali**
Il primo caso si risolve utilizzando il **risolutore di visibilità** per risolvere il conflitto. Il secondo caso si può trattare definendo degli **alias** (**sinonimi**). Il nome di un namespace è **estendibile**: ciò consente di **suddividere** un namespace tra più blocchi e di distribuirlo in header file **diversi**. Per alleviare il problema dei namespace con lo stesso nome che raggruppano identificativi uguali si può ricorrere all'uso dei **sinonimi**. E' meglio dare nomi **corti** ai namespace, anche se questo aumenta la possibilità di avere nomi uguali. Per questo è possibile usare nomi elaborati e poi definire un **sinonimo corto** con minor probabilità di conflitto con il nome di altri namespace. I namespace devono essere dichiarati in un **ambiente globale**, cioè **non** è possibile dichiarare un namespace all'interno di un blocco. Tuttavia i namespace possono essere **innestati**.

Domanda: Cos'è una **CLASSE ASTRATTA** e a cosa serve una **CLASSE ASTRATTA PURA** ?
Risposta: Una **CLASSE ASTRATTA** è una classe che nella specifica presenta almeno una **FUNZIONE VIRTUALE PURA**. Una funzione virtuale pura è una funzione che **è dichiarata ma non è definita** (non implementata). Sintatticamente: **virtual T f(…) = 0**. Una classe astratta **non** può essere usata per istanziare oggetti. Una classe astratta è usata **solo come base per la derivazione** e **per definire puntatori ad essa** in modo da attivare il **polimorfismo**. Una funzione virtuale pura deve essere overridden e definita nella classe derivata, in caso contrario la derivata erediterà la funzione pura e diventare astratta a sua volta. Pertanto la classe astratta offre un meccanismo per dettare delle interfacce obbligatorie alle sue derivate.

Domanda: Cos'è la **RICORSIONE** ?
Risposta: Si dice che un oggetto è **RICORSIVO** se è possibile darne una definizione in termini di **sé stesso** (ad esempio il fattoriale). In programmazione la **ricorsione** è una tecnica potente che permette di suddividere un problema da risolvere in **sottoproblemi** simili a quello originale. La ricorsione si applica ad **algoritmi ricorsivi**. La tecnica ricorsiva permette di scrivere programmi eleganti e sintetici. La potenza della ricorsione nasce dalla possibilità di definire un insieme infinito di oggetti con regola finita. Gli algoritmi ricorsivi sono caratterizzati da: una **funzione generatrice** che produce una sequenza, un **predicato vero** e una **funzione ricorsiva** (quindi: 1. Possibilità di formulare l'algoritmo in funzione di se stesso; 2. Condizione di terminazione; 3. Convergenza insiemistica;). La chiamata ricorsiva deve quindi essere subordinata ad una **condizione** che ad un certo istante risulti soddisfatta (il predicato). Il numero di chiamate necessarie viene detto **profondità** della ricorsione. Esistono diversi tipi di algoritmi ricorsivi:
- si parla di **MUTUA RICORSIONE** quando nell'algoritmo una funzione ne richiama un'altra che a sua volta richiama la prima, altrimenti si parla di **RICORSIONE DIRETTA**
- si parla di **RICORSIONE LINEARE** quando vi è solo una chiamata ricorsiva all'interno della funzione, altrimenti si parla di **RICORSIONE NON LINEARE** nel caso in cui le chiamate ricorsive siano più di una
- la distinzione più importante ai fini pratici si ha fra **RICORSIONE DI CODA** (tail recursion) e **RICORSIONE NON DI CODA**. Si parla di **RICORSIONE DI CODA** quando la chiamata ricorsiva è l'**ultima** istruzione eseguita nella funzione. Questo tipo di algoritmo ricorsivo è possibile trasformarlo semplicemente in una versione **iterativa**, che di solito è più efficiente. I linguaggi che realizzano il meccanismo della ricorsione, possono realizzarlo in due modi:
- gestione **LIFO** di più copie della stessa funzione
- gestione mediante **record di attivazione**; c'è un'unica copia del codice del sottoprogramma, ma ad ogni chiamata è associato un record di attivazione (diversi record dello stesso sottoprogramma possono essere presenti nello stack).
Un problema ricorsivo può sempre essere risolto in termini **iterativi** (ma **non** è vero il viceversa). Nel caso in cui la ricorsione è in **coda**, la trasformazione è semplice; se **non** è in coda, c'è bisogno di uno **stack di supporto** che simula ciò che fa lo stack di sistema. Perché usare la ricorsione? La ricorsione permette di scrivere **poche linee di codice** per risolvere un problema anche molto complesso. Algoritmi che per loro natura sono ricorsivi, conviene sempre risolverli in maniera ricorsiva. La ricorsione va evitata quando esiste una **soluzione iterativa ovvia** e in situazioni in cui le **prestazioni del sistema** sono un elemento critico.

Domanda: Perchè nel costruttore di un oggetto di classe derivata si chiama il costruttore di classe base in lista di assegnazione?
Risposta: Perchè la parte base deve essere costruita come prima cosa, dato che la costruzione della parte derivata può dipendere dalla parte base.

Domanda: Descrivere l'eccezione **BAD ALLOC** e scrivere un **codice di esempio** che la utilizzi.
Risposta: Quando la **memoria heap** è **esaurita**, il programma termina con un **messaggio di errore** ed il programma va in **abort** (**BAD ALLOC**). Cosa è successo? L'operatore **NEW** può lanciare un'**eccezione** di tipo **std::bad_alloc** (**delete** invece **non** lancia alcuna eccezione). Nel codice seguente, l'eccezione lanciata da **NEW** viene gestita facendo terminare il ciclo while (memoria insufficiente):

```
#include <iostream>
using namespace std;
int main()
{
int n = 1000000;
int * vett;
while (true) {
try {
```

```
vett = new int[n];
}
catch (bad_alloc e) {
cout << “Memoria insufficiente” << endl;
break;
}
}
cout << “Il programma può continuare…”;
return 0; }
```

Domanda: Fare un esempio di **SERIALIZZAZIONE** e **DESERIALIZZAZIONE** di record su file.

Risposta: La **SERIALIZZAZIONE** di un oggetto consiste nel salvare il suo stato (o parte del suo stato) su un **supporto di memorizzazione lineare**. L'obiettivo è memorizzare lo stato dell'oggetto in modo che esso possa essere **successivamente ricostruito** esattamente con lo stesso stato applicando il processo inverso (**deserializzazione**). La serializzazione può essere fatta in forma **binaria** o **testuale**. I vantaggi sono:

- è un mezzo potente per lavorare con oggetti persistenti
- consente facilmente di distribuire oggetti in un ambiente distribuito
- è di supporto alle chiamate di procedura remota

Ha lo svantaggio che mina l'incapsulamento delle informazioni. Inoltre presenta il problema che serializzare un oggetto di classe A richiede la conoscenza di tutte le superclassi di A e di tutte le classi di oggetti variabili membro di A. Inoltre cosa fare con variabili membro di tipo puntatore?

Serializzazione:

```
bool serialize(const string & filepath, const Studente Studenti[], const int n)
{
ofstream out(filepath.c_str(), ios_base::binary);
if(!out)
return false;
size_t count = n;
out.write((char*)&count, sizeof(count));
for (unsigned int i=0; i<count; ++i)
{
write(out, Studenti[i].getId());
write(out, Studenti[i].getCognome());
write(out, Studenti[i].getNome());
}
return true;
}
```

Deserializzazione:

```
bool deserialize(const string & filepath, Studente Studenti[], const int n)
{
ifstream in(filepath.c_str(), ios_base::binary);
if(!in)
return false;
int count = n;
in.read((char*)&count, sizeof(count));
for (int i=0; i<count; ++i)
{
string Id, Nome, Cognome;
read(in, Id);
read(in, Cognome);
read(in, Nome);
Studenti[i]=(Studenti(Id, Nome, Cognome));
}
return true;
}
```

Domanda: Scrivere i codici di **PUSH** e **POP** per una lista linkata (può chiedere qualsiasi funzione di un contenitore).

Risposta:

PUSH:

```
bool push(const T& t)
{
if(full()) return false;
nodo* nuovo;
nuovo = new nodo;
nuovo -> elem = e;
nuovo -> next = testa;
testa = nuovo;
return true;
}
```

POP:

```
{
if(empty()) return false;
nodo* temp = testa;
e = temp -> elem;
testa = testa -> next;
delete temp;
return true;
}
```

Domanda: **Compatibilità** tra classe **base** (b) e classe **derivata** (d).

Risposta:

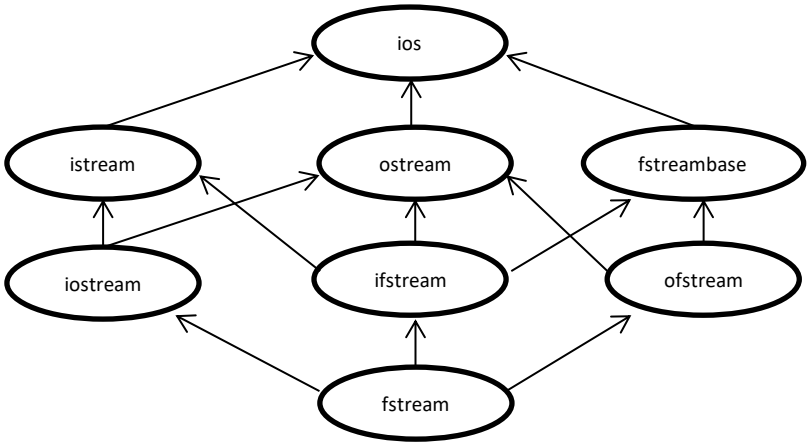
b = d -->> OK! Si può scrivere perché **d ha più componenti di b**

d = b -->> NO! Non si può scrivere perché **b ha meno componenti di d**

- **d può essere assegnato a b** mediante l'**operatore di assegnazione di default**
- **il sotto-oggetto b di d viene assegnato a b** (**object-slicing** = affettamento dell'oggetto)

Domanda: Gerarchia delle librerie di **STREAM**.

Risposta:



Domanda: Fare un esempio di **POLIMORFISMO.** Chiamare un funzione (operatore freccetta).

Risposta:

```
class Persona {
protected:
//…
public:
//…
virtual void print() const;
```

```
class Impiegato : public Persona {
private:
//…
public:
//…
virtual void print() const;
```

```
int main ()
{
Persona p1 (“Rossi”, “Mario”);
p1.print(); //binding statico
Impiegato i1(“Gargano”, “Marco”, “AnsaldoSTS”, 2);
i1.print(); //binding statico

Persona * Pptr;

Pptr=&p1;
Pptr->print(); //binding dinamico perché print è virtual
Pptr=&i1;
Pptr->print(); //binding dinamico perché print è virtual
```

Domanda: Quando è meglio **NON** usare le **ECCEZIONI**?
Risposta: Non bisogna usare le eccezioni quando le strutture di controllo locali sono sufficienti.

Domanda: Qual è la differenza tra **static const** / **static** / **const** ?
Risposta:
const = una variabile dichiarata **const** non varia (diventa **costante**).
static = una variabile **static** resta **condivisa** per ogni istanza di una classe. Non ci sono più versioni, ma **una sola**.
static const = è una **static** che **non può variare** (condivisa e costante).

Domanda: Qual è la differenza tra funzioni amiche, funzioni membro e funzioni ordinarie ?
Risposta:
amiche = funzione **NON** membro che controlla l’accesso di una funzione **NON** membro che accede ai dati della classe.
membro = qualsiasi funzione **non static** interna ad una classe.
ordinarie = funzione **classica**.

Domanda: Perché il distruttore di una classe base deve essere **virtual**?
Risposta: Serve se usi il **polimorfismo**, se hai un **puntatore al distruttore della classe base**, se vuoi che venga chiamato il distruttore della classe derivata e **non** quello della classe base. Se non metti **VIRTUAL**, il **comportamento non è definito** (dipende dal compilatore).

Domanda: Qual è la differenza tra parametro formale e parametro effettivo?
Risposta:
formale = **copia** al livello **locale** nello **scope** della funzione (se modifichi un parametro **formale**, stai modificando una copia dell’**effettivo** ma **NON** l’effettivo proprio).
effettivo (detto anche “**argomento**”) = è quello che **passi** ad una funzione.

Domanda: Scrivi un algoritmo di **ricerca incerta** in una **lista** (ad esempio ricerca tutte le persone che hanno una certa età).
Risposta: (dovrebbe essere così ma NON sono sicuro al 100%)

```
bool ricerca (E& e, int eta)
{
  if(empty()) return false;

  nodo* temp = testa;
  bool trovato = false;

  while(temp && !trovato)
  {
    if (temp --> elem == eta)
      {
        trovato = true;
        cout << "Elemento trovato!" << endl;
      }
      else
      {
        cout << "Elemento non trovato..." << endl;
        temp = temp --> next;
      }
  }

return trovato;
}
```

Domanda:Parlare delle classi di memorizzazione.
Risposta: Per le variabili in C++, la definizione specifica la classe di memorizzazione, cioè:
- la visibilità spaziale,
- la durata temporale (ciclo di vita),
e
- l'allocazione in memoria durante l'esecuzione

Classi di memorizzazione del C++ (storage classes):
- variabili automatiche
- variabili esterne
- variabili automatiche
statiche
- variabili esterne statiche
- variabili register
- variabili dinamiche

Modello di gestione della memoria
per l'esecuzione di un processo

Area programmi e costanti: destinata a contenere le istruzioni (in linguaggio macchina) e le costanti del programma
Area dati statici: destinata a contenere variabili allocata staticamente e quelle esterne (globali)

Area heap (a mucchio): destinata a contenere variabili dinamiche esplicite, di dimensioni non prevedibili a tempo di compilazione
Area stack: Destinata a contenere le variabili automatiche e quelle definite all'interno delle funzioni

Variabili Automatiche
Sono le variabili locali ad un blocco di istruzioni.

- Sono definite in un blocco di istruzioni e sono allocate e deallocate con esso.
- La visibilità lessicale è locale al blocco.
- Sono allocate nell'area stack

Variabili Esterne

Sono le variabili globali e vengono definite all'esterno di ogni
blocco.
- La visibilità lessicale si estende a tutto il programma, ma per renderle visibili anche da procedure contenute in file diversi, devono essere ivi dichiarate con la parola chiave extern.
- La loro estensione temporale si estende a tutto il programma (sono allocate all'inizio del programma per essere poi deallocate alla fine
dello stesso).
- In memoria vengono allocate nell'area dati statici o globale

Variabili automatiche statiche

Sono variabili automatiche prefissate con la parola chiave static.
- La visibilità lessicale è locale al blocco in cui sono
definite.
- La loro estensione temporale si estende a tutto il programma (sono allocate all'inizio del programma per essere poi deallocate alla fine dello stesso).
- In memoria vengono allocate nell'area dati globale.

Variabili esterne static

Sono variabili esterne prefissate con la parola chiave static.
- La visibilità lessicale è locale al file in cui sono definite.
- La loro estensione temporale si estendea tutto il programma
(sono allocate all'inizio del programma per essere poi deallocate alla fine dello stesso).
- In memoria vengono allocate
nell'area dati globale.

Variabili register

Sono variabili automatiche prefissate con la parola chiave register.
- La visibilità lessicale è locale al blocco in cui sono definite.
-Sono allocate e deallocate con il blocco di istruzioni in cui sono contenute.
- Se possibile vengono allocate in un registro del processore, altrimenti nell'area stack.
- Può convenire dichiarare register una variabile automatica usata di frequente, in modo che l'allocazione in un registro di macchina ne aumenti la velocità di accesso.

Variabili dinamiche

Sono definite durante l'esecuzione del programma mediante puntatori.
- Sono accessibili dovunque esiste un riferimento ad esse.
- L'allocazione e la deallocazione sono controllate dal programmatore.
- In memoria vengono allocate nell'area heap.

Domanda: Cos'è un puntatore?

Risposta: Una variabile di tipo puntatore può assumere come valore un indirizzo di memoria.
Non esiste un unico tipo puntatore, ma un insieme di tipi puntatori a seconda del tipo dell'oggetto puntato.
Non può contenere un indirizzo qualunque: **esiste un vincolo di tipo.**
Il C/C++ prevede puntatori a funzione e puntatori a dati di qualsiasi natura,semplici o strutturati.(Esempi:variabili dinamiche,funzioni,array,parametri formali,per riferimento a locazioni specifiche della memoria).E' buona normal associare il valore 0 o "NULL" alla definizione.
L'indirizzo può essere ottenuto attraverso l'operatore "**&**",mentre l'operatore di de-referenziazione "*****" restituisce il valore puntato.E' possibile definire un puntatore generico attraverso "void".
Un utilizzo dei puntatori può essere per gli array, la variabile puntatore punta alla prima locazione del'array.
Mentre per accedere alla singola componente di un record si utilizza "->".

Utilizzo di const con i puntatori:

-const T* p: l'oggetto puntato non può essere modificato attraverso p

-T* const p:il puntatore p deve essere inizializzato alla sua definizione e non può essere modificato
- const T* const p:Il puntatore p deve essere inizializzato alla sua definizione e il dato puntato non può essere modificato attraverso il punattore.

Altre due domande:

- Scrivere una funzione che, data una lista e un elemento x, sommi tutti gli elementi della lista diversi da x.

- Date 3 funzioni innestate (f1 che chiama f2 che chiama f3) tra cui una (f3) lancia un'eccezione, determinare il comportamento del programma che chiama f1.