Domande Esame Orale

Il materiale è una reinterpretazione dei documenti dello studente Alex Parri ed uno senza firma, consiglio di consultare il materiale di Gabriele Frassi per la parte della rappresentazione dell'informazione. Il contenuto non è esente da errori di scrittura ed il materiale non intende sostituirsi a quello fornito dal docente e l'autore non si assume nessuna responsabilità dell'uso e dei risultati derivanti dall'uso del seguente materiale, non basate il vostro studio unicamente su di esso ma solo come punto di riferimento, le risposte effettive all'esame solo più articolate e pratiche all'esame.

1. Introduzione

Cosa è un metalinguaggio?

È un linguaggio che ci permette di descrivere la sintassi e la semantica di un ulteriore linguaggio. Per descrivere il linguaggio C++ vengono adoperate notazioni non ambigue che possono essere spiegate mediante il linguaggio naturale. Per la semantica adoperiamo il linguaggio naturale. La notazione deriva dalla forma di Backus Naur.

Tipi fondamentali e derivati

I tipi **predefiniti** (aritmetici) sono i tipi base del C++:

- int;
- unsigned;
- double;
- bool;
- char;

Insieme a **enum** sono i tipi **fondamentali**.
I tipi **derivati** sono ottenuti a partire dei **predefiniti** e permettono di costruire strutture più **complesse**:

- reference;
- puntatori;
- array;
- struct;
- union;
- class;

Possiamo concatenare più tipi derivati per aumentare la complessità.

Quanto occupano i tipi primitivi ed il massimo storage value

- **bool**: 1 bit [0,1];
- **char**: 1 byte [0, 2^8-1];
- **short int**: 2 byte [-2^{16-1}+1, 2^{16-1}-1];
- **unsigned int**: 2 byte [0, 2^{16}-1];
- **int**: 4 byte [-2^{32-1}+1, 2^{32-1}-1];
- **float**: 4 byte;
- **long int**: 8 byte;
- **double**: 8 byte [-2^{64-1}+1, 2^{64-1}-1];
- **long double**: 12 byte [-2^{96-1}+1, 2^{96-1}-1];

Come si rappresentano i numeri reali? Cos'è il tipo enum?

I numeri reali vengono rappresentati attraverso il tipo **double** ed occupano 32 bit in memoria, **float** su 16 bit e **long double** su 64 bit. Sui reali sono definite tutte le operazioni aritmetiche e di confronto.
L'enumeratore è di tipo derivato che viene definito dal programmatore che quindi crea un nuovo tipo. Queste variabili possono assumere solo i valori dichiarati dall'utente.

```
enum giorni {lunedi, martedi, mercoledi, ecc...}
```

- Le costanti dichiarate all'interno del enum giorni vengono viste come costanti numeriche a partire da 0 per lunedì e 6 per domenica (conversione automatica);

```
cout << giorni; //se giorni è lunedì, stampa 0
```

- Creando una variabile di tipo giorni gli si può assegnare solo le costanti dichiarate all'interno del enum giorni (e non interi);

```
giorni = martedi;
giorni = 1; //illecito
```

- Il tipo enum viene usato per le operazioni di confronto;

Cosa sono i literals?

Sono costanti letterali, o literals, tutte quelle **codifiche** che un programma riceve da **tastiera** e rappresentano il valore che assume l'oggetto a cui è stato fatto l'assegnamento:

```
int a=5; //5 è un literal che rappresenta il numero naturale 5
```

Cos'è la conversione implicita?

Il compilatore esegue implicitamente delle conversioni di un tipo di variabile in un'altra:

- Tra int e bool
  bool trovato=1; //1 convertito in true
- Tra enum e int
  cout << giorni; //stampa l'indice dell'enum
- Tra reali e naturali
  int n=15.5; //troncamento in 15
- Tra char e int
  if (c >='a' && c <='z') {...} //carattere convertito in intero
- Tra argomenti e classi
  c1=4.0+3.0; //chiamata dell'operatore + con argomenti 4.0 e 3.0, poi chiamata al costruttore di copia con il complesso appena creato e c1.

Differenza tra approccio compilato e interpretato

Un programma si può sviluppare in due approcci principali: il compilato ed interpretato.

- **Compilato**: caratterizzati dalla presenza di un **linker** e di un **compilatore**, il quale compila il codice sorgente in linguaggio macchina: per ogni file esegue una analisi sintattica del codice controllando che siano rispettate le regole di sintassi del linguaggio e successivamente converte ogni file in un **object** file contenente **linguaggio macchina** ottimizzato e, dopo il linking (unione devi file object con le librerie), si ha il file pronto per essere inviato all'utilizzatore.
  Vantaggi: compilazione più veloce dell'interpretazione, il codice sorgente rimane privato in quanto i file sono tradotti e trasformati in linguaggio macchina.
  Svantaggi: non cross platform, ogni volta che facciamo il debug, il programma deve ricompilare tutto il programma, può creare problemi se i programmi sono di grandi entità.
  Esempio: C, C++
- **Interpretato**: non hanno un compilatore ma un **interprete**, **non** viene eseguita la conversione in codice ed il file viene tradotto in codice macchina **riga per riga** dall'utilizzatore.
  Vantaggi: cross platform (basta scrivere il codice e l'utilizzatore con l'interprete traduce il source code in linguaggio macchina e lo esegue), non c'è la compilazione ogni volta che bisogna debuggare.
  Svantaggi: l'interpretazione è più lenta della compilazione, il source code è noto all'utilizzatore.

Esempio: Javascript, Python, PHP

Cos'è il linker?

Il linker è presente in tutti i linguaggi con approccio compilato, insieme al compilatore. Il suo ruolo succede il compilatore ed ha diversi ruoli:

- Combinare tutti gli object file generati ed unirli in un singolo eseguibile;
- Collegare i file libreria, incluse con #include. La maggior parte dei linker collega **automaticamente** la C++ standard library.
- Controllare che non si siano dipendenze non risolte tra i file.

Quando il linker ha finito il suo compito si ha l'eseguibile (.exe) pronto.

Cos'è un Segmentation Fault? (SIGSEGV)

Il Segmentation Fault è un'eccezione lanciata a tempo di compilazione dal compilatore. SIGSEGV sta a significare che il software sta cercando di accedere ad una zona di memoria riservata oppure inesistente. Si verifica spesso quando si lavora con i puntatori:

```
char* c=NULL;
*c=5; //dereferenziare una zona di memoria inesistente: SIGSEGV
char* c="ciao"; //questo non si può modificare, si può riassegnare
c[10]='z'; //scrivere in read-only: SIGSEGV
char* c=new char [10];
delete [] c;
c[2]= 'z'; //accedere a memoria deallocata: SIGSEGV
...
delete [] c;
delete [] c; //double free: segnato come errore su Linux ma non su
Windows: SIGSEGV
```

Stesso concetto quando si prova ad accedere o modificare puntatori delle struct utilizzando "->".

2. Operatori

Quali sono gli operatori Bit a Bit?

- **|** (**OR** inclusivo): vale 1 se almeno un bit è 1;
- **&** (**AND**): vale 1 solo se entrambi i bit valgono 1;
- **^** (**OR** esclusivo): vale 1 solo se 1 bit è 1 e l'altro 0;
- **NOT** (complemento): cambia il valore del bit;

Si applicano sugli unsigned.

Quali sono gli operatori logici?

Il loro impiego è quello di controllo sulle variabili. Vediamo quelli principali:

- **AND**: operatore binario che verifica se due o più condizioni sono verificate contemporaneamente. La sintassi è '&&';
- **OR**: operatore binario che verifica che almeno una delle condizioni siano vere. La sintassi è '||';
- **NOT**: operatore unario che controlla che una condizione sia falsa. La sintassi è '!'.

Operatori di shift dei bit

Sui tipi interi della memoria (int, long, short, byte, char) si può effettuare l'operazione di "bit shift", ovvero shiftare a destra o a sinistra tutti i bit dell'intero. Un buon compilatore riesce ad utilizzare il più possibile lo shift dei bit per effettuare le operazioni di moltiplicazione/divisione tra numeri anche abbastanza grandi perché è un'operazione molto leggera. Lo shift si esegue con l'identificatore del numero seguito da una coppia di segni maggiore ">>" o minore "<<" in base alla direzione e infine un literal che specifiche di quante posizioni avviene lo shift. I bit **non ciclano** se vanno "troppo a destra" o "troppo a sinistra" ma sono **persi** (i "nuovi" bit generati dello shift a sinistra sono invece degli zeri).
Si può dedurre che l'operazione di bit shift equivale ad una moltiplicazione per potenze del 2 di $2^{literal}$ con direzione = segno:

- Shift a **destra**: non è un'assegnazione

```
a >> 1 //a=01010010 -> 00101001
```

- Shift a **sinistra**: non è un'assegnazione

```
a << 1 //a=01010010 -> 1010010
```

- Shift e **assegnazione**: è un'assegnazione, il numero risultante viene salvato in a

```
a <<=1
```

L'operatore di shift effettua questa operazione attraverso la formula: a << n % sizeof(a):

```
int a=5;
a >> 32; //ritorna 5 dato che 32%(4*8bit) fa 0, equivale ad a >> 0
a >> 33; //ritorna 2 dato che 33%(4*8bit) fa 1, equivale ad a >> 1
```

Cos’è il cortocircuito?

Questa tecnica riguarda gli operatori logici **AND** e **OR**. Quando valutiamo una condizione con l’**AND**, se la prima è falsa non viene valutata anche la seconda mentre con l’**OR** se la prima è vera la seconda non viene valutata. Così facendo evitiamo errori di dereferenziazione a NULL di un puntatore.

Operatori di incremento

La differenza tra incremento **prefisso** e **postfisso** è il momento in cui avviene l’incremento.
Il **prefisso** restituisce un **lvalue** incrementando prima la variabile per poi essere utilizzata.

```
cout <<++a; // prima incrementa poi stampa
```

Nel **postfisso** prima utilizza la variabile poi la incrementa, restituisce un rvalue rendendo impossibile la concatenazione.

```
cout <<a++; //stampa poi incrementa
```

Proprietà degli operatori

- **Posizione**: prefissa, infissa, postfissa;
- **Priorità**: ordine di esecuzione di diversi operatori nella stessa espressione;
- **Arietà**: numero di oggetti su cui può agire un operatore;
- **Associativà**: a parità di priorità determina quali operatori devono essere usati per primi. Può essere destra oppure sinistra.

Left e right value

Un left-value è un valore a sinistra dell’operatore di assegnamento che definisce una zona di memoria in cui è possibile la scrittura. Un right-value si trova a destra dell’operatore di assegnamento è determina una zona di lettura, un valore senza indirizzo oppure una costante. Tutti i left-value sono anche right-value, dove è possibile scrivere è anche possibile leggere, ma non il contrario.

3. Puntatori e riferimenti

Cosa sono i puntatori?

Sono un particolare **tipo di variabile** che hanno come **valore** l'**indirizzo** di una variabile.

Come si creano?

I puntatori devono avere lo **stesso tipo** della variabile alla quale puntano aggiungendo un asterisco *****, così creiamo un tipo derivato, ed infine diamo un **identificatore** alla variabile.

```
int* p=a; //puntatore di tipo int che punta all'indirizzo di a
```

Come si usano?

Un **puntatore** deve **sempre** essere **inizializzato** con l'indirizzo di una **variabile** oppure con **NULL** per evitare puntamenti di **indirizzi casuali**. Con la **dereferenziazione** possiamo accedere **direttamente** alla **variabile** puntata in modo da **modificarla**.

```
*p=5; //modifico valore variabile puntata da p in 5
```

Nel caso in cui il puntatore punta ad una struct utilizziamo la freccia "->":

```
elem* p=p0; //p0 deve essere un elem*
p->inf=5; //dereferenziazione del membro inf
```

Possiamo anche memorizzare il primo elemento di un array o di una matrice:

```
p=&v[0];
p=v;

p=m;
p=&m[0][0];
p=&m[0];
```

Un puntatore può avere solo indirizzi e non altro.

Quanto occupano in memoria?

Come gli interi, un puntatore occupa **4 byte**.

Sono possibili degli errori con i puntatori non inizializzati?

Se un **puntatore non** è **inizializzato** ad una variabile oppure a NULL punta in modo randomico ad un indirizzo di memoria rischiando **Segmentation Fault**.

Aritmetica dei puntatori

L'**aritmetica** dei puntatori rappresenta una **serie di operazione** che possiamo eseguire sui puntatori.
Per esempio, se utilizziamo un puntatore per puntare un vettore, il puntatore prende l'indirizzo del primo elemento, ma incrementando il puntatore passiamo all'elemento successivo, essendo esso presente nella cella di memoria successiva. Possiamo scorrere un un vettore oppure una matrice con la seguente espressione: (p+i*n-colonne+j) detta anche **matrice linearizzata**. Attenzione alle parentesi:

```
int* p=v; //v[0]
p++; //v[1]
p+=2; //v[1+2]=v[3]
*p=5; //v[3]=5;
(*p)++; //v[3]++
*(p++)+=1; //v[4]++
```

Differenza tra puntatori a costante e puntatori costanti?

Ci sono due rappresentazione della keyword con i puntatori:

```
int const* b=&a;
```

Questo puntatore **non** può **modificare** il **valore** puntato.

```
int* const b=&a;
```

Questo puntatore è **costante**, deve essere **inizializzato** e l'**indirizzo** a cui punta **non** può **variare**.

Cosa rappresenta il nome del puntatore?

Il nome del puntatore rappresenta l'identificativo che si riferisce all'indirizzo della cella di memoria occupata dalla variabile che ad esso è assegnata.

Cosa significa il simbolo '&'?

La sua interpretazione varia in base al contesto:

```
int* p=&a; //p punta all'indirizzo di a
```

```
int &a=b; //sto creando una reference che referenzia b
if (op1 && op2) //qui è un AND logico
cout <<a & b; //0100 & 0110 = 0100 = 4 //AND operatore bit a bit
complesso& scala (); //rende possibile la concatenazione con il
return* this di una funzione
```

Cosa sono i riferimenti?

I riferimenti (reference) sono dei tipi derivati che **referenziano** un'altra variabile dello stesso tipo della reference, spesso l'identificatore è un sinonimo dell'identificatore della variabile che referenzia:

```
int& felino=gatto; //felino referenzia gatto
```

Una proprietà importante dei riferimenti è la possibilità di modificare la variabile referenziata modificando il riferimento:

```
felino=5; //sto assegnando 5 a gatto tramite felino
```

I riferimenti trovano largo impiego nel passaggio per reference ad una funzione, perché le modifiche sul parametro formale hanno effetto su quello attuale. La dichiarazione deve essere accompagnata da un'immediata inizializzazione e la variabile referenziata non può variare.

```
int& felino;
int& felino=tigre; //se felino=gatto non può diventare
felino=tigre
```

In quale circostanza una funzione membro ritorna un oggetto reference?

L'oggetto reference viene ritornato quando:

- Si vuole rendere **concatenabile** la funzione membro, quindi per permettere la possibilità di ritornare l'indirizzo di memoria dell'istanza di classe con*this;
- Se la funzione membro NON ritorna un oggetto nuovo ma l'istanza di classe stessa è quindi necessario avere la possibilità di ritornare il suo indirizzo di memoria tramite &;

Se ritorna un oggetto nuovo, deve serve la reference perché l'oggetto creato è appunto nuovo (reference si utilizza per modificare gli oggetti già presenti in memoria). **NB**: gli operatori aritmetici sono concatenabili di default.

```
class& operator += (const class&); //modifica di un oggetto
c1 += c2 += c4
```

```
class operator * (const class&); //creazione di un oggetto nuovo
c1 * c2 * c4
```

Differenze tra passaggio per valore e per riferimento

Tramite il passaggio per valore avviene una copia dei valori attuali in quelli formali e le eventuali modifiche a questi ultimi non si ripercuotono su quelli attuali. Con il passaggio per riferimento andiamo a copiare l'indirizzo di memoria della variabile assegnata potendo modificarne il contenuto.

Differenza tra argomenti formali ed attuali

Gli **argomenti formali** sono dei **contenitori** inizializzati con gli argomenti attuali passati alla funzione. Ad eccezione dei puntatori o delle reference, i **parametri formali copiano** il **valore** di quelli attuali ma non ne modificano il valore ed alla fine del blocco vengono distrutti. Le **variabili attuali** sono variabili dichiarate **fuori** dal blocco e **vengono** distrutte al **termine** del **programma**.

Effetti collaterali

Detti anche Side-Effects, si hanno quando una funzione modifica degli elementi fuori dalla funzione e può avvenire solo se abbiamo a che fare con riferimenti e puntatori.
Dato che i puntatori possono essere valori di ritorno di una funzione, se il puntatore che restituiamo, punta ad una variabile locale della funzione, commettiamo un errore logico, alla fine dell'esecuzione della funzione la variabile verrà deallocata ed il puntatore punterà ad una zona di memoria il cui valore non sarà più quello della variabile locale alla funzione.

4. Classi

Cosa sono le classi?

Le classi sono delle particolari **strutture dati** che permettono di definire un nuovo **tipo** di dato **astratto**. Si differenziano da una struct perché i membri possono essere suddivisi in parte **privata** e parte **pubblica**. La parte **privata** può essere raggiunta dalle funzioni membro o dalle funzioni **friend**. La classe è un **insieme** di **funzioni** e variabili che trattano lo stesso argomento e **sfruttano** gli **stessi elementi** per svolgere **azioni**, permettendo anche l'**overloading** di operatori.

Cos'è un costruttore?

I costruttori sono delle particolari funzioni membro con la stessa denominazione della classe, essi servono ad inizializzare le istanze della classe. Esiste un costruttore predefinito della classe (default), ma possiamo anche definire dei costruttori "personalizzati", i quali possono essere di tipi diversi:

- Costruttore con argomenti;
- Costruttore senza argomenti (default);
- Costruttore di conversione;

Il compilatore non viene invocato dall'utente ma dal compilatore attraverso le regole di chiamata:

- Oggetti statici: all'inizio del programma;
- Oggetti automatici: quando viene incontrata la dichiarazione;
- Oggetti dinamici: quando viene incontrato l'operatore **new**;
- Oggetti membro di un oggetto: quando viene chiamato il costruttore di quest'ultimo;

Non è lecito definire un oggetto classe senza inizializzarlo se presenta solo un costruttore con parametri. Se viene passato un parametro formale errato il costruttore deve comunicarlo con un **exit(1)** e **non** return, è necessario questa accortezza per evitare di lavorare su oggetti non creati effettivamente.

Cosa sono i distruttori?

Il distruttore è una funzione membro della classe, la cui sintassi è ~Classe () e non ha parametri oppure valori di ritorno, che ha come compito di deallocare la memoria utilizzata dagli oggetti della classe di appartenenza. È importante definirli quando la classe utilizza l'allocazione sulla memoria dinamica e, come per i costruttori, non possiamo richiamarli esplicitamente ma viene richiamato dal compilatore secondo le regole di chiamata:

- Oggetti statici: fine del programma;
- Oggetti automatici: alla fine del blocco di definizione;
- Oggetti dinamici: quando viene incontrato l'operatore **delete**;
- Oggetti membro di un oggetto: quando questi ultimi vengono distrutti;

Gli oggetti allocati in memoria vengono deallocati in ordine opposto rispetto all'allocazione, viene prima deallocato l'oggetto inserito per primo e così via.

Operatori di conversione

L'operatore di **conversione** converte il **tipo classe** in un **tipo fondamentale** con la sintassi: operator type (), il **valore** di **ritorno** si **omette** perché è **implicito** nel nome.

Operator

Gli operator sono delle funzioni composte dalla keyword operator seguito dall'identificatore che si vuole ridefinire. Il linguaggio C++ ci permette di rimodellare gli operatori che operano sul tipo classe. Non possiamo ridefinire gli operatori dei tipi fondamentali oppure inventarne di nuovi. Gli operatori possono essere funzioni friend oppure membro:

- Operatori unari: 1 parametro se **friend**, 0 se **membro**;
- Operatori binari: 2 parametri se **friend**, 1 se **membro**;

Se ridefiniamo un operatore come funzione membro, l'operando di sinistra sarà sempre di tipo classe, per mantenere la simmetria degli operatori utilizziamo le funzioni friend. Gli unici operatori non ridefinibili sono:

- Risolutore di visibilità cioè ::
- Selettore di membro cioè .
- Selettore di membro attraverso puntatore a membro .*

Gli operatori di assegnamento, di indicizzazione e di chiamata di funzione devono essere sempre ridefiniti come funzioni membro.

Cosa succede se richiamo un operatore non ridefinito?

Non compila.

Costruttore di copia

Il costruttore di copia è una funzione che ha lo stesso nome della classe e si occupa di effettuare una copia membro a membro di tutte le variabili già inizializzate dal costruttore così da ricreare un oggetto identico a quello che si intende copiare. Il costruttore di copia esiste predefinito anche senza ridefinirlo ma la sua ridefinizione è necessaria quando i membri sono troppo complessi oppure abbiamo membri allocati in memoria dinamica, così da evitare che un puntatore punti all'elemento dinamico e comprometta la corretta copia. Richiamiamo il costruttore di copia quando:

- Un parametro formale è di tipo classe;
- Viene ritornato il tipo classe in un return;
- Viene richiamato esplicitamente;
- Utilizziamo l'operatore di assegnamento di classe non ridefinito, in questo caso il costruttore di copia è chiamato implicitamente (non vale il contrario);

La sintassi è:

```
Class (const Class&)
```

Passiamo per riferimento perché cercheremo di richiamarlo infinite volte il costruttore di copia che definiamo mentre costante perché non vogliamo modificare l'elemento che stiamo copiando.

Differenza tra operatore di assegnamento e costruttore di copia

Il **costruttore di copia** esegue una **copia membro a membro** dell'oggetto fornitogli nell'argomento formale mentre l'operatore di assegnamento **operator =** funziona similmente con delle differenze:

- **Dichiarazione** con **return type** del tipo classe seguito dalla reference per la concatenazione e con un singolo argomento formale const Class&;
- **Check dell'aliasing** utilizzando il puntatore this, nel quale controlliamo che il parametro formale non sia l'istanza della classe stessa in tal caso non si esegue nulla.

```
if (this==&p) return* this;
```

- **Copiatura membro a membro** basandosi sui valori dei membri dell'argomento formale;
- In caso di **memoria dinamica**, **deallocazione** e successiva **allocazione** di lunghezza medesima alla lunghezza degli elementi dell'argomento formale;
- Se la lunghezza è già la medesima, si sostituiscono semplicemente;
- Ritornare ***this** per la concatenazione.

Funzioni friend

Sono le uniche **funzioni** assieme a quelle membro che **possono accedere** alla parte **privata** di una classe, esse devono essere dichiarate nella classe e **definite** come funzioni **globali**. Possono essere **friend** anche altre **classi** oppure **funzioni** membro di altre classi.

Cosa sono i membri statici?

I membri **static** sono membri che si **riferiscono** all'**intera classe** e non ad una singola istanza. Sono spesso utilizzati per contare il numero di istanze attive contemporaneamente in una classe.
Le funzioni membro **static non** possono accedere né alla **parte privata** della classe **né** possono utilizzare il **puntatore this** perchè si riferiscono alla classe e non all'oggetto. Possono essere richiamate usando il risolutore di visibilità ::

Keyword static

Static è una keyword con più utilizzi:

- Modificare il **collegamento** di una variabile da esterno ad **interno**;
- Se utilizzata all'interno di un blocco per definire un oggetto statico, esso manterrà il suo valore anche alla fine del blocco;
- Può rendere una variabile di una classe globale per lasciarla inalterata e condivisa rispetto alla creazione di istanze di classe;

Keyword extern

Eseguire una dichiarazione con **extern** ci permette di accedere ad una funzione/oggetto presente in un'altra unità di compilazione. Per default, gli identificatori a livello di blocco hanno collegamento interno, mentre quelli a livello di file hanno collegamento esterno. Extern permette di condividere variabili/funzioni tra linguaggi diversi, i compilatori però devono però rispettare degli standard di compatibilità.

Funzioni membro con attributo const

Funzioni che non possono accedere in scrittura ai membri dati.

Lista di inizializzazione

Una lista di inizializzazione è la parte iniziale di un costruttore di classe. Questo definisce il costruttore in due parti diverse quali lista di inizializzazione e corpo del costruttore. La lista esiste se nella classe sono stati dichiarati dei membri stato **const**. La sintassi prevede che, dopo la chiusura delle parentesi tonde dell'argomento del costruttore, inseriamo i : per inizializzare i membri **const** con un valore oppure con un argomento formale del costruttore.

```
Complesso (float n1, float n2) : re(n1), im(n2)
```

Array oggetti di tipo classe

Creando degli array con oggetti di tipo classe, al momento dell'inizializzazione i singoli componenti possono essere inizializzati con il costruttore con argomenti, quelli che non vengono inizializzati così lo faranno in automatico con il costruttore di default.
Possiamo allocarli anche nella memoria dinamica però l'operatore new è al più binario per questo verranno inizializzati con il costruttore di default quindi è necessario che sia presente.

Puntatore* this

Puntatore delle funzioni membro che si riferisce all'oggetto su cui agiscono, non è presente nelle funzioni static.

5. Array e matrici

Cosa sono le stringhe?

Il tipo stringa non esiste in C++, per far fronte a questa mancanza utilizziamo gli array di caratteri come stringhe. Ogni stringa termina con il carattere '\0' che, mettendo lo stream in una situazione di errore, interrompe le operazioni sulla stringa. Possiamo eseguire svariate operazioni sulle stringhe attraverso l'uso della libreria <cstring>:

- **strcpy**: copia il contenuto di una stringa **sorgente** in una **destinazione**, se la stringa destinazione ha dimensione minore non causerà nessuna eccezione ma c'è la possibilità di una perdita di dati;
- **strncpy**: simile alla **strcpy** però con una troncatura dopo un determinato numero di caratteri;
- **strcmp**: confronto tra due stringhe che ritorna 0 se le stringhe sono uguali;
- **strlen**: ritorna la lunghezza di una stringa;

Se non conosciamo la lunghezza in fase di inizializzazione ci appelleremo successivamente all'operatore **new** per allocare un array dinamico. La dimensione delle stringhe deve essere sempre +1 per includere anche il carattere di fine stringa.

Cosa sono gli array?

Un array è un sequenza di elementi dello stesso tipo memorizzato in celle contigue. Essendo un tipo derivato, non possiamo eseguire operazioni di assegnamento o confronto. Possiamo riferirci ai singoli componenti attraverso l'uso di [i], i-esima posizione.

Come si impedisce la modifica di un array?

Bisogna aggiungere la keyword **const** prima del tipo ricordandosi di **dichiararlo** ed **inizializzarlo immediatamente** con i valori necessari.

Come trovare la lunghezza di un array?

Utilizziamo il metodo **sizeof ()** con argomento l'array diviso il sizeof () del suo tipo:

```
int len = sizeof(v) / sizeof (int) //se v è un vettore di interi
```

Questo metodo **non funziona** se all'interno di una **funzione** che come **argomento formale l'array** anche se esso è passato per riferimento.

Cosa sono le matrici?

Le matrici oppure array multidimensionali sono anch'esse degli array avendo però due dimensioni memorizzate per righe. Sia per gli array che per le matrici possiamo riferirci ai singoli componenti attraverso l'uso dei puntatori:

```
&v[0]=v;
v[i]=*(v+i);
```

Possiamo accedere ai singoli componenti anche così:

```
*(v+i*C+j) equivale a [v+i*C+j]
```

C è il numero di colonne, i e j sono gli indici riga e colonna, in questo modo la matrice si dice **linearizzata**.

Definizione di algoritmo

Sequenza precisa e finita di operazioni che portano alla realizzazione di un compito. Le operazioni utilizzate appartengono ad una delle seguenti categorie:

- Operazioni **sequenziali**: realizzano una singola azione. Quando l'azione è terminata passano all'operazione successiva;
- Operazioni **condizionali**: controllano una condizione. In base al valore della condizione, selezionano l'operazione successiva da seguire;
- Operazioni **iterative**: ripetono l'esecuzione di un blocco di operazioni, finché non è verificata una determinata condizione;

Gli algoritmi vengono eseguiti in modo corretto solo se essi sono codificati in un linguaggio comprensibile al calcolatore, vengono eseguiti e produrranno un risultato.

Proprietà degli algoritmi

Gli algoritmi devono essere:

- **Corretti**: perviene alla soluzione del problema direttamente;
- **Efficienti**: necessitano del minor numero di risorse per essere eseguiti;
- Eseguibili in tempo **finito**;
- **Non ambigui**: ogni azione deve essere univocamente interpretabile dal calcolatore;

Algoritmi ordinamento di un vettore

Gli algoritmi di ordinamento ci permettono di riordinare un vettore in base ad una determinata condizione. I principali due sono:

- **BubbleSort**: Si valutano i valori adiacenti e si sposta verso l'alto il valore più alto fino ad avere sull'ultima cella il valore massimo. Dopo la prima esecuzione si valuterà il vettore fino alla cella n-1. Attraverso una variabile flag possiamo capire il p-esimo momento in cui il vettore è già ordinato.
- **SelectionSort**: Viene cercato il valore minimo e viene messo nella prima posizione. Alla seconda iterazione si parte dalla seconda cella e le iterazioni sono n in base alla celle del vettore. Il SelectionSort non può sapere se un vettore è già ordinato.

Algoritmi di ricerca su un vettore

I principali algoritmi di ricerca su un vettore sono:

- Algoritmo di ricerca lineare: viene cercato il valore richiesto fino a quando non lo trova;
- Algoritmo di ricerca binaria: si mette nel centro e verifica se il valore è quello cercato, se non corrisponde si analizza se è minore o maggiore, allora la ricerca riprende nella seconda metà della posizione k+1 alla posizione n-esima, ripartendo sempre nel centro tra queste due posizioni. Algoritmo che funziona solo se il vettore è ordinato.

6. Strutture dati

Cosa sono le liste?

Una lista è un tipo derivato composto da una concatenazione di una o più strutture contenenti uno o più membri e uno o più puntatori. Necessitiamo delle liste quando non sappiamo il numero di elementi a tempi di compilazione, creare un array sarebbe sbagliato perché:

- Spreco di memoria per l'allocazione di una dimensione arbitraria;
- Memoria insufficiente all'interno dell'array;

Il concetto alla base delle liste è: esiste un puntatore di testa che che punta al primo elemento della lista e da questo elemento è possibile arrivare a tutti gli altri, solo l'ultimo punta a NULL. Quando la lista è vuota, la testa punta a NULL. Ogni elemento deve avere un suo spazio in memoria (operatore new) e deve essere deallocato con il distruttore. La sintassi è la seguente:

```
struct elem {
      int inf;
```

```
      elem* pun;
};
elem* p0=NULL;
elem* p= new elem;
p->inf =1;
p->pun=p0;
p0=p;
```

Operazioni sulle liste

Sulle liste possiamo effettuare le seguenti operazioni:

- **Inserimento in testa**: Operazione che non può fallire, viene creato un puntatore a struttura, il cui puntatore è la testa. La testa diventa l'elemento aggiunto;
- **Estrazione in testa**: Operazione che fallisce se la lista è vuota. Un puntatore di appoggio punterà al primo elemento mentre la testa punterà all'elemento successivo. Si elimina il puntatore del primo elemento al successivo e si elimina l'elemento.
- **Inserimento in coda**: Utilizziamo la tecnica dei due puntatori per scorrere la lista, si utilizza un puntatore a NULL per creare un nuovo elemento e controlliamo se la lista è vuota:
    - Se non è vuota, l'ultimo elemento punterà al nuovo, il quale avrà come puntatore NULL;
    - Se vuota, inseriamo in testa;
- **Estrazione dalla coda**: Si scorre la lista con il doppio puntatore finché il puntatore davanti non arriva all'ultimo elemento, a questo punto controlliamo se il primo puntatore sia diverso dalla testa:
    - Se è diverso, il puntatore che precede diventerà l'ultimo e si elimina l'elemento;
    - Se è uguale, si elimina l'elemento e la testa diventa NULL;
- **Estrazione di un dato elemento dalla lista**;
- **Stampa della lista**;

Cos'è una struct ed una union

Una struct è un insieme di informazioni di un determinato oggetto, essa si presenta come un tipo di dato aggregato. Non sono definite le operazioni logiche sulle struct. Gli unici operatori definiti per le struct sono:

- **L’operatore selettore di membro** (.): ci permette di accedere ai campi dato della struct;
- **L’operatore ->**: implica una dereferenziazione ed una selezione di membro, usato quando abbiamo un puntatore a struct;

Possono essere parametri di funzione e valori di ritorno delle funzioni.
Le Union sono simili alle struct a differenza che nelle prime i campi dato condividono le celle di memoria, nelle struct ogni dato ha le proprie celle di memoria. Non possiamo avere due membri attivi contemporaneamente, si può disattivare uno per attivarne un altro ma non averli attivi entrambi.

La sintassi è:

```
union U {
      int a;
      int b;
};

U u;
u.a=1; //a attivo
u.b=2; //b attivo, a inattivo

U u1 {1,2} //illecito
```

7. Generali

Cosa usiamo noi per programmare?

Noi utilizziamo il linguaggio C++ con CLion che è un **IDE** (Integrated Development Environment) ovvero uno strumento software integrato che supporta i programmatori nello sviluppo del codice sorgente di un programma. Sono utili per la segnalazione degli eventuali errori di sintassi durante la scritture del codice, fornendo anche degli utili strumenti per la fase di sviluppo e per il debugging. CLion per funzionare necessita di un compilatore (esempio: MingGW per C++) e come debugger CMAKE.

Funzioni iterative e ricorsive

Le **funzioni ricorsive** hanno la particolarità di poter **invocare**, oltre alle altre funzioni, **loro stesse**. Essa è caratterizzata da un **caso base** che permette di evitare il **loop** ed un **passo ricorsivo**. Il **caso base** ha un **argomento** che viene **modificato**

ad ogni passo della ricorsione fino a raggiungere un determinato valore e stoppare le ricorsione. Così vengono terminate tutte le istanze aperte dalla ricorsione fino al ritorno al chiamate.
Le funzioni iterative non invocano loro stesse nel corpo della funzione, tutte le funzioni possono essere implementate sia in modo iterativo che ricorsivo.

Preprocessore

Il **preprocessore** è un programma che effettua una preliminare elaborazione del testo del programma prima che avvenga l'analisi lessicale e sintattica. Il suo **output** rappresenta l'**input** del **compilatore** e viene utilizzato ogni qualvolta **utilizziamo** il **carattere #**.
Tra le direttive per il preprocessore troviamo:

- **#include**: include altri file come per esempio librerie;

```
#include <libreria> //se presente nella dir default
#include "libreria.h" //se nella stessa dir del .cpp
```

- **#define**: per definire un simbolo oppure un valore associato per esempio per definire una costante. Metodo sconsigliato perché il preprocessore non fa' controlli sul tipo.
- **#ifndef**, **#elif**, **#else**: istruzioni condizionali del preprocessore, ci permette di mandare al preprocessore solo i rami attivi dell'if;

L'uso di **#define** può portare all'uso di vere e proprie azioni che si eseguono durante il programma, le **MACRO**.
Una **MACRO** è un'operazione che viene svolta più volte all'interno di un programma.

```
#define scambia (x, y, temp) (temp) = (x); (x)=(y); (y)=temp;
```

Le **direttive** del **preprocessore** aiutano anche ad **evitare** dei **doppi include**:

```
//compito.h
#ifndef compito_h
#define compito_h
//main.cpp
#include "compito.h"
#include "compito.h" //l'aggiunta di questo non causerà una
ridefinizione della classe
```

Serve prestare attenzione ai DEFINE perché cambiano ogni singola occorrenza dell'identificatore indifferentemente dal tipo che incontra.

Unità di compilazione

É il file sorgente generato dal preprocessore e preso in ingresso dal compilatore per creare il file .object. Il linker si occuperà di collegare i vari file per l'eseguibile, l'importante è che ci sia solo un file main.

Visibilità e scope, using, namespace e risolutore visibilità

Per **campo di visibilità** (scope) in C++ si intende il **limite** oltre il quale una **variabile/funzione** è **visibile**. Lo **scope** definisce anche il **tempo di vita** di una variabile. Una variabile può essere dichiarata più volte nello stesso scope, ma definita una sola volta all'interno di esso (**one-definition-rule**).
Gli **scope** permettono di definire variabili con lo stesso identificatore più volte all'interno del programma ma in scope diversi per evitare le eccezioni date dalla **one-definition-rule**. Esistono diversi tipi di scope:

- Nomi definiti **globalmente**: **durano** per **tutto il programma** come le variabili **static**;
- Nomi definiti dentro un **namespace**: visibili entro i suoi confini;
- Nomi definiti dentro una **funzione**: visibili dentro la funzione;
- Nomi definiti dentro le **classi**: hanno scope entro il loro blocco;
- Nomi definiti dentro **for**, **if**, **while**, **switch**: hanno scope dentro il loro blocco;

Il **risolutore di visibilità** '**::**' è un operatore che serve a riferirsi a nomi di scope esterni ad esso soprattutto quando dobbiamo riferirci a variabili con lo stesso nome ma scope esterni. Quando è preceduto da un nome esso rappresenta il **namespace** a cui appartiene:

```
static int x;
if ( x!=0) {
      int x=1; //questa x ha scope dentro l'if
      ::x=2; //si riferisce alla x static fuori dall'if
}
```

Il **namespace** è una regione dichiarativa che provvede allo scope dei nomi dichiarati al suo interno. Sono usati per organizzare i nomi in gruppi logici, visibili tra loro nel namespace, e **prevenire collisioni** quando il codice include più librerie. I nomi con scope locale non hanno bisogno di namespace. Con la direttiva **using** evitiamo di specificare sempre il namespace. Un namespace può essere dichiarato più volte in un file, ci penserà il preprocessore ad unire tutto, bisogna prestare attenzione all'ordine:

```
namespace V {
      void f ();
}
```

```
void V::f () {} //ok
void V::g () {} //illecito, g () non fa' parte di V
namespace V {
      void g();
}
```

Si può assegnare un nome alternativo ad un namespace.

```
namespace AVLNN=a_very_long_namespace_name; //adesso si chiama
AVLNN
```

Un namespace con un identificativo vuoto ha un collegamento interno. Il codice rimane visibile sono nel file dichiarato.

```
namespace {
      int fun () {};
}
```

I namespace possono essere anche annidati ma il padre non può vedere i nomi del figlio, a meno che non contenga la keyword inline, il contrario è lecito:

```
namespace padre {
      void Foo ();
      namespace figlio {
            int x;
            void Ban () { return Foo (); } //vede il nome del padre
      }
      int Bar () {...}
      int Baz (int i) { return figlio::x; } //se fosse inline
figlio non servirebbe figlio::
}
```

La **using** è una direttiva che permette l'uso dei nomi di un namespace senza specificarlo con il risolutore di visibilità e può essere applicato ad un unico namespace:

```
using namespace std;
```

Cos'è la grammatica di un linguaggio?

É l'insieme è l'insieme di regole che permettono l'uso dello stesso, essa è divisa in sintassi e semantica:

- Sintassi: Spiega le regole con cui vengono scritte le istruzioni;
- Semantica: Indica il loro significato;

Cos'è il polimorfismo?

Il polimorfismo indica la possibilità di definire metodi e proprietà con lo stesso nome. Esso può essere suddiviso in due parti:

- **Polimorfismo programmazione ad oggetti**: Si riferisce al fatto che in una classe derivata possa ridefinire un metodo della classe base con lo stesso nome. Per esempio, prendiamo una funzione, questa deve essere preceduta dalla keyword virtual che significa che è possibile avere un'altra definizione per la stessa funzione all'interno della classe derivata. La funzione, dentro la classe derivata, deve avere la keyword override che implica una sovrascrittura di un metodo che ha lo stesso nome del metodo della classe base;
- **Polimorfismo programmazione generica**: Si riferisce al fatto che è possibile stilare un programma senza un tipo specifico. Questo nel C++ avviene attraverso l'uso di Templates.

Classi di Memorizzazione

Le classi di memorizzazione (**storage class**) è una particolare proprietà che definisce il tempo di vita di un oggetto dentro il programma. Esse sono tre:

- **Automatica**: Oggetti dichiarati all'interno di un blocco. Vengono creati quando viene incontrata la sua dichiarazione e vengono distrutti alla fine del blocco. Se non vengono inizializzati il loro valore è indefinito;
- **Statica**: Sono oggetti dichiarati fuori da una funzione. Vengono creati/inizializzati all'inizio dell'esecuzione del programma e vengono distrutti a fine programma. Se non sono stati inizializzati, il loro valore è 0;
- **Dinamica**: Oggetti creati con l'operatore **new**. Vengono distrutti automaticamente oppure con l'operatore **delete** oppure a fine programma stesso. Se non inizializzati hanno un valore indefinito.

Cos'è e come si gestisce la memoria dinamica?

Nella memoria dinamica/Heap gli indirizzi degli oggetti sono allocati in modo sparso a differenza dello stack che è allocato in modo sequenziale. Utilizziamo la memoria dinamica quando non conosciamo la dimensione di un oggetti a tempo di compilazione ma solo a tempo di esecuzione. Per allocare oggetti in questa memoria bisogna usare la keyword **new** seguita dal tipo. L'operatore new restituisce l'indirizzo del blocco che deve essere salvato da un puntatore creato sullo stack e, se questo puntatore acquisisce un nuovo indirizzo, perdiamo quello dell'oggetto sullo heap ma l'oggetto esisterà finché non verrà cancellato con l'operatore delete oppure a fine programma.

Cosa si intende per programmazione a moduli?

Con la programmazione a moduli segmentiamo il programma in più file:

- Modulo **server**: quello che offre servizi formato da file di dichiarazione nome.h e di implementazione nome.cpp;
- Modulo **client**: quello che sfrutta i servizi del modulo server attraverso la direttiva del preprocessore #include “nome.h”;

Utilizziamo questa organizzazione per:

- **Semplificare** lo sviluppo, il test e la manutenzione del programma;
- Mascherare l’implementazione di una classe o libreria della sua definizione (**information hiding**);
- **Prestazioni migliori** dato che la modifica di un solo modulo di implementazione richiede solo la ricompilazione di quello unico mentre quello di dichiarazione implica la ricompilazione di tutti i moduli che lo includono;

Lettura e scrittura su file

Per leggere/scrivere su file si utilizza la libreria <fstream> che contiene la scrittura e lettura su file. Si crea così un oggetto **fstream** con un proprio identificatore e si utilizzano le funzioni presenti nella libreria per aprire/chiudere un file:

```
fstream ff;
ff.open ("[directory/]nomefile.txt", ios::out)
```

Questo crea un oggetto fstream chiamato ff su cui invoca la funzione open che crea un file chiamato “nomefile.txt” in scrittura (ios::out) nella directory “directory”.
Il secondo parametro della funzione open (enum) può essere:

- ios::out (**sovrascrittura**): se il file non esiste viene creato;
- ios::in (**sola lettura**): se il file non esiste dà errore;
- ios::out | ios::app (**scrittura alla fine del file**): se il file non esiste viene creato;

Possiamo usare anche le classi **ifstream** e **ofstream** invece di **fstream**:

```
ifstream ff ("[directory/]nomefile.txt"); //sovrascrittura
ofstream ff ("[directory/]nomefile.txt"); //sola lettura
ofstream ff ("[directory/]nomefile.txt", ios::app); //scrittura
alla fine del file
```

Successivamente, utilizzando l’identificatore assegnato allo stream di lettura collegato al nostro file ed una variabile dello stesso tipo quelle che dovrà essere letto dal file, possiamo effettuare la lettura o scrittura considerando il nostro identificatore come uno stream cin o cout, in base a che tipo enum abbiamo usato.
È importante fare un check per verificare che il file esista tramite if (!ff), se non esiste dovremmo implementare un messaggio di errore:

```
cerr <<"messaggio"
```

Le operazioni di stream potrebbero dare errore se:

- Se dobbiamo crearlo ma non abbiamo permessi di scrivere nella directory dove si trova il file;
- Se dobbiamo leggerlo ma se non abbiamo permessi per accedervi;
- Se dobbiamo leggerlo ma ci sono caratteri non supportati dalla codifica;
- Se il file ha lo stesso nome di una cartella nella directory in cui si trova;

```
int a=0; nome >> a; //lettura (è come fare cin >> a)
char s='S'; nome >> s; //scrittura (è come fare cout << s)
```

Si può chiudere lo stream tremite ff.close (), alla fine del programma tutti gli stream vengono chiusi automaticamente.

Cosa mi serve per poter usare cin e/o cout?

Per poter usare le operazioni di cin e/o cout serve includere necessariamente la libreria **iostream** ed opzionalmente il **namespaces std**.
#include <iostream>
using namespace std;

```
cout << a;

std::cout << a; //alternativa senza includere il namespace std
```

Senza usare il namespace std bisogna utilizzare il risolutore di di visibilità "::" altrimenti avremmo un errore nel main.cpp.

Concetto e manipolazione di stream

Lo stream è una sequenza di celle illimitate da 1 **byte** che termina con la marca di file. Gli stream sono principali:

- **cin** (see-in);
- **cout** (see-out);
- **cerr** (see-err);

Gli stream di ingresso ed uscita **non** sono standardizzati nel linguaggio ma vengono inclusi dalla libreria **iostream**. **Cin** e **cout** sono rispettivamente **istream** ed **ostream**. Nella sua struttura lo stream presenta un **unsigned int**, tra i vari bit del unsigned ricordiamo:

- **Fail bit** / ios::failbit: se è ad 1, l'errore è **recuperabile**;
- **Bad bit** / ios::badbit: se è ad 1, l'errore è **irrecuperabile**;

- **Good bit**: se è ad 1, lo stream è in uno stato **consistente**;
- **Eof bit** / ios::eofbit: bit di fine file/fine stream.

Se il stream va' in errore, bisogna usare cin.clear () per ripristinare la situazione.

```
int re, im;
cin >> c; //leggo il primo carattere
if (c != '(') cin.clear (ios::failbit); //se non risponde faccio
clear del failbit
else {
      cin >>re >>c; //altrimenti vado avanti mettendo il valore in
re e leggendo c
      if (c != ',') cin.clear (ios::failbit);
      else {...}
}
```

Per stampare a video oppure leggere in ingresso tipi di dato classe, bisogna ridefinirli come funzioni friend (non essendo funzioni membro, non vedono gli attributi privati):

```
friend ostream& operator << (ostream&, const Classe&);
friend istream& operator >> (ostream&, const Classe&);
```

Parametri:

- **riferimento di stream**: dato che dobbiamo prendere lo stream e modificarlo, non crearne uno nuovo e renderlo concatenabile ad altri stream;
- **riferimento costante all'oggetto classe**: costante per non permettere la sua modifica per nessuna ragione e riferimento per evitare la chiamata al **costruttore di copia**;

Come funziona lo scambio di variabili? Come si rappresenta in memoria?

Lo scambio avviene attraverso una variabile di appoggio temporanea temp, questo perché sennò l'inserimento di una variabile in un'altra provoca la perdita di quest'ultima:

```
int a=5;
int b=2;
a=b; //in questo modo metti 2 in a, il valore precedente di a è
perso

int a=5;
int b=2;
int temp=a; //mi salvo il valore di a
```

```
a=b; //ora cambio il valore di a
b=temp; //mi ero salvato il valore precedente di a e lo metto in b
```

Che cosa è la typedef e come si usa?

Typedef è una keyword che permette di utilizzare un identifier per nominare un tipo anche composto. Esso potrà essere utilizzato per richiamare il tipo. Si definisce scrivendo typedef seguito dal tipo che vogliamo assegnare e subito dopo quello che vogliamo definire:

```
typedef elem* lista; //in questo caso possiamo "trasformare" elem* in lista
```

Questo metodo non modifica il tipo ma gli affianca un semplice “sinonimo” con un semplice scopo mnemonico per facilitare la comprensione del codice.

Cos’è l’overloading?

Possiamo vedere l’overloading nel linguaggio C++ in due modi:

1. **Overloading delle funzioni**, sovrapposizione di due funzioni con il medesimo identificatore che differiscono per il **tipo** degli argomenti formali e/o **numero** degli argomenti formali e/o **ordine** del tipo degli argomenti formali. **NB**: In C++, l’overloading non può essere fatto solo sul tipo restituito (non differiscono per il return type). C++ decide anche la priorità con la quale richiamare le funzioni in caso di overloading: prima quelle con minor numero di argomenti, poi in base al tipo. Nel caso di egual numero si dà priorità alla funzioni con **minor numero di conversioni implicite possibili**.

```
int fun (int a, int b) {...}
double fun (double a) {...}
int fun (double a, int b) {...}

fun (2, 7); //viene richiamata la prima
fun (4.5, 2); //viene richiamata la terza, per le zero conversioni implicite
fun (7.8); //viene richiamata la seconda
fun (2); //viene richiamata la seconda, priorità del numero di argomenti sul tipo
```

2. **Overloading degli operatori**, essendo le classi tipi derivati, ci sono operazioni che non possiamo eseguire su di loro, quindi una determinata operazione deve essere ridefinita per la classe potendo così estendere ai tipi derivati proprietà fondamentali.
   NB: Non si può ridefinire il risolutore di visibilità “::”, il selettore di membro “.”

ed il selettore di membro attraverso puntatore ".*".
Gli operatore si possono definire **unario** oppure **binari**.

- **Unari**: Si possono dichiarare funzioni membro della classe, in questo caso non serve metterli **friend** e possiamo usare il puntatore **this** (l'argomento a sinistra dell'operazione) ed il singolo argomento formale classe (l'argomento a destra dell'operazione) per effettuare varie operazioni.

```
classe operator + (const Classe&);
```

- **Binari**: Serve dichiarare la funzione come globale, aggiungere friend dato che le funzioni globali non possono usare **this** e non può accedere ai membri privati senza **friend**. Qui i due parametri formali sono rispettivamente quello a sinistra e quello a destra dell'operatore che siano ridefinendo.

```
friend class operator + (const class&, const class&)
```

Questa regola vale per tutte le funzioni membro della classe, ad eccezione delle funzioni membro che ritornano reference, gli operatori di "x e assegnazione" (operator +=, -=,...) e degli operatori binari (operator +, -,..) con due argomenti formali const.
Nelle soluzione d'esame, non viene incluso nelle funzioni membro void. In tal caso un oggetto classe const non è in grado di richiamarle:

```
operator int () const; //return type dell'operatore int è
implicito
int valuta (int) const;
```

L'operator << include l'argomento formale come const:

```
friend ostream& operator << (ostream&, const Polinomio&);
SongPlaylist& operator += (const SongPlaylist&); //+= non vale per
i const
```

Il const non si mette negli operatori binari perché il primo argomento formale è un left-value:

```
friend Polinomio operator * (const Polinomio&, const Polinomio&);
```

Nel prossimo esempio serve perché il left-value è l'istanza di classe **this**, quindi devo renderla **const**.

```
Polinomio operator * (const Polinomio&) const; //operatore unario
```

Si mette in tutte le funzioni membro (escluso le void nell'esame) che non ritornano reference ad eccezioni degli operatori binari e di "x e assegnazione".

Collegamento interno ed esterno

La differenza risiede nella visibilità tra file di un'unità di compilazione:

- Un identificatore ha collegamento **esterno** se è visibile all'unità di compilazione, di conseguenza, nessun altro oggetto globale tra i file può avere tale nome. I nomi globali non static hanno di default collegamento esterno.
- Un identificatore ha collegamento **interno** se è visibile sono nel file in cui è dichiarato. Un nome con collegamento interno quindi può esistere anche più volte in altri file, tutti con collegamento interno. I nomi non globali, const, typedef hanno di default collegamento interno.

Funzionamento di alcune funzioni/operatori comuni

Regola fondamentale da sapere per le stringhe/vettori: allocare una stringa di dimensioni insufficienti rispetto a quelli necessari o accedere a indici superiori/inferiori alle dimensioni del vettore NON causano eccezioni:
allocare meno caratteri di quelli necessari funziona, solo che solamente i caratteri allocati sono **riservati** alla stringa, gli altri sono scritti in zone successive ad essa, durante l'esecuzione del programma può capitare che quelle zone di memoria successive a quelle allocate possono essere modificate da altre cose che non siano modificare il vettore stesso.

```
char* s1=new char [2];
strcpy (s1, "ciao come stai"); //viene fatta, ma si modificano
celle successive
cout << s1; //stampa "ciao come stai"
int v []= {1,2,3};
v[4] ='4'; //lecito, modifica la cella successiva al vettore
for (int i=0;i<8;i++) cout <<v[i] <<" ";
//output: 1 2 3 4 21723 409443 0, 8, 123901 valori casuali
```

- **sizeof(obj)**: ritorna la dimensione in memoria (in byte) dell'algoritmo specificato.

```
char c [100] = "FdP";
sizeof(c); //ritorna 100 (100* 1 byte)
int v []={1,2,3};
sizeof(v); //ritorna 12 (3*4 byte)
sizeof(c) /sizeof(int); //così ritorna 3, gli elementi del vettore
```

- **<cstring> strlen(s)**: ritorna la lunghezza della stringa passata come argomento contando dall'inizio della stringa fino al **carattere di fine stringa** "/0";

```
strlen(c); //ritorna 3
```

- **<cstring> strcmp (s1,s2)**: controlla i primi due caratteri di s1 e s2, se corrispondono, continua con le coppie successive, finché non trova la differenza tra la coppia, ritornandola come valore oppure finché non trova un carattere di fine stringa. Se non trova nessuna differenza torna zero.

```
char s1 []= "Giuseppe";
char s2 []= "Giusoppe";
//ritorna la differenza tra 'e' ed 'o' in intero
```

T.G.N.