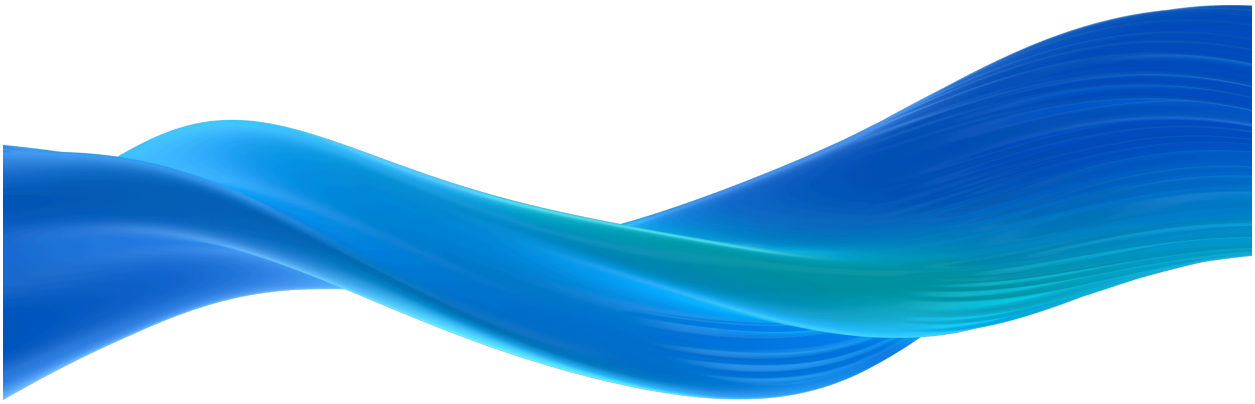


👨‍💻

Argomenti fondamentali

📊 Slide Dump

Slide 1 2023 - Introduzione alla programmazione ad oggetti

- ▼ Ereditarietà

  Una classe può ereditare degli attributi da un'altra classe estendendola

- ▼ Polimorfismo

  Tramite **Sovraccarico (Overloading)** di funzioni e operatori e tramite **Sovrascrittura (Overriding)** di funzioni, possiamo avere "le stesse funzioni" che fanno cose diverse a seconda del tipo/classe

- Differenza tra programmazione ad oggetti e programmazione procedurale

- ▼ Argomenti predefiniti per le funzioni

  È possibile fornire valori predefiniti per i parametri della funzione, ad esempio:

  ```
  void three_ints(int a, int b=4, int c=3); // Dove b,c hanno dei valori predefiniti
  ```

- ▼ I/O Stream

```
//Esempio di input e output
#include <iostream>

int main(){
   int num;

   std::cin >> num;
   std::cout << "Il numero inserito è:" << num //no flush needed
}
//L'operazione di input sovraccarica l'operatore di shift a destra ">>"
//L'operazione di input sovraccarica l'operatore di shift a sinistra "<<"
```

▼ Sovraccarico dell'operatore

in C++ è possibile definire funzioni e operatori con nomi identici ma che eseguono azioni diverse.
Le funzioni devono differire nella loro lista di parametri (e/o nel loro attributo const).

Esempio sovraccarico operatori `<<` e `>>`:

```
#include <iostream>

class Punto {
private:
   int x, y;

public:
   Punto(int x = 0, int y = 0) : x(x), y(y) {}

   // Dichiarazione delle funzioni amiche per sovraccaricare gli operatori
   friend std::ostream& operator<<(std::ostream& out, const Punto& p);
   friend std::istream& operator>>(std::istream& in, Punto& p);
};

// Definizione della funzione per l'operatore <<
```

```
std::ostream& operator<<(std::ostream& out, const Punto& p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}

// Definizione della funzione per l'operatore >>
std::istream& operator>>(std::istream& in, Punto& p) {
    in >> p.x >> p.y;
    return in;
}

//*********************************
#include <iostream>
#include "Punto.h" // Assumendo che la classe Punto sia definita in Punto.h

int main() {
    Punto p1(3, 4);
    Punto p2;

    std::cout << "Punto p1: " << p1 << std::endl;

    std::cout << "Inserisci le coordinate di p2 (formato: x y): ";
    std::cin >> p2;

    std::cout << "Punto p2: " << p2 << std::endl;

    return 0;
}
```

▼ Stringhe (Classe)

```
std::string //classe standard per le stringhe in c++
```

▼ Namespace

Un meccanismo per la suddivisione dei nomi (di tipi, variabili, funzioni, ecc.)

- Permette di costruire programmi di grandi dimensioni a partire da vari componenti senza il rischio di scontri di nomi
- Uno spazio dei nomi (namespace) è un meccanismo utilizzato per mantenere i nomi di funzioni, variabili e tipi in raggruppamenti separati, in modo che i nomi di gruppi diversi possano essere uguali senza scontrarsi tra loro

La libreria standard del C++ ha nomi in un namespace chiamato std

▼ Nomi qualificati

Un nome può essere qualificato, utilizzando l'operatore “::” detto risolutore di scopo/contesto (scope resolution), con lo spazio dei nomi in cui è stato dichiarato, ad esempio:

```
std::cout
```

▼ Direttiva “Using”

La direttiva “using” viene utilizzata per includere un intero namespace all’interno del codice, senza dover specificare per ogni istruzione da quale namespace proviene:

```
//Riprendo lo stesso esempio utilizzato per I/O Stream
using namespace std

int main(){
    int num;

    cin >> num;
    cout << "Il numero inserito è:" << num //no flush needed
}
//Notare come davanti a "cin" e "cout" non è stato utilizzato il risolutore disc
```

▼ Passaggio per riferimento (by reference) e puntatori (pointers)

Il simbolo * viene utilizzato per dichiarare un puntatore, e il simbolo & viene utilizzato per ottenere l'indirizzo di una variabile, creando così un riferimento che punta al valore memorizzato da un puntatore.

- Se in C vogliamo modificare il valore di una variabile all'interno di una funzione dobbiamo passare quella variabile tramite puntatore, simulando cosi quello che in C++ è il cosiddetto passaggio per riferimento "Pass by Reference"
- Un riferimento è sostanzialmente un'alias e non una copia, nel caso degli argomenti di una funzione, viene passato il riferimento all'argomento vero e proprio e non una copia
- Un riferimento è indicato col carattere "&" dopo il tipo dell'argomento
- I riferimenti possono essere anche usati per restituire il risultato di metodi/funzioni, ad esempio:

```
T& doSomething(T& value);
T& T::doSomethingElse(T& otherValue);
```

- Definire una variabile come riferimento è simile all'utilizzo dei puntatori in C, con la differenza principale che con i riferimenti non si deve dereferenziare la variabile per ottenere il valore (come con i puntatori) {Dereferenziare un puntatore significa quando in C, per accedere al valore che esso punta e non alla cella di memoria utilizziamo ad esempio `(*pointer)` {In C++ si può anche scrivere come: `*pointer`}, mentre invece se scriviamo solo `pointer` facciamo riferimento alla cella di memoria, ad esempio `0x1001`}
- Un riferimento va sempre inizializzato
- Non è possibile ridefinire un riferimento
- Un riferimento non può fare aritmetica
- Un riferimento non può essere assegnato
- Un riferimento non può avere riferimenti a void

- Un riferimento può essere specificato come const, la funzione/il metodo non può modificare il contenuto della
  variabile
- Utile per passare strutture di dati di grandi dimensioni che non devono essere modificate come riferimenti const
- Se non c'è bisogno di mutare un parametro in ingresso, usare un riferimento const, a meno che non si sappia che la copia è più conveniente (in caso di dubbio, usare un riferimento)
- I riferimenti servono per il passaggio dei parametri (e dei valori di ritorno); a volte possono essere utili come variabili locali (ma è raro)
- I riferimenti sono molto meno potenti dei puntatori, tuttavia, sono molto più sicuri dei puntatori: il programmatore non può accidentalmente usare male il riferimento, cosa che può avvenire con i puntatori
- Preferire T* (Puntatore) a T&(Riferimento) quando "nessun argomento" è un'opzione valida, un puntatore (T*) può essere nullo, mentre un riferimento (T&) non può esserlo; non esiste un "riferimento nullo" valido.
- Non restituire mai (direttamente o indirettamente) un puntatore o un riferimento a un oggetto locale, come per esempio:

```
int& f() { intx=7;
// ...
return x; // scorretto: restituisce un riferimento a un oggetto che sta per e

int* f() {
intfx=9;
return &fx; // male
```

In poche parole possiamo dire che:

```
void add(int x, int y, int* sum) {
   (*sum) += x+y;
}
//La funzione sovrastante in C coincide con la seguente funzione in C++
```

```
void add(int x, int y, int& sum) {
   sum += x+y;
}
```

Ad esempio:

```
int x =1;
int& y= x; // riferimento
y= 2; // anche x viene modificato
cout << x << endl;

//int& z; // errore, non si compila, perché?
int* z; // puntatore
z= &x; // & a sinistra è diverso da & a destra di = *z= 3; // x è modificato
cout << x << endl;
```

▼ Nullptr

Lo standard C++11 fornisce una nuova parola chiave speciale: `nullptr` per impostare e determinare se un puntatore punta a niente, ad esempio:

```
int *pa = &a;
pa = nullptr;
if (pa != nullptr) { }
```

- Linee guida sullo stile di scrittura

▼ Heap & Stack

Esistono due modalità di allocazione, statica (stack) e dinamica (heap).

Slide 2_Parte_I 2023 - Classi e Oggetti

▼ Astrazione e tipo di dati astratto

Un ADT (Abstract Data Type) è una specifica di un insieme di dati e dell'insieme di operazioni (l'interfaccia dell'ADT) che possono essere eseguite sui dati, è astratto poiché non dipende dall’effettiva implementazione dei dati.

Ad esempio una classe astratta da un’idea di come dovrà poi essere implementata/scritta nel codice vero e proprio.

▼ Incapsulamento

Nei programmi Object Oriented, come abbiamo già osservato, si è soliti mettere in stretta relazione tra loro un pezzo di informazione con il comportamento specifico che agisce su tale informazione. Questo è ciò che abbiamo definito oggetto.

L'incapsulamento è proprio legato al concetto di "impacchettare" in un oggetto i dati e le azioni che sono riconducibili ad un singolo componente.

▼ Classi e oggetti

Una classe è una rappresentazione effettiva di un ADT, fornisce i dettagli di implementazione per la struttura dati utilizzata e le operazioni.

Gli oggetti vengono istanziati dalle classi, cioè sono variabili di un determinato tipo di classe

Importante distinzione tra una classe e un oggetto, una classe è una rappresentazione astratta di un insieme di
oggetti che si comportano in modo identico.

▼ Principio aperto-chiuso

Se una classe ha un particolare comportamento, codificato nel modo in cui vogliamo, se nessuno può cambiarlo, abbiamo la chiusura per le modifiche, ma se, per qualche motivo, dobbiamo estendere quel comportamento, dobbiamo permettere di estendere la classe per sovrascrivere dei metodi e fornire nuove funzionalità, in questo caso la classe sarà aperta all'estensione.

Vedremo come l'ereditarietà e la composizione ci aiuteranno a seguire questo principio.

▼ Riferimento a se stessi (this)

Tramite la parola chiave `this` possiamo fare riferimento ad un attributo della classe, cosi facendo, quando definiamo una funzione della classe possiamo avere nomi uguali, ma variabili diverse, come ad esempio(il seguente esempio è in Java):

```
public class Book {

    private String name; // *
    private Author[] authors;
    private Double price;
    private Integer qty = 0;

    public Book(String name, Author[] author, Double price) {
        this.name = name; //Quando utilizziamo this, facciamo riferimento a que
        this.authors = author;
        this.price = price;
    }
}
```

In C++ quando vogliamo scrivere la parola chiave this, utilizziamo la seguente sintassi: `this->name`

▼ Livelli di accesso (public private protected) (Regola dei massimi)

Tutti i membri di una struct sono visibili non appena c'è un riferimento alla struttura, mentre in una classe è possibile differenziare l'accesso come pubblico (`public`), privato (`private`) e protetto (`protected`), l'accesso predefinito della classe è privato.

In questo modo possiamo progettare meglio l'interfaccia della classe, cioè decidere cosa è nascosto e cosa è visibile nella classe (incapsulamento).

- public: un membro pubblico è visibile a chiunque abbia un indirizzo o un riferimento all'oggetto.
- private: un membro privato è visibile solo ai metodi della classe in cui è definito
- protected: un membro protetto è visibile solo ai metodi della classe in cui è definito e nelle classi derivate (passaggio per ereditarietà)

Specificare sempre il tipo di accesso.
Evitare di usare dati membro (attributi) pubblici, usare invece metodi pubblici

per accedere o modificare (get/set) i loro valori, la maggior parte degli IDE possono creare questi metodi automaticamente.

▼ Implementazione dei metodi

I metodi (Le funzioni) di una classe sono di solito definiti (implementati) in file .cpp separato dalla definizione della classe (file intestazione .h).

Bisogna aggiungere il nome della classe di fronte, ad esempio:

```
bool Stack::push(data value) {
    // codice che implementa il metodo
    //....
}
```

E' possibile anche implementarli (inline) nei file intestazione, ma di solito si fa solo se sono molto corti (ad esempio ~5-7 linee).

▼ Attributi

- I metodi possono sempre accedere agli attributi (Le variabili) della classe (sono sempre visibili)
- Gli attributi mantengono lo "stato" dell'oggetto
- Sono una sorta di contesto condiviso tra i metodi, questo riduce la complessità dei metodi, rispetto ad una implementazione classica procedurale (come in C)

In C++ quando abbiamo un puntatore di tipo Struct o una Classe, quando dobbiamo accedere ad un suo attributo o ad un suo metodo non scriviamo `(*classe).attr`, ma scriviamo `classe->var`

▼ Sovraccarico (Overloading)

Possiamo definire più di un metodo con lo stesso nome e tipo restituito, ma con parametri (numero e tipi) diversi (firma), il metodo si dice sovraccaricato.

E' utilizzato di solito per fornire versioni alternative di funzioni da utilizzare in situazioni diverse, la stessa operazione astratta può avere diverse

implementazioni concrete.

Non possiamo semplicemente cambiare il tipo del valore restituito, il compilatore dovrebbe sempre controllare il tipo della variabile in cui abbiamo inserito il valore... e cosa succede se lo scartiamo (cioè se non usiamo il valore restituito)?

il compilatore creerà un segmento di codice e un simbolo diverso (attraverso il name mangling), ottenuto estendendo il nome del metodo con suffissi relativi ai tipi di parametri.

▼ Sovraccarico degli operatori

E' possibile anche sovraccaricare gli operatori, non solo i metodi, ad esempio + è un operatore per interi, reali, etc.

E' opportuno sovraccaricare gli operatori SOLO quando ha davvero senso, ad esempio, sovraccaricare == per confrontare le stringhe, non sovraccaricare * per le stringhe.

- Gli operatori vengono sovraccaricati in modo che gli oggetti si comportino come tipi primitivi
- Non si possono creare nuovi operatori, ma solo modificare la funzionalità degli operatori esistenti sugli oggetti
- Se si sovraccarica + NON aspettarsi che += venga dato automaticamente !Definire anche questo operatore !
- Spesso gli operatori sono solo "amici"(friend)
- L'operatore sovraccaricato può essere definito come membro della classe (quindi dichiarato dentro la definizione della classe), oppure esternamente, quindi funzione globale, ovvero non-membro della classe, in questo caso per operare sugli attributi della classe viene dichiarato come "friend" (vedere più avanti).
- operatori unari, se dichiarati come funzioni membro non hanno argomenti, se dichiarate come funzioni globali, hanno un argomento.
- operatori binari, se dichiarati come funzioni membro hanno un argomento, se dichiarate come funzioni globali, hanno due argomenti

Esempio di sovraccarico di un operatore:

```
//Array.h
classe Array {
   public:
      Array(int size); // costruttore
bool operator ==(const Array& right) const; // il metodo non può modificare r
private: int size;
int* data; // puntatore al primo elemento dell'array
};

//Array.cpp
bool Array::operator==(const Array& right) const {
// inizia a controllare la dimensione degli array
if ( size != right.size) return false;
// poi controlla l'intero contenuto degli array
for ( int i=0; i < size; i++ ) {
if ( data[i] != right.data[i] ) return false;
}
return true; // sia la dimensione che il contenuto sono uguali }
```

▼ Controllo dei tipi

Il C++ ha un controllo di tipo più severo del C: a seconda del cast dei parametri si determina il metodo che viene eseguito

---

Slide 2_Parte_II 2023 - Classi e Oggetti

▼ Operatori ridefinibili

Operatori binari (con due operandi) ridefinibili:

- Aritmetici + - / * %
- Binari << >> & ^ |
- Logici && ||
- Assegnamento = *= /= %= += -= <<= >>= &= ^= |=

- *Relazione == < > <= >= !=*
- *Selezione membro → →*
- Virgola ,

Operatori unari ridefinibili:

- Logici !
- Deferenziazione * &
- Binari ~
- Aritmetici ++ -- -

▼ Sovraccarico di operatori come membro o come non-membro

Gli operatori binari = , [ ] , →, nonché l'operatore n-ario () (chiamata di funzione), devono essere implementati come funzioni membro, la sintassi dell'operatore lo richiede.

Gli altri operatori possono essere implementati come membri o come non membri, ma può essere preferibile:

- Se operatore unario, implementarlo come funzione membro.
- Se un operatore binario tratta entrambi gli operandi allo stesso modo (li lascia invariati), implementarlo come funzione non membro.
- Se un operatore binario non tratta allo stesso modo entrambi i suoi operandi (di solito cambierà il suo operando sinistro), implementarlo come funzione membro, soprattutto se deve accedere alle parti private dell'operando.
- Se binario e l'operando di sinistra non può essere modificato dall'utente, implementarlo come funzione esterna (non membro), ad esempio gli operatori di input e output << e >> su stream.

▼ Creazione dinamica di oggetti

- Una volta definita una classe, possiamo creare le istanze (oggetti) da essa.
- La creazione può essere statica (sullo stack) o dinamica (sull'heap)

- Una volta compilato e messo in esecuzione gli oggetti con i loro attributi saranno nello stack o nell'heap, il codice dei metodi si troverà in memoria nel segmento del codice ("text" segment) e ogni oggetto avrà gli indirizzi delle funzioni membro.

La creazione dinamica di oggetti è simile all'uso delle funzioni malloc/free (C), ma sintatticamente semplificato, con l'uso di new e delete (C++):

- L'operatore `new` restituisce un puntatore all'oggetto creato.
- All'operatore `delete` si deve passare un puntatore dell'oggetto che si vuole rimuovere.
- L'operatore new calcola automaticamente la dimensione della memoria da allocare, come in esempio:

```
int* p= new int[5]; // ottiene spazio per 5 int
delete[] p; // se new ha [ ] allora anche delete deve avere [ ]
```

- Nel caso in cui avessimo una classe definita tramite `new` , essendo un puntatore avremo due modi per accedere ai relativi membri:

```
int main(){
    Student *studente1 = new Student(...);

    //Essendo puntatori possiamo utilizzare la dereferenziazione del C
    (*studente1).attributo;
    (*studente1).metodo();

    //Oppure utilizzare l'operatore -> del C++
    studente1->attributo;
    studente1->metodo();

    //Ovviamente nel caso di un array, proprio come in C, la dereferenziazi
}
```

▼ Costruttori

- Il costruttore è una funzione membro che verrà invocata quando viene creato un oggetto di quella classe, non restituisce alcun valore e ha sempre lo stesso nome della classe.
- È comune sovraccaricare una funzione costruttore (cioè fornirne diverse versioni), in modo che l'oggetto possa essere creato in vari modi diversi.
- Se non viene definito alcun costruttore, il compilatore genera un costruttore "predefinito" (di default) senza parametri.
- Il costruttore di default viene invocato senza parentesi: `Class myclass;`
- Oppure se tramite new, si possono usare entrambi: `classPointer = new Class;` `classPointer = new Class();`
- E' invece un errore: `Class myclass();` questa dichiara una funzione.
- Va bene invece: `Class myclass{}`
- Se una classe ha dei costruttori ma non un costruttore di default, la sua creazione sarà vincolata ai casi dove si applicano i costruttori definiti, è corretto non fornire un costruttore di default se vengono definiti altri costruttori.
- I costruttori sono tipicamente pubblici (ma non necessariamente).
- Se non si vuole che una classe venga istanziata, si può dichiarare un costruttore come protetto, in questo caso possiamo istanziare solo classi derivate (se il loro costruttore è pubblico).
- In altri casi, possiamo dichiarare un costruttore come privato, in genere il suo uso è legato ai metodi statico, oppure come costruttore delegato (vedi seguito)

▼ Lista di inizializzazione

- Poiché compito principale del costruttore è quello di inizializzare l'oggetto, di solito dovrebbe inizializzare i suoi attributi.
- Esiste un modo compatto e leggero per inizializzare gli attributi in un costruttore, ovvero la lista di inizializzazione.

- Usare gli inizializzatori del costruttore (lista di inizializzazione) è come eseguire la definizione e l'inizializzazione allo stesso tempo, se non si inizializza esplicitamente un membro nell'elenco degli inizializzatori del costruttore, quel membro viene inizializzato per default prima che il corpo del costruttore venga eseguito, gli attributi costanti devono essere inizializzati esplicitamente.
- Linee guida: utilizzare se possibile la lista di inizializzazione

Esempio:

```
class Stack {
   public:
      Stack(int s) : top(0), max(s), buf(new Data[s]) {...}; //Utilizzare lo stesso
   private:
     int top, max;
      Data* buf;
};
```

▼ Inizializzatori predefiniti

E' anche possibile inizializzare gli attributi quando li si dichiara nella definizione della classe rendono i costruttori meno prolissi (riducendo la loro lista di inizializzazione):

```
class X {
  public:
      X() {} // i==4, j==5
   private:
   int i= 4, j=5;
};
```

Questi inizializzatori vengono utilizzati in qualsiasi costruttore, a meno che non si inizializzino esplicitamente nella lista di inizializzazione del costruttore, come ad esempio:

```
class X {
   public:
      X() {} // i==4, j==5
      X(int ni): i(ni) {...} // j==5
   private:
   int i= 4, j= 5;
};
```

▼ Inizializzazione uniforme

- Sintassi introdotta con il C++11 per uniformare l'inizializzazione di oggetti: `type var_name{arg1, arg2, .... arg n}`
- L'inizializzatore {} (inizializzazione di una lista) usato per tutte le inizializzazioni risolve alcuni problemi di parsing e permette di avere una sintassi uniforme e permette di evitare facili errori.
- Esempio:

```
//Senza inizializzazione uniforme
int a = 0;
int b; // la varabile non viene inizializzata
int v[]= {1, 2, 3};
string str("hello");
T t;

//Il codice qui sopra può essere riscritto tramite inizializzazione uniforme:
int a{0};
int b{}; // la varabile viene inizializzata a 0 int v[]= {1, 2, 3};
string str{"hello"};
T t{};
```

- Quindi ora sappiamo che per definire un oggetto possiamo utilizzare i seguenti “metodi”:

```
X x1 = X{1,2}; // X(1,2)
X x2 = {1,2};
```

```
X x3{1,2}; // x3(1,2)
X* p = new X{1,2}; // new X(1,2)
```

- Da notare che `{}` non permette di fare conversione automatica restrittiva:

```
long double ld = 3.1415926536;
int c(ld), d= ld; // ok: but value will be truncated
int a{ld}, b = {ld}; // error: narrowing conversion required
```

- Ovunque potreste usare le parentesi `()` nella costruzione di un oggetto, si può preferire usare le parentesi graffe `{}`

▼ Costruttori con delega

- Un costruttore con delega utilizza un altro costruttore della sua stessa classe per eseguire la sua inizializzazione.
- È utile se c'è molto codice duplicato nei costruttori, in questo caso spostarlo in un costruttore comune.
- Prima del C++11, veniva generalmente utilizzata una funzione privata per eseguire compiti comuni, richiamati da diversi costruttori.

Esempio:

```
class SaleData {
   public:
      // non delegating constructor, uses init list
      SaleData(std::string s, unsigned cnt, double price): bookNo(s), unitsSold

      // remaining constructors all delegate to another constructor
      SaleData(): SaleData("", 0, 0) {}
      SaleData(std::string s): SaleData(s, 0,0) {}
      SaleData(std::istream &is): SaleData() {}
      // other members
      // ...
};
```

▼ Costruttori espliciti

- In C++, il compilatore è autorizzato a effettuare una conversione implicita per risolvere i parametri di una funzione.
- Se necessario, i costruttori che hanno un solo parametro eseguono automaticamente una conversione implicita del tipo, ad esempio: se si passa un int quando il costruttore si aspetta un parametro puntatore a stringa, il compilatore aggiungerà il codice che deve avere per convertire l'int in un puntatore a stringa.
- È possibile aggiungere `explicit` alla dichiarazione del costruttore per evitare queste conversioni implicite.
- Dichiarare esplicito (explicit) un costruttore che ha argomenti multipli non ha alcun effetto, perché tali costruttori non possono partecipare alle conversioni implicite, avrà invece un effetto se un costruttore ha più argomenti e tutti, tranne uno, hanno un valore predefinito.

▼ Distruttori

- È un metodo con il nome della classe preceduto da ~, ad esempio: `~GenericClass();`
- Non richiede parametri e non ha valori di ritorno, pertanto non può essere sovraccaricato.
- Richiamato automaticamente quando un oggetto viene distrutto, dovrebbe eseguire delle operazioni di pulizia.

Esempio:

```
//IntStack.h
class IntStack {
    public:
      IntStack(int max);
      ~IntStack();
      //..
    private:
      int len;
      int max;
```

```
    int* buffer;
};

//IntStack.cpp

// Il costruttore alloca la memoria
IntStack::IntStack(int s) { len= 0;
max= s;
 buffer= new int[max];
}

// Il distruttore deve rilasciare la memoria
IntStack::~IntStack() { delete[] buffer;
}
```

▼ Membri statici

- Nel caso di un attributo statico, quell'attributo/variabile verrà condiviso con tutti gli oggetti di quella classe, non è esclusivo ad un solo oggetto di quella classe, ma non possiamo accedere a quel determinato attributo al di fuori della classe, come ad esempio:

```
class Quadrato {
private:
   int lato;
   static quadratiTot; //Questo attributo indica il numero totale di variabili

public:

   Quadrato(int lato){
      this→lato = lato;
      quadratiTot++:; //Ogni volta che viene istanziato un oggetto di tipo Q
   }

   int area(){
      return lato*lato;
```

```
    }
};
```

- Nel caso di un metodo statico, dobbiamo suddividerlo in due utilizzi:
  1. Nel caso in cui volessimo accedere ad un attributo statico, allora anche la funzione che utilizziamo per accedervi dovrà essere statica.
  2. Nel caso in cui volessimo accedere a quella determinata funzione senza però istanziare un oggetto di quella classe.

  Come ad esempio:

```
//Questo esempio estende il precedente
class Quadrato {
private:
    int lato;
    static quadratiTot; //Questo attributo indica il numero totale di variabili

public:

    Quadrato(int lato){
        this→lato = lato;
        quadratiTot++:; //Ogni volta che viene istanziato un oggetto di tipo Q
    }

    int area(){
        return lato*lato;
    }

    static int getTotal(){ //Funzione statica oer ottenere il numero di quadra
        return quadratiTot;
    }
};

int main(){
    Quadrato quad1(3);
```

```
    Quadrato quad1(7);

    cout << "Quadrati totali: " << Quadrato::getTotal() << endl;
    //Come si può notare, abbiamo chiamato la funzione getTotal() senza p
}
```

- Una funzione statica non può accedere a membri non statici della classe.

▼ Friend

- Una classe può consentire l'accesso ai suoi membri (anche se privati) a funzioni (o classi) esterne dichiarando le funzioni (o classi) come amici (`friend`).
- `friend` dovrebbe essere usato solo in situazioni molto particolari, ad esempio per sovraccarichi di operatori di I/O, dove non è desiderabile fornire funzioni membro accessorie, questo perché ostacola l'incapsulamento.
- Una funzione friend, non fa parte della classe, viene solo dichiarata come "amica" all'interno della classe, ma non viene definita all'interno di essa, come nel seguente esempio:

```
class MyClass {
    friend void double_X(MyClass &x);
    //Qua stiamo dicendo alla classe quale sia la funzione friend, ma come

private:
    int x;

    void add(int n){
        x += n;
    }

public:
    MyClass(x) : x(x) {}; //Lista di inizializzazione

    void print(){
```

```
        cout << "x: " << x << endl;
    }
};

void double_x(MyClass &x){ //Notare come non abbiamo riutilizzato la pa
    object.x *= 2
}

int main(){
    MyClass myobject{7};

    myobject.print();

    double_x(myobject); //La funzione può accedere ai membri privati e pr
}
```

- Se volessimo utilizzarla per sovraccaricare un operatore:

```
Class Counter {
    friend Counter operator+(Counter c1, Counter c2);

private:
    int count;
public:
    Counter(count) : count(count) {};

    void print(){
        cour << count << endl;
    }

    void increment(){
        count++;
    }
};
```

```
Counter operator+(Counter c1, Counter c2){
    //Essendo una funzione friend può accedere ai membri privati e protett
    Counter new_counter{c1.count + c2.count};
    return new_counter;
}
```

▼ Classi annidate

- Una classe interna o classe annidata (`inner class`) è una classe dichiarata interamente all'interno del corpo di un'altra classe o interfaccia.
- Le classi annidate in C++ hanno la visibilità (`scope`) delle classi che le racchiudono.
- Le classi annidate sono trattate come amiche (friend) delle classi racchiuse e, usando puntatori, riferimenti e nomi di oggetti espliciti, possono accedere a tutti i membri.
- Le classi interne sono utili per nascondere i dettagli dell'implementazione, esempio:

```
class List {
public:
    List(): head(nullptr), tail(nullptr) {};

private:
    class Node { // not exposed to users of List class
    public:
      int data;
    Node* next;
    Node* prev;
    };

    Node* head;
  Node* tail;
};
```

- E' possibile consentire a una classe esterna di accedere ai membri della classe annidata utilizzando friend:

```
class Algorithm {
public:

   class AlgorithmResults {
      friend class Algorithm; // La classe Algorithm può accedere ai memb
   public:
     void readAlgorithmResult();
  private:
     void writeAlgorithmResult();
  };

   void calculate(AlgorithmResults& results, Arguments...) { //calculate stu
      results.writeAlgorithmResult();
   }
};
```

▼ Costanza / Metodi e Oggetti costanti

- In alcuni casi si vogliono definire delle variabili il cui valore non vogliamo che cambi.
- E' possibile rendere una variabile non modificabile tramite il qualificatore `const`
- Per variabili globali è preferibile rispetto ad usare `#define`
- Vantaggi rispetto a `#define`:
  1. Permette di mantenere il controllo sui tipi.
  2. Permette di racchiudere la variabile all'interno di un ambito di nomi (namespace), evitando conflitti.
  3. Rimane visibile in caso di debugger (non viene rimosso il nome dal compilatore).
- Possiamo usare `const` per:

1. Variabili globali o variabili all'interno di uno spazio di nomi
2. Variabili all'interno di funzioni o blocchi - parametri di funzioni
3. vValore di ritorno di funzioni
4. Variabili membro (attributi) di una classe (statiche e non)
5. Funzioni membro (metodi) di una classe (statiche e non)

- Nel caso di variabili globali const, queste sono visibili solo all'interno del file, per usarle esternamente bisogna dichiararla col qualificatore `extern`

▼ Costanti di classe

E' possibile utilizzare const insieme al qualificatore `static` per definire una costante all'interno di una classe, come ad esempio:

```
class VideoFrame {
private:
    static const int PALFrameRate; //Attributo costante e comune a tutti gli ogg
    ...
};

const int VideoFrame::PALFrameRate= 25;
```

▼ Puntatori e riferimenti costanti

- Come qualsiasi altro tipo, possiamo dichiarare sia puntatori che riferimenti const, come ad esempio:

```
char greeting[] = "Hello";
const char* p = greeting; // pointer to const data
const char& c= greeting[0]; // refererence to const

cout << p << endl;
cout << c << endl;
// p[0]= 'h'; // errore
p= greeting+3; // ok
cout << p << endl;
```

```
greeting[0]= 'h'; // ok
cout << greeting << endl;
//c= 'H' // errore
```

- Possiamo specificare anche che un puntatore è costante:

```
char* const cp = greeting;
//cp= greeting+3; // errore
```

▼ Funzioni e costanza: parametri

- L'uso più potente di const è la sua applicazione alle dichiarazioni di funzione, possiamo fare riferimento al valore di ritorno della funzione, ai parametri della funzione e (per le funzioni membro) alla funzione stessa.
- Aiuta a ridurre gli errori, ad esempio quando si passa un oggetto come parametro usando un riferimento/puntatore e non si vuole che venga modificato (usare quando possibile parametri const):

```
void func(const bar& b);
//b non può essere modificato
```

▼ Funzioni e costanza: valore di ritorno const

- L'uso di un valore di ritorno const riduce gli errori nel codice client, ad esempio:

```
class Rational {
   //...
};
const Rational operator*(const Rational& lhs, const Rational& rhs);

int main(){
   Rational a,b,c;
}
```

```
//ad esempio se ci è sfuggito un = per fare un confronto:
(a*b)=c; //ora è illegale grazie al valore di ritorno const
```

- Quando si restituisce un riferimento, a volte è meglio restituirlo come costante se non si vuole che venga utilizzato per modificare l'oggetto a cui si fa riferimento, ad esempio:

```
class Person {
public:
   string& badGetName() {
      return name;
   }
 //...
private:
   string name;
};

void myCode(Person& p) {
   p.badGetName()= “Igor”; // can change the name attribute of Person
}

const string& badGetName() {...} //MEGLIO!!
```

▼ Funzioni membro const

- Una funzione membro (metodo) const è un metodo che può essere invocato solo su oggetti const.
- Queste funzioni possono utilizzare ma non mutare l'oggetto (gli attributi dell'oggetto).
- è possibile sovraccaricare i metodi differenziando anche solo in const, è utile se si vuole di un metodo con lo stesso nome per accedere o mutare l'oggetto.
- Questo è utile quando si ha a che fare con oggetti passati come riferimenti const.

▼ Funzioni membro const e attributi mutabili

Si può usare `mutable` per indicare che un particolare attributo di una classe può essere modificato anche se un particolare oggetto o metodo della classe è `const` , come ad esempio:

```
class TextBlock {
public:

   size_t length() const {
      if(!isValidLength) {
         length = text.size();
         isValidLength=true;
      }
      return length;
   }

private:
   string text;
   mutable size_t length;
   mutable bool isValidLength;
};
```

▼ Metodi di copia

▼ Copia superficiale

- I metodi impliciti copiano il contenuto degli attributi della classe, cioè eseguono una copia bit per bit del loro contenuto (copia superficiale).
- Se l'oggetto copia superficiale viene distrutto, anche l'oggetto originale perde le proprie risorse.

▼ Copia profonda

- Se un attributo della classe è un puntatore, ad esempio a un array, la copia bit per bit non è sufficiente, poiché si ottiene la copia dell'indirizzo e non degli oggetti che sono puntati.

- Occorre una copia profonda che copi tutti gli oggetti puntati, altrimenti si rischia che la modifica di un oggetto copiato modifichi l'originale e la distruzione di un oggetto copiato distrugga l'originale.
- Un oggetto copiato in profondità non causa alcun problema alla sorgente: ha la propria copia delle risorse

▼ "Regola del tre”

- Quando la nostra classe gestisce una risorsa, cioè quando un oggetto della classe è responsabile di quella risorsa, allora dobbiamo dichiarare metodi espliciti per copiare e creare oggetti da altri oggetti, in genere la risorsa viene acquisita nel costruttore (o passata nel costruttore) e rilasciata nel distruttore.
- Implementare
  1. Il costruttore di copia
  2. Il distruttore
  3. L'operazione di assegnazione

▼ Come creare metodi di copia

- Sia il costruttore di copia che l'operatore di assegnazione ricevono un riferimento all'oggetto originale (sorgente) (in realtà un riferimento const), i metodi condividono molto codice, è possibile racchiuderlo in un metodo privato "di copia" comune.
- L'operatore di assegnazione deve gestire le risorse esistenti dell'oggetto di destinazione e restituire un riferimento per consentire assegnazioni multiple.
- Il distruttore deve rilasciare la risorsa.

▼ Disabilitare la copia/clonazione

Se non si vuole consentire la copia di un oggetto, disabilitare il costruttore copy e gli operatori di assegnazione con la sintassi `=delete` , come nel seguente sempio:

```
class Foo {
public:
    Foo& operator=(const Foo&) = delete;
    // "deactivate" use of assignment operator

  Foo(const Foo&) = delete;
    // "deactivate" copy construction
};
```

▼ Stile Camel case

- I nomi che rappresentano i tipi (cioè le classi) e gli spazi dei nomi devono essere in lettere miste che iniziano con le maiuscole.
- I nomi delle variabili devono essere in lettere miste che iniziano con le minuscole

```
MyClass
myVariable
myFunction()
```

▼ Stile Snake case

- Utilizzo dei trattini bassi "snake case”

```
My_class
my_variable
my_function()
```

▼ Direttive di processamento

- Non usare macro se non per il controllo del sorgente, usando `#ifdef` e `#endif`
    - Le macro non rispettano le regole di tipo e di ambito e rendono il codice difficile da leggere.
    - Tutto ciò che si può fare con le macro può essere fatto con le funzioni del C++.

- Le `#include` dovrebbero precedere tutte le dichiarazioni non legate ai preprocessori, nessuno noterà l'#include nel mezzo di un file.

Slide 3 2023 - Librerie standard C++; Utilizzo Auto,Decltype,For

▼ Librerie standard del C++

Alcuni esempi:

- string
- vector (array di dimensioni variabili)
- ifstream
- ofstream
- cin
- cout

▼ Il tipo string

- Permette di operare facilmente sulle stringhe grazie a dei metodi e degli attributi presenti nella classe.
- Permette di convertire la stringa in altri tipi molto facilmente.
- La classe string, come la maggior parte degli altri tipi di libreria, definisce anche dei tipi membro, ad esempio: `size_type` e `iterator`
- Alcune operazioni con le stringhe:

```
s.empty() // restituisce true se s è vuoto; altrimenti restituisce false  s.size()
s[n] // restituisce un riferimento al carattere in posizione n in s
s.at(n) // simile a s[n] ma con controllo dei limiti
s1 + s2 // restituisce una stringa che è la concatenazione di s1 e s2  s1+= s2
s1 = s2 // sostituisce i caratteri in s1 con una copia di s2
s1 == s2 // uguali se contengono gli stessi caratteri
s1 != s2 // non uguali
<, <=, >, >= // confronto utilizzando l'ordinamento del dizionario
os << s // scrive s sul flusso di uscita os; restituisce os
```

```
is >> s // legge una stringa separata da spazi bianchi da is; restituisce is
std::getline(is, s) // legge una riga di input da is in s; restituisce is
```

▼ Specificatore automatico di tipo: `auto`

- Per memorizzare il risultato di un'espressione in una variabile abbiamo bisogno di conoscere il tipo dell'espressione, ma a volte è molto prolisso o difficile da ricavare, quindi è possibile lasciare che il compilatore deduca il tipo con la parola chiave `auto`:

```
auto x = espressione;
//ad esempio:
auto y = val1 + val2;
auto z = doSomething();

//Una variabile definita tramite auto va sempre inizializzata, poichè aktrim
auto x1; // Non si compila

int num;
auto x1 = num // Si compila
```

- `auto` ignora la costanza dei tipi (ma non la costanza dei tipi puntati, cioè un puntatore a const):

```
const int ci = i, &cr = ci;
auto b = ci; // b è un int (il livello superiore di const in ci viene abbandona
auto c = cr; // c è un int (cr è un alias di ci il cui const è di primo livello)
auto d = &i; // d è un int* (& di un oggetto int è int*)
auto e = &ci; // e è una const int* (& di un oggetto const è const di basso

//Se vogliamo mantenere la costanza dobbiamo specificarlo:
const auto f = ci;
```

- Possiamo anche chiedere un riferimento ad auto:

```
auto& g = ci; // g è una const int& legata a ci
```

- `auto` semplifica le dichiarazioni complesse come in presenza di iteratori, contenitori STL, o puntatori intelligenti:

```
std::shared_ptr<some_type_t> mySmartPtr= std::make_shared<some_typ
// |
// V
auto mySmartPtr= std::make_shared<some_type_t>();

//*****************************************************************

for (std::map<std::string,std::map<std::string,int>>::iterator mapIter= myC
// |
// V
for (auto mapIter= myContainer.begin(); mapIter!=myContainer.end(); ma
// |
// V
for (const auto& iter : myContainer)
```

▼ Specificatore di tipo: `decltype`

- Se si vuole dedurre il tipo di una variabile utilizzando un espressione, come nel caso di `auto`, ma non si vuole che la variabile prenda come valore il risultato di tale espressione, allora si può utilizzare `decltype`, come ad esempio:

```
decltype(f()) sum = x;
// La variabile sum prednte il tipo restituito da f, ma prende il valore di x
```

- A differenza di auto, decltype prende come tipo anche il riferimento & e il const.
- decltype restituisce un tipo di riferimento per le espressioni che producono oggetti che possono stare sul lato sinistro dell'assegnazione:

```
int i = 2, *p = &i, &r = i;
decltype(r + 0) b; // ok: l'addizione restituisce un int; b è un int (non inizia
decltype(*p) c= i; // ok: c è un int& inizializzato a i
decltype(*p) d; // errore: c è un int& e deve essere inizializzato
```

▼ Ciclo for su sequenze (`foreach`)

- Se vogliamo scorrere tutti gli elementi di una sequenza, invece del for classico possiamo utilizzare il for-each, che in C++ ha la seguente sintassi:

```
for (dichiarazione : espressione) {...}
//Dove dichiarazione indica il tipo che utilizziamo per scorrere tale seque
//Mentre dichiarazione indica la sequenza che vogliamo scorrere

//Esempio:
string str("ciao");
for (char c : str) { // che possiamo leggere come: foreach character c in s
   cout << c << " ";
}
//se volessimo cambiare i caratteri, potremmo definire la variabile del cicl
```

▼ Operazioni con i caratteri: libreria `cctype`

- La libreria `cctype` contiene i metodi necessari per maneggiare al meglio i caratteri, ogni oggetto della collezione ha un indice associato, che dà accesso a quell'oggetto.
- Alcune operazioni:

```
isalnum(c) // true se c è una lettera o una cifra
isalpha(c) // true se c è una lettera
iscntrl(c) // true se c è un carattere di controllo
isdigit(c) // true se c è una cifra
isgraph(c) // true se c non è uno spazio ma è stampabile
islower(c) // true se c è una lettera minuscola
isprint(c) // true se c è un carattere stampabile
```

```
ispunct(c) // true se c è un carattere di punteggiatura
isspace(c) // true se c è uno spazio bianco (spazio, tab, newline, ecc.) [illegible] isup
isxdigit(c) // true se c è una cifra esadecimale
tolower(c) // restituisce il suo equivalente minuscolo
toupper(c) // restituisce il suo equivalente maiuscolo
```

▼ Il tipo vector

- Un vettore (`vector`) è un insieme di oggetti, tutti dello stesso tipo
- Un vettore è un modello di classe (`template`)
  - I template sono istruzioni per il compilatore per la generazione di classi o funzioni.
  - Quando usiamo un template, specifichiamo che tipo di classe o funzione vogliamo che il compilatore istanzi.
  - Forniamo questa informazione all'interno di una coppia di parentesi angolari: `<>`
- Definire un vettore:

```
vector<int> ivec; // ivec contiene oggetti di tipo int
vector<string> nomi; // contiene stringhe
vector<vector<string> file; // vettore i cui elementi sono vettori che a loro
```

- I modi più comuni per definire i vettori:

```
vector<T> v1; // vettore vuoto che contiene oggetti di tipo T
vector<T> v2(v1); // v2 ha una copia di ogni elemento di v1
vector<T> v2 = v1; // equivalente a v2(v1)
vector<T> v3(n, val); // v3 ha n elementi con valore val
vector<T> v4(n); // v4 ha n copie di un oggetto inizializzato di default
vector<T> v5{a,b,c . . .}; // v5 ha tanti elementi quanti sono gli inizializzat
vettore<T> v5 = {a,b,c . . . }; // equivalente a v5{a,b,c . . . }
```

- Generalmente quando creiamo un vettore, non sappiamo quanti elementi ci serviranno, o non conosciamo il valore di tali elementi, quindi ci

avvaliamo del metodo `push_back` per aggiungere elementi, ad esempio:

```
vector<int> v1; // vettore vuoto
for (int i = 0; i < 100; i++) v1.push_back(i);
// aggiunge interi sequenziali a v1 alla fine
```

- Un errore comune è quello di voler provare ad aggiungere un elemento ad un vettore tramite l'operatore `[]`, però purtroppo questo non è possibile, utilizzare SEMPRE `push_back`.

▼ Operazioni con i vettori

```
v.empty() // restituisce true se v è vuoto; altrimenti restituisce false
v.size() // restituisce il numero di elementi in v
v.push_back(t) // aggiunge un elemento con valore t alla fine di v
v[n] // restituisce un riferimento all'elemento in posizione n in v
v1 = v2 // sostituisce gli elementi in v1 con una copia degli elementi in v2
v1= {a,b,c . . . } // sostituisce gli elementi in v1 con una copia degli elementi n
v1 == v2 // v1 e v2 sono uguali se hanno lo stesso numero di elementi e ogni
v1 != v2 // v1 e v2 non sono uguali
<, <=, >, >= // hanno il loro significato normale utilizzando l'ordinamento del
```

▼ Iteratori

- Sebbene sia possibile utilizzare i pedici per accedere ai caratteri di una stringa o agli elementi di un vettore, è possibile usare gli `iteratori`, sono oggetti simili ai puntatori, associati a classi che contengono sequenze e sono il meccanismo più generale per accedere agli elementi di una sequenza.
- Alcuni iteratori possono essere ad esempio: `std::string::iterator` e `std::vector::iterator`.
- Come i puntatori, gli iteratori ci danno accesso indiretto a un oggetto, possiamo usare un iteratore per recuperare un elemento e gli iteratori hanno operazioni per spostarsi da un elemento all'altro.

- Come per i puntatori, un iteratore può essere valido o non valido, un iteratore valido o denota un elemento o denota una posizione successiva all'ultimo elemento di un contenitore.
- A differenza dei puntatori, non si usa l'operatore address-of (`&`) per ottenere un iteratore, di solito ci sono funzioni membri che restituiscono iteratori, come begin() e end():

```
vector<int> v={1,2,3,4,5};
vector<int>::iterator b = v.begin(), e = v.end();// b ed e hanno lo stesso tipo
cout << *b << endl;
++b;
cout << *b << endl; cout << *e << endl;
```

- Alcuni esempi:

```
string s("ciao mondo");
for (auto it=s.begin(); it!=s.end(); ++it)
    *it= toupper(*it);
```

```
string s("ciao mondo");
if (s.begin() != s.end()) { // assicurarsi che s non sia vuoto
    auto it = s.begin(); // indica il primo carattere in s
    *it = toupper(*it); // trasforma il carattere in maiuscola
}
```

▼ Operazioni con gli iteratori

```
*iter // restituisce un riferimento all'elemento indicato dall'iteratore
iter→memb // dereferenzia iter e recupera il membro chiamato memb dall'ele
++iter // incrementa iter per riferirsi all'elemento successivo
--iter // diminuisce iter per riferirsi all'elemento precedente
iter1 == iter2 // confronta due iteratori per la disuguaglianza di uguaglianza; c
iter1 != iter2 // confronta due iteratori per la disuguaglianza
iter + n // restituisce un iteratore che denota n elementi in avanti
```

```
iter - n // restituisce un iteratore che denota n elementi in avanti
iter+= n
iter-= n
>, >=, <, <= // un iteratore è minore di un altro se si riferisce a un elemento c
```

▼ Differenti tipi di iteratori

```
vector<T>::iterator // può leggere e scrivere elementi di un vettore<T>
string::iterator // può leggere e scrivere i caratteri di una stringa
vector<T>::const_iterator // può leggere ma non scrivere elementi
string::const_iterator // può leggere ma non scrivere caratteri
```

- Il tipo restituito da `begin` e `end` dipende dal fatto che l'oggetto su cui operano sia o meno `const`

▼ IO stream

- La libreria standard include una famiglia di tipi (classi) per maneggiare operazioni di IO, questa libreria definisce meccanismi e operazioni per poter leggere e scrivere valori di tipi elementari quali int, float, etc.
- In aggiunta, diverse classi, come per esempio string, tipicamente definiscono operazioni di IO simili per lavorare anche su oggetti del loro tipo.
- File intestazione per IO e principali tipi definiti:

```
<iostream>
• istream // leggono da uno stream,
• ostream // scrivono su uno stream,
• iostream // possono sia leggere che scrivere su uno stream

<fstream>
• ifstream // legge da un file,
• ofstream // scrive su un file,
• fstream // legge e scrive un file

<sstream>
```

- istringstream // legge da una stringa (string stream)
- ostringstream // scrive su una stringa
- stringstream // legge e scrive una stringa

- I tipi `ifstream` e `istringstream` ereditano da `istream`, quindi oggetti di questi tipi li possiamo usare come oggetti generici `istream` (stream di ingresso).
- Ad esempio possiamo utilizzare oggetti di questi tipi nello stesso modo in cui utilizziamo `cin`, per esempio, possiamo chiamare `getline()` su un oggetto `ifstream` o `istringstream`, e possiamo usare la funzione `>>` per leggere i dati da un `ifstream` o `istringstream`.
- Allo stesso modo, i tipi `ofstream` e `ostringstream` ereditano da `ostream`.
- Non è possibile copiare o assegnare oggetti di tipo IO stream:

```
ofstream out1, out2;
out1 = out2; // errore: non è possibile assegnare/copiare oggetti stream

ofstream print(ofstream ofs){...}
// errore: non è possibile inizializzare il parametro ofstream; se necessario, b
// errore: impossibile restituire copia di uno stream; usare riferimento
```

▼ Stato della condizione IO

- Facendo operazioni di IO è possibile che si verifichino errori, quindi le classi IO definiscono funzioni e flag che ci permettono di accedere e manipolare lo stato di condizione di un flusso.
- Una volta che si è verificato un errore, le successive operazioni di IO su quello stream falliranno.
- Poiché uno stream potrebbe trovarsi in uno stato di errore, il codice dovrebbe normalmente verificare se uno stream è corretto prima di tentare di utilizzarlo.
- Il modo più semplice per determinare lo stato di un oggetto stream è utilizzare tale oggetto come condizione:

```
while (cin >> word)
// ok: operazione di lettura riuscita ...
```

- Esempio:

```
int i;
while (cin >> i) {
    // here if an int is correctly read
}
cout << "not an integer\n";
cout << cin.rdstate();
//Il metodo rdstate() restituisce un iostate intero che codifica lo stato dello str
```

- E' possibile interrogare lo stato tramite appositi metodi che restituiscono `true` o `false`, ad esempio:

```
eof()
fail()
etc...
```

▼ Input e output su file

- L'intestazione `fstream` definisce tre tipi per supportare l'IO su file:
  - ifstream per leggere da un dato file
  - ofstream per scrivere su un dato file
  - fstream, che permette di leggere e scrivere su un dato file
- Questi tipi forniscono le stesse operazioni che abbiamo usato in precedenza sugli oggetti `cin` e `cout`.
- Esempio scrittura di una stringa in un file:

```
ofstream fileout;
fileout.open("temp.txt");
```

```
fileout << "Hello world!\n";
fileout.close();
```

- Esempio lettura di stringhe da un file:

```
ifstream infile("temp.txt");
if (!infile) {
    cout << "Impossibile aprire il file";
    return 1;
}

std::string str;
while (infile>>str)
    std::cout << str << '\n';

infile.close();
```

▼ Operazioni specifiche di fstream

```
fstream fs // crea un flusso di file non vincolato
fstream fs(name) // crea un fstream e apre il file denominato 'name'
fstream fs(name, mode) // come il costruttore precedente, ma apre 'name' ne
fs.open(name), fs.open(name, mode) // apre il file denominato da name e lo l
fs.close() // chiude il file
fs.is_open() // restituisce un bool che indica se il file è stato aperto con succe
```

▼ String stream

- L'header `sstream` definisce tre tipi per supportare l'IO in memoria, questi tipi leggono da o scrivono su una stringa come se la stringa fosse un flusso IO.
- Alcune operazioni specifiche di stringstream:

```
sstream ss; // sstream vuoto
sstream ss(str); // sstream che contiene una copia di str
```

```
ss.str() // restituisce una copia della stringa contenuta in ss
```

▼ Buffer di uscita

- Ogni flusso di uscita contiene un buffer usato per mantenere i dati che si stanno scrivendo e per ottimizzare la funzione di scrittura sull'eventuale sistema collegato (e.g. un file o standard output).
- Quando si effettua una scrittura non è detto che questo buffer venga svuotato immediatamente.
- Ad esempio, quando viene eseguito il seguente codice: `os << "inserire un valore: ";` la stringa letterale potrebbe essere stampata immediatamente, oppure il sistema operativo potrebbe memorizzare i dati in un buffer per stamparli successivamente.
- Se si vuole forzare la scrittura immediatamente si può usare un manipolatore come `endl` o `flush`, come ad esempio:

```
cout<<"ciao"<<flush;
```

▼ Supporto caratteri estesi (wchar_t)

- Per supportare lingue con alfabeti diversi, la libreria definisce un insieme di tipi e oggetti che manipolano dati `wchar_t` (wide char o caratteri estesi).
- I nomi dei tipi, oggetti e funzioni con caratteri estesi iniziano con una w, ad esempio, `wcin`, `wcout` e `wcerr` sono gli oggetti per effettuare IO con caratteri estesi che corrispondono rispettivamente a `cin`, `cout` e `cerr`
- Sono definiti nello stesso file intestazione che contiene i tipi analoghi per caratteri semplici, ad esempio, l'intestazione `fstream` definisce sia i tipi `ifstream` che `wifstream`

Slide 4 2023 - Ereditierietà

▼ Ereditarietà

- Dopo il nome della classe derivata ci sono i due punti `:` seguiti dalla parola chiave "public" e poi dal nome della classe da cui si eredita.

- La parola chiave public dopo i due punti indica che stiamo usando l'ereditarietà pubblica.
- I differenti tipi di ereditarietà indicano se i membri pubblici della classe base sono o meno accessibili alla classe derivata
- Esempio:

```
class Person { public: const string& getName() const; // ... };

class Student : public Person { // ... }; class Staff : public Person { // ... };

class Permanent : public Staff { // ... }; class Casual : public Staff { // ... };
```

▼ Specificatori di accesso

		Accesso ai membri della classe base		
		pubblico	protetto	privato
Accesso all'ereditarietà delle classi derivate	pubblico	qualsiasi funzione	metodi di D amici di classi D derivate da D	non accessibili
	protetto	metodi di D amici di classi D derivate da D	metodi di D amici di classi D derivate da D	non accessibili
	privato	metodi di D amici di D	metodi di D amici di D	non accessibili

- Una classe derivata `public` eredita i membri pubblici e protetti della base mantenendo il loro livello di accesso.
- Una classe derivata `protected` eredita i membri pubblici e protetti della base ma li espone come protetti.
- Una classe derivata `private` espone i membri pubblici e protetti della base come privati

▼ Interfaccia di classe

- Una classe ha due interfacce distinte per due gruppi distinti di utenti:

    - Un'interfaccia pubblica che serve le classi esterne (non correlate).
    - Un'interfaccia protetta che serve le classi derivate.

▼ Membri privati

- I membri privati di una classe rimangono solo privati, una classe derivata NON PUO' accedere ai membri privati della classe base, anche se li eredita (sono inclusi in un oggetto della classe derivata).
- I membri privati sono accessibili solo attraverso i metodi pubblici della classe base.

▼ Membri protetti

- Se vogliamo che un membro della classe sia visibile ai metodi di una classe derivata, ma non agli oggetti/utenti della classe base o della classe derivata, allora lo definiamo come protetto.
- Se ci sono livelli di ereditarietà indiretta attraverso una gerarchia di classi, i membri protetti saranno accessibili in tutta la gerarchia di classi
- Esempio:

```
class BaseClass {

public:
    void method1();

protected:
  void method2();
};

class DerivedClass : public BaseClass {

public:
    void method3() {
        method2(); // OK
    };
```

```
};

DerivedClass d;
d.method1(); // OK
d.method3(); // OK
d.method2(); // ERROR! method2 is protected
```

▼ Accesso ai membri della classe base

- Un oggetto di una classe derivata eredita i membri della classe base.
- “Il vero potere dell’ereditarietà è quando non sappiamo il tipo di un oggetto” - Gandhi
- L’ereditarietà è un esempio di polimorfismo.

▼ Ereditarietà vs. Composizione

- L’ereditarietà estende una classe già esistente.
- La composizione è quando si utilizza una classe all’interno di un’altra classe.
- Esempio:

```
//Esempio eredit.
class Person {
private:
    string name;

public:
    int getName(){
        return member;
    };
};

class Student : public Person {
 //...
};
```

```
int main(){
    Student s;

    cout << "Name: " << s.getName();
}
//Esempio comp.

class Student {
public:
    Person details;
    // ...
};

int main() {
    Student s;
    cout << "Name: " << s.details.getName();
    // Notare che in questo caso, per accedere alla funzione della classe Perso
}
```

- Usare l'ereditarietà per le relazioni "è_un" ("is_a"), mentre la composizione per le relazioni "ha_un" ("has_a") o "contiene" o "è_composto_da".
- Considerare il caso di istanze multiple di una classe all'interno di un'altra classe, ad esempio:

```
class Person {
    public:
    Address home;
    Address office;
    // ...
};
//Questo non si può fare con l'ereditarietà
```

▼ Usare le classi derivate

- È possibile usare un oggetto istanziato da una classe derivata ogni volta che è possibile usare un oggetto istanziato dalla classe base:

```
class Employee {
    string first_name, family_name;
    Date hiring_date;
    short department;
    // ...
};

class Manager : public Employee {
    set<Employee*> group;
    short level;
  // ...
};

void paySalary(const Employee& e) {
    //... code to pay salary
}

int main(){
    Employee e1;
    Manager m1;
    //...
    paySalary(e1);
    paySalary(m1);
    // Possiamo utilizzare la funzione paySalary anche con un oggetto della cla
    // poichè estende la classe Employee
}
```

▼ Eredità pubblica e "è_un”

- Se D estende pubblicamente B, allora D “è_un” B e qualsiasi funzione che si aspetta un B (o un puntatore a B o un riferimento a B) prenderà anche D (o un puntatore a D o un riferimento a D).
- Esempio:

```
class Person {...};

class Student : public Person {...};

void eat(const Person& p);
void study(const Student& s);

int main(){
    Person p;
    Student s;
    eat(p); // OK
    eat(s); //OK: s "è_un"
    p study(s); // OK
    study(p); // NO: p non "é_un" s
}
```

- L'ereditarietà pubblica afferma che tutto ciò che vale per l'oggetto base vale per l'oggetto derivato.

▼ Principio di sostituzione di Liskov

- La `sostituibilità` è un principio della programmazione orientata agli oggetti che afferma che se S è un sottotipo di T, allora gli oggetti di tipo T possono essere sostituiti con oggetti di tipo S.
- In altre parole, se si crea una classe che estende una classe base, essa non dovrebbe alterare in modo significativo il comportamento dei suoi genitori

▼ Ereditarietà privata

- Il comportamento è molto diverso quando si eredita privatamente, non abbiamo più una relazione "è_un", il compilatore non convertirà la classe derivata in base:

```
class Student : private Person { ... };

void eat(const Person& p);
```

```
Student s;
eat(s); // error: now a Student is not a Person
```

- Tutto ciò che viene ereditato diventa privato, è un dettaglio di implementazione.
- Ereditarietà privata significa che la classe derivata D è_implementata_basata_su classe base B, non che D è_un B.
- Usare l'ereditarietà privata se si vuole ereditare l'implementazione della classe base, usare l'ereditarietà pubblica per ottenere anche l'interfaccia.
- Usare la composizione ogni volta che si può e l'ereditarietà privata quando è necessario, per esempio quando:
  - Si deve accedere a parti protette di una classe
  - Si devono ridefinire metodi virtuali (vedi più avanti)

▼ Costruttori ed ereditarietà

- Quando viene creato un oggetto di una classe derivata, i costruttori (se esistono) di ogni classe ereditata vengono invocati in sequenza prima del costruttore finale della classe (se esiste), è un processo bottom-up.
- I costruttori predefiniti vengono invocati automaticamente.
- Se una classe base non ha un costruttore predefinito, un altro costruttore deve essere invocato esplicitamente dal costruttore della classe derivata nella sua lista di inizializzazione.

▼ Distruttori ed ereditarietà

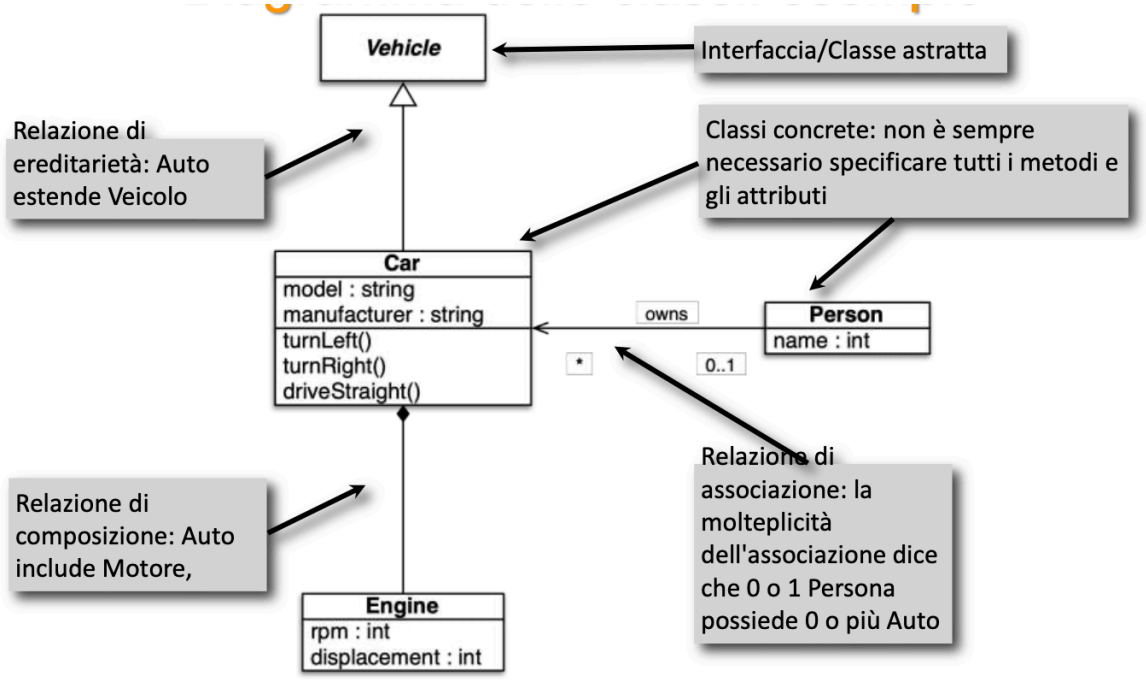
- Proprio come i costruttori, tranne per il fatto che l'ordine è invertito, è un processo dall'alto verso il basso (top-down).
- Quando una classe derivata viene distrutta, viene invocato prima il distruttore della classe derivata e poi il distruttore della classe base.
- I distruttori non vengono sovraccaricati o invocati esplicitamente, in modo da non creare confusione su quale distruttore venga invocato

▼ UML

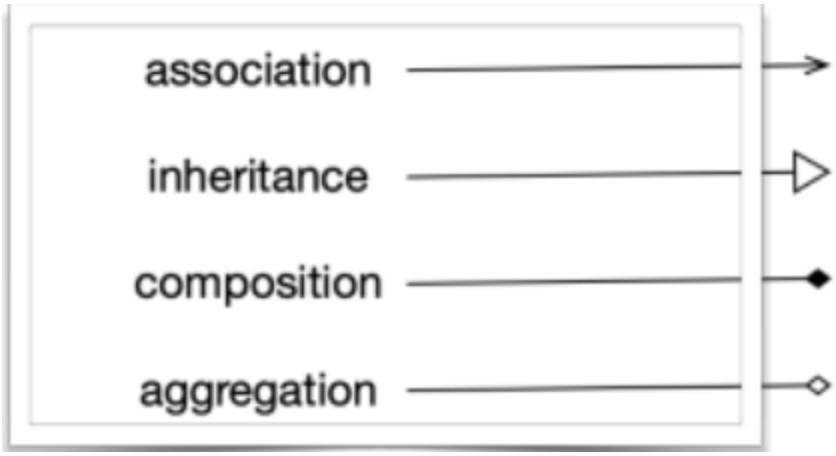
- UML (Unified Modeling Language) è un linguaggio visivo per specificare, costruire e documentare dei sistemi.
- Principali tipi:
  - Diagramma dei casi d'uso (Use Case diagram): insiemi di casi d'uso, in cui sono presenti gli attori e loro relazioni.
  - Diagramma delle attività (Activity diagram): consiste in attività e collegamenti tra queste e descrive il flusso di controllo del sistema.
  - Diagramma delle classi (Class diagram): diagramma UML più usato, consiste nella descrizione delle classi, interfacce e loro associazioni e collaborazioni e rappresenta la visione orientata agli oggetti del sistema.
  - Diagramma di sequenze (Sequence diagram): diagramma delle interazioni che descrive il flusso di messaggi e interazioni tra gli oggetti.

▼ Diagramma delle classi

- Un diagramma UML di classe descrive la struttura di un sistema mostrando le classi del sistema, i loro attributi e le relazioni tra le classi.
- Nel diagramma delle classi le classi sono rappresentate da riquadri che contengono tre parti:
  1. La parte superiore contiene il nome della classe.
  2. La parte centrale contiene gli attributi della classe (e il loro tipo).
  3. La parte inferiore contiene i metodi o le operazioni che la classe può fare o intraprendere.
- Ci sono strumenti che consentono di generare codice a partire dai diagrammi UML delle classi, o di effettuare il reverse engineering del codice in diagrammi delle classi
- Esempio:



▼ Relazioni tra classi



- **L'associazione** rappresenta una famiglia di collegamenti, può essere denominata e le estremità possono essere arricchite con nomi di ruoli, indicatori di proprietà, molteplicità, ecc.
- **L'aggregazione** è una variante più specifica della associazione ed è di tipo "ha_un", rappresenta una relazione parte-intera o parte-di e può verificarsi quando una classe è una collezione o un contenitore di altre classi, senza che le classi contenute abbiano una forte dipendenza nel ciclo di vita dal contenitore.

- **La composizione** rappresenta relazioni tra oggetti e loro parti fisiche, più "fisica" dell'aggregazione (e.g. un motore è parte di un'auto).

▼ Polimorfismo

- Nei linguaggi di programmazione e nella teoria dei tipi, il polimorfismo (cioè "molte forme", dal greco) è la fornitura di una singola interfaccia a entità di tipi diversi.
- Esistono diversi tipi di polimorfismo, il C++ li fornisce tutti utilizzando:
  - sovraccarico
  - sotto-tipizzazione: ereditarietà e funzioni virtuali in C++
  - programmazione generica (generic programming): C++ templates(modelli)
- Il concetto principale del polimorfismo dei sottotipi: possiamo riferirci a un'istanza di una sottoclasse come se fosse un'istanza della sua superclasse, ma ogni oggetto risponde alle chiamate di metodo come specificato dal suo tipo effettivo.
- In C++ una classe derivata può sovrascrivere (overriding) un metodo ereditato da una classe base il metodo sovrascritto può richiamare il metodo di base.
- Il metodo “sovrascritto” è polimorfico perché ha un'implementazione diversa a seconda che sia invocato sulla classe base o su quella derivata

▼ Overriding vs. overloading

- Metodo sovraccaricato: stesso nome del metodo ma parametri diversi (nella stessa classe).
- Metodo sovrascritto: stesso nome e parametri, ridefinito all'interno di una gerarchia di classi.

- I metodi sovraccaricati in una classe base possono essere sovrascritti in una classe derivata.
- Una classe derivata può sovraccaricare un metodo sovrascritto, aggiungendo un nuovo comportamento alla sua interfaccia.

▼ Polimorfismo statico

- La funzione di sovrascrittura consente l'esistenza di diverse implementazioni di un metodo, questo introduce il problema di vincolare l'invocazione di un metodo a una particolare implementazione.
- La decisione si basa sul tipo di classe utilizzata per riferirsi a un metodo.
- Questo è fatto a tempo di compilazione: associazione statica (o anticipata) (Static Binding).

▼ Polimorfismo dinamico: metodi virtuali

- I metodi virtuali evitano che l'utente di una classe debba conoscere il tipo concreto dell'istanza che sta utilizzando.
- Uno o più metodi di una classe possono essere dichiarati come `virtual` aggiungendo la parola chiave nella loro dichiarazione: Associazione dinamica (late binding).
- Un metodo virtuale nella classe base rimane virtuale nelle classi derivate
- Per capire meglio, come sappiamo, tramite il polimorfismo, possiamo creare una array del tipo “super-classe” e all’interno inserire anche degli elementi del tipo “sotto-classe”, ma notiamo che se utilizziamo una funzione della super-classe, nonostante sia stata sovraccaricata nella sotto-classe, la funzione che verrà utilizzata sarà sempre quella della super-classe, per fare si che ciò non accada, possiamo utilizzare la parola chiave `virtual` sulla funzione della super-classe per utilizzare nel caso specifico quella sovraccaricata, come nel seguente esempio:

```
class Student {

public:
  string name;
  Student(string name) : name(name) {} // Costruttore
  virtual void print() {
     cout << name << endl;
   }
};
```

```
class MedicalStudent : public Student {
public:
  string speciality;
  MedicalStudent(string name, string speciality) : speciality(speciality), Stude
  void print() { cout << name << ": " << speciality << endl; }
};

int main()
{
  Student *students[] =
  {
    new Student("Kevin"),
    new Student("Mary"),
    new Student("Kulvinder"),
    new MedicalStudent("Ali", "Family Medicine"),
    new MedicalStudent("Harry", "Pediatrics"),
    new MedicalStudent("Lisa", "Radiology")
  };

  for (int i = 0; i < 6; i++)
    students[i]→print();
    // Senza virtual manderebbe a video solo i nomi degli studenti di medicina,
    // ma non manderebbe a video la loro specializzazione come vorremmo

  for (int i = 0; i < 6; i++)
    delete students[i];

  return 0;
}
```

- Grazie al late binding un puntatore (o un riferimento) di tipo classe base non ha bisogno di sapere a quale tipo sta puntando.
- I metodi virtuali sono la struttura chiave del polimorfismo, la funzione che viene invocata utilizzando un puntatore (o un riferimento) alla classe base

può avere diverse forme, a seconda del tipo di oggetto che viene utilizzato.

- Una funzione in una classe derivata con la stessa firma di una funzione virtuale nella classe base sarà virtuale anche se non contrassegnata come virtuale, è bene marcarla comunque come virtuale.
- Le funzioni al di fuori di qualsiasi classe non possono essere virtuali.
- Una definizione separata (cioè non all'interno della dichiarazione della classe) di una funzione virtuale non è marcata come virtuale.

▼ Override e final

- `override` indica che un metodo in una classe derivata intende essere un override di un metodo virtuale nella classe base, usare la parola chiave override solo se il metodo della super-classe è virtual.

```
Class cl1 {
   //...
   virtual void f1(int i) const;
   void f2();
   //...
};

Class cl2 : cl1 {
   void f1(int i) override; // OK
   void f2() override; // NO f2() non è virtual
};
```

- `final` indica che un metodo di una classe base non può essere sovrascritto in una classe derivata.

```
Class cl1 {
   //...
   final void f1(int i) const;
   void f2();
   //...
```

```
};

Class cl2 : cl1 {
   void f1(int i); // NO f1() è final
   void f2(); // OK
};
```

- `final` può anche bloccare la possibilità di derivare da una classe:

```
classe SuperCar final : public Car{...};

//Non è possibile derivare la classe SuperCar
```

▼ Costruttori e distruttori virtuali

- I costruttori non possono essere virtuali.
- I distruttori possono essere virtuali e fanno in modo che vengano invocati quelli giusti.

▼ Distruttori virtuali

- Se una classe non contiene un metodo virtuale, allora probabilmente non è destinata a essere una classe base o è una classe base da non usare in modo polimorfico.
- Non è utile dichiarare un distruttore virtuale se non ci sono altri metodi virtuali nella classe: si spreca memoria per la creazione della tabella virtuale utilizzata per gestire le funzioni virtuali.

▼ Tipo di ritorno covariante

- Un metodo sovrascritto in una classe derivata può restituire un tipo derivato dal tipo restituito dal metodo della classe base, esempio:

```
class A {};
class B : public A {};
class C : public B {};
```

```
class X {
public:
   virtual B *method1() {
      return new B();
   }
};

class Y : public X {
public:
   virtual C *method1() {
      return new C(); // La classe C è derivata dalla classe B, quindi posso farl
   }
};

int main() {
   X x;
   Y y;
   x.method1();
   y.method1();
}

// Da notare come Y::method1() non può restituire un oggetto B
// se X::method1() restituisce un oggetto C
```

- Il tipo di ritorno covariante è utile per implementare alcuni progetti, come permettere a una classe di clonare gli oggetti:

▼ Name hiding

- Se una classe base dichiara una funzione membro e una classe derivata dichiara una funzione membro con lo stesso nome ma con diversi tipi di parametri e/o costanti, allora il metodo base è "nascosto" piuttosto che "sovraccaricato" o "sovrascritto" (anche se il metodo è virtuale)

▼ Classi astratte

- Una classe astratta è una classe in cui è dichiarato almeno un metodo come virtuale senza implementazione, una base da cui partire per definire

altre classi concrete, se i metodo virtuali che non sono definiti sono detti astratti o virtuali puri.

- Una classe astratta pura è una classe che ha solo metodi astratti, cioè senza alcuna implementazione dei suoi metodi.
- In altri linguaggi di programmazione una classe astratta pura (con alcune differenze rispetto al C++) è chiamata interfaccia.
- Questo è utile perché un utente può affidarsi all'"interfaccia" fornita da una classe astratta senza dover conoscere i dettagli delle classi che la implementano.
- Una funzione virtuale pura viene dichiarata utilizzando una sintassi speciale:

```
virtual void abstractMethod() = 0;
// La funzione di cui sopra non ha bisogno di essere definita,
// poiché non esiste realmente e non può essere mai chiamata
```

- Una classe derivata da una classe base astratta deve sovrascrivere tutte le sue funzioni virtuali pure o sarà anch'essa una classe astratta.
- Esempio:

```
class Shape {
public:
    virtual void draw() const = 0;
};

class Square : public Shape {
public:
    virtual void draw() const; // implemented
};

class Circle : public Shape {
public:
    virtual void draw() const; // implemented
```

```
};

void drawAllShapes(Shape* list[], int size) {
   for(int i=0; i < size; i++) {
      list[i]→draw();
   }
}
```

- Utilizzando le classi astratte possiamo creare una `gerarchia delle classi`.

▼ Eliminazione di metodi

- In C++ è possibile non consentire la definizione di alcune funzioni aggiungendo "= delete" alla fine della dichiarazione di un metodo:

```
class A {
public:
   A(int a) {};
   A(double) =delete; // conversion from double disabled
   A& operator=(const A&) =delete; // assignment disabled
};
```

- Può essere applicato a qualsiasi metodo di una classe, come anche ad uno dei metodi creati automaticamente dal compilatore.
- Può essere applicato ai metodi ereditati da una classe base.

▼ Conversione di tipo

- Vecchio stile per fare "cast" (simile al C):

```
(tipo)espressione; // C-style cast notation
tipo(espressione); // function-style cast notation

//Esempio:
floatg=((float)i)/2;
floath=(float(i))/2;
```

- Stile C++:

```
static_cast<tipo>(espressione)
// Viene usato per la conversione esplicita di tipi non polimorfici
// Il cast viene effettuato durante la compilazione
// Forza le conversioni tra tipi primitivi come da int a double, da void* a punta
// Forza le conversioni da puntatori (o riferimenti) a classe base a puntatori o
// riferimento ad una classe derivata, ma senza controllo a runtime

const_cast<tipo>(espressione)
// Viene usato per eliminare la costanza di un oggetto

reinterpret_cast<tipo>(espressione)
// Viene usato per la reinterpretazione a basso livello de byte di memoria
// ad esempio per eseguire conversioni tra tipi non correlati,
// come la conversione tra puntatori e riferimenti non correlati
// o la conversione tra un intero e un puntatore

// Produce un valore di un nuovo tipo che ha lo stesso schema di bit del suo a
// può essere pericoloso: stiamo chiedendo al compilatore di fidarsi di noi

dynamic_cast<tipo>(espressione)
// Viene utilizzato per verificare a tempo di esecuzione se un cast è sicuro pe
// Funziona solo con tipi polimorfici, cioè classi che hanno almento un metod
// Esegue un downcasting sicuro, con controllo a runtime: run-time type infor
// Determina se un oggetto è di un particolare tipo in una gerarchia ereditaria
// Il tipo sorgente (tra parentesi tonde) deve essere un puntatore o un riferime
// Il tipo di destinazione (tra parentesi angolari) deve essere un puntatore o u
```

- In generale, meno cast si fanno e meglio è

▼ Funzioni membro const

- Tramite `const_cast` è possibile evitare la duplicazione del codice tra funzioni membro const e non const.

- il metodo non-const richiama il metodo const e poi ne annulla la costanza con const_cast

▼ Eredità multipla

- Una classe può ereditare da (estendere) più classi contemporaneamente.
- E' più complessa dell'eredità singola, la gerarchia di ereditarietà non è più un albero ma un grafo
- Esempio di sintassi:

```
class Consultant: public Temporary, public Manager { /*...*/ };
```

- Possono verificarsi conflitti di nome nel caso in cui i membri delle classi base abbiano lo stesso nome (risolvibili con l'uso appropriato delle dichiarazioni o con la qualificazione completa dei nomi)
- I costruttori di ciascuna classe base saranno invocati prima del costruttore della classe derivata.
- I distruttori saranno invocati allo stesso modo, ma nell'ordine inverso.
- I costruttori di base sono chiamati nell'ordine dell'ereditarietà, esempio:

```
class Consultant: public Temporary, public Manager {
public:
    Consultant(); //Viene chiamata prima Temporary() e poi Manager()
};
```

- Le ambiguità sui nomi dei membri possono essere risolte utilizzando la risoluzione dell'ambito (`::`)

▼ Problema del diamante

- Un altro problema importante di ambiguità che si può avere con l'eredità multipla è quando due classi B e C ereditano da A, e la classe D eredita sia da B che da C, questo problema è detto problema del diamante.
- In questo caso, secondo le regole di ereditarietà, la classe A è ereditata, e quindi i suoi membri sono duplicati, sia in B che in C.

- Per risolvere questi problemi si utilizza l'ereditarietà virtuale che chiarisce ad esempio l'ambiguità su quali membri della classe base usare.

▼ Eredità virtuale

- Una classe base con eredità multipla può essere indicata come virtuale con la parola chiave virtual.
- In questo modo non vi è alcuna duplicazione della classe base ereditata, esiste una sola versione per un'istanza di oggetto di una classe derivata.
- Nella derivazione virtuale, la base virtuale viene inizializzata dal costruttore più derivato, altrimenti sarebbe inizializzata più di una volta lungo ogni percorso di ereditarietà che contiene la classe base virtuale.

Slide 5 2023 - Programmazione generica (Template)

▼ Programmazione generica

- In un linguaggio di programmazione con tipizzazione statica come il C++ il controllo avviene in fase di compilazione.
- In un linguaggio di programmazione con tipizzazione dinamica il controllo dinamico dei tipi avviene in fase di esecuzione.
-

▼ Compiti identici per tipi di dati diversi

- Un certo numero di algoritmi è indipendente dal tipo di dati.
- Similmente può avvenire per alcune classi che maneggiano elementi di altri tipi, come ad esempio strutture dati o classi contenitori
- Alcuni approcci possibili per funzioni che implementano compiti identici per diversi tipi di dati
  - Funzioni diverse con nomi diverse
  - Sovraccarico di funzioni
  - Utilizzo di macro
  - Modelli di funzioni e classi (C++ template)

- Questi approcci ci premettono anche di creare funzioni uniche con nomi unici per ogni combinazione di tipi di dati, come `atoi()` `atof()` `atol()`

▼ Sovraccarico di funzioni

▼ Utilizzo di macro

- Le macro sono utilizzabili sia con funzioni che con classi
- Tipi di dati parametrici vengono definiti tramite macro e risolti in fase di preprocessamento, prima della compilazione.
- Richiede di includere l'implementazione (*.cpp) della libreria per creare l'istanza specifica col tipo (o tipi) specificati tramite `#define`.
- Ma purtroppo è approccio complicato e poco flessibile.
- Esempio:

```
#define T int

class Array {
public:
    Array(int len): length{len}, data{new T[len]} {}
    // ...
    int length;
    T* data;
};
```

▼ Programmazione generica (template)

- Esistono 2 tipi di modelli:
  - Modelli di funzione (function template): funzioni che possono operare con tipi generici.
  - Modelli di classi (class template): possono avere attributi di tipi generici e possono avere membri che usano tipi generici.
- I template del C++ sono usati solitamente per classi e funzioni che possono applicarsi a differenti tipi di dati, esempi comuni sono le classi contenitore e gli algoritmi STL, infatti algoritmi come `std::sort()` sono

programmati per essere in grado di ordinare sequenze di elementi di tipi diversi.

▼ Modelli e ereditarietà

- In C++ l'ereditarietà funziona anche con i template e supporta:
  - Una classe template basata su una classe template
  - Una classe template basata su una classe non template
  - Una classe non template basata su una classe template

▼ Codifica dei template in C++

- Il template inizia con l'intestazione `template<typename T>`
- Il tag T viene utilizzato ovunque sia richiesto il tipo base, anche se si può utilizzare la lettera o la combinazione di lettere che si preferisce.
- Spesso la lettera T viene usata nel caso di un parametro di template di tipo singolo.
- In generale, le classi (e le funzioni) template possono avere più argomenti di "tipo" e possono anche avere argomenti di "non tipo".
- In fase di compilazione vengono generate più versioni di una funzione o classe in base ai tipi (e parametri) indicati nel codice.

▼ Dichiarazione dei modelli di funzione

- Sintassi della dichiarazione:

```
template <typename identifier> function_declaration;
```

- La parola chiave template deve apparire prima dei template di funzione o di classe, seguita dall'elenco dei tipi generici o template e può avere più di un tipo generico.
- Utilizzo di una funzione template:

```
function<TemplateArgList>(FunctionArgList)
```

- Esempi:

```
template <typename myType> myType getMax (myType a, myType b) {
   if (a > b) {
      return a
   } else {
      return b
   }
}

//Utilizzando riferimenti const

template <typename myType> const myType& getMax(const myType& a, cor
   if (a > b) {
      return a
   } else {
      return b
   }
}
```

▼ Istanziare un modello di funzione

- ???Quando il compilatore istanzia un modello, sostituisce l'argomento del modello al parametro del modello in tutto il modello di funzione???
- Esempio:

```
//Specializzazione del tipo automatico (quando possibile)
int main() {
   int i=5, j=6, k;
   long l=10, m=5, n;
   k=getMax(i,j); // OK
   n=getMax(l,m); // OK

   cout << k << endl;
   cout << n << endl;

   k=getMax(i,l); // Wrong: can’t mix types
```

```
    return 0;
}

int main() {
    int i=5, j=6, k;
    long l=10, m=5, n;

    k=getMax<int>(i,j); // OK
    n=getMax<long>(l,m); //OK

    cout << k << endl;
    cout << n << endl;

    return 0;
}
```

▼ Tipi di template multipli

- è possibile definire dei template che gestiscono più di un tipo di variabile, ad esempio:

```
template <typename T1, typename T2> const T1& getMax(T1& a, T2& b) {
    if (a > (T1)b)
        return a;
    else
        return (T1)b;
}
// In questo caso abbiamo 2 tipi T1 e T2, e sapendo che la funzione deve ritor
// facciamo facciamo il cast dell parametro b quando lo ritorniamo
```

▼ Specializzazione dei modelli di funzione

- La specializzazione dei modelli di funzione permette di specializzare il codice per alcuni tipi specifici, come ad esempio:

```
template<typename T>
std::string stringify(const T& x) {
    std::ostringstream out;
    out << x;
    return out.str();
}

template<>
std::string stringify<bool>(const bool& x) {
    std::ostringstream out;
    out << std::boolalpha << x;
    return out.str();
}
```

▼ Modelli di classe

- Le classi possono avere membri che utilizzano parametri template come tipo, ad esempio:

```
template <typename T>
class Array {
public:
    Array(int len): length{len}, data{new T[len]} {}
    // ...
    int length;
    T* data;
};
```

- Esempio una classe che memorizza due elementi di qualsiasi tipo valido:

```
template <typename T>
class mypair {
private:
    T values[2];
public:
```

```
    mypair(T first, T second) {
        values[0]= first;
    values[1]= second;
    }
};
```

▼ Modello di classe: definizione non in linea

- Per definire un membro di funzione al di fuori della dichiarazione del modello di classe, fate sempre precedere la definizione dal prefisso template `<...>` come nel seguente esempio:

```
template <typename T>
class mypair {
    T values [2];
public:
    mypair(T first, T second) {
        values[0]=first;
        values[1]=second;
    }
    T maximum();
};

template <typename T>
T mypair<T>:: maximum() {
    //T retval= a>b? a : b; Forma compatta, può anche essere scritto come:
    T retval;

    if (a > b)
        retval = a;
    else
        retval = b;

    return retval;
}
```

```
int main () {
   mypair<int> myobject {100, 75};
   cout << myobject. maximum();
   return 0;
}
```

- Esempio:

```
template <typename T>
T mypair<T>:: maximum() {
  T ret= a>b? a : b;
   return ret;
}

// Ci sono tre T in questa dichiarazione del metodo:
// - la prima è il parametro del template
// - la seconda T si riferisce al tipo restituito dalla funzione
// - la terza T (quella tra parentesi angolari) specifica che il parametro templa
//   di questa funzione è anche il parametro template della classe
```

▼ Istanziare un modello di classe

- Gli argomenti dei modelli di classe devono essere espliciti.
- Il compilatore genera tipi di classe distinti chiamati classi template o classi generate.
- Quando istanzia un modello, il compilatore sostituisce l'argomento del modello con il parametro del modello in tutto il modello della classe, per garantire l'efficienza, il compilatore utilizza una politica di "istanziazione su richiesta" dei soli metodi richiesti.
- Per capire meglio questo concetto, avvaliamoci di un esempio:

```
// Per creare elenchi di diversi tipi di dati da una classe modello GList:
GList<int> list1;
GList<float> list2;
GList<string> list3;
```

```
list1.insert(356);
list2.insert(84.375);
list3.insert("Muffler bolt");

// Il compilatore genera 3 tipi di classe distinti:
// - GList_int list1;
// - GList_float list2;
// - GList_string list3;
```

- È possibile specializzare un template con un altro template:

```
template<typename T>
class Array {
   // . . .
};

Array<Array<int>> aai;
```

▼ Parametri predefiniti

- I modelli di classe possono avere argomenti di tipo predefinito.
- Come per i valori predefiniti degli argomenti di una funzione, il tipo predefinito di un modello offre al programmatore una maggiore flessibilità nella scelta del tipo ottimale per una particolare applicazione, ad esempio:

```
template <typename T, typename S= size_t > // Il secondo tipo sarà per forza
class Vector {
   /*..*/
};

Vector<int> ordinary; //second argument is size_t Vector<int, unsigned char>
```

- Se un modello ha valori predefiniti per tutti i suoi parametri, può essere istanziato senza usare alcun parametro:

```
template <typename T=int>
class Foo {
   //...
};

Foo<float> foo1;
Foo<double> foo2;
Foo<> foo3; // è un Foo<int>
```

▼ Parametri non di tipo

- I modelli possono anche avere parametri normali, simili agli argomenti delle funzioni.
- Similmente, possono avere anche valori predefiniti.
- Esempio:

```
template <typename T, int N= 3>
class Array {
public:
   int length= N;
   T data[N];
};
```

Slide 6 2023 - Strutture Dati

- Tipi di dati astratti e concreti
- Alcune strutture dati astratte
- Alcune strutture dati concrete
- Array
- Allocazione collegata
- Visitare una lista collegata
- Allocazione indicizzata

- Pila (Stack ADT)
- Operazioni su una pila
- Possibili implementazioni di una pila
- Pila basata su array
- Capacità dell'array
- Pila collegata
- Operazione push()
- Coda (Queue ADT)
- Operazioni su una coda
- Array-based Queue
- Aumento della capacità dell'array
- Linked Queue
- Coda doppia (Deque ADT) Double-Ended Queue
- Implementazioni di deque
- Lista astratta
- Lista indicizzata
- Operazioni su lista
- Lista con cursore (Positional List ADT)
- Operazioni su lista con cursore
- Lista basata su array
- Lista basata su lista collegata
- Albero con radice (Rooted Tree ADT)
- Definizione di albero (con radice)
- Terminologia degli alberi
- Cammino, altezza, livello, grado
- Interfaccia di ADT albero

- Alberi n-ari e alberi binari
- Albero binario
- Balanced Binary Tree
- Implementazione con struttura collegata (Linked Binary Tree)
- Implementazione (alberi) con array
- Attraversamento di un albero binario
- Visita in preordine
- Visita in postordine
- Visita in ordine
- Visita per livelli
- Albero binario: Nodi pieni
- Albero binario pieno
- alberi di espressioni matematiche
- Albero binario perfetto
- Albero binario completo
- Insieme (Set ADT)
- Interfaccia di Set ADT
- Implementazione dell'insieme: array booleano
- Implementazione dell'insieme: lista
- Implementazione dell'insieme: lista ordinata
- Aggiunta e ricerca di elementi in una lista ordinata
- Ricerca binaria
- Multiinsieme (Multset ADT)
- Interfaccia multiset
- Implementazione del multiset
- ADT Mappa

- Interfaccia di un ADT mappa
- Implementazione della mappa:tabella con indirizzamento diretto
- Implementazione della mappa: lista non ordinata
- Implementazione di mappe: lista ordinata
- Nozione intuitiva di mappa
- Funzione Hash
- Funzioni di compressione
- Tabelle Hash
- Risoluzione delle collisioni con concatenamento separato
- Risoluzione delle collisioni con indirizzamento aperto e ispezione lineare
- Ricerca con ispezione lineare
- Aggiornamenti con l'ispezione lineare
- Albero di ricerca binario (Binary Search Tree - BST)
- Ricerca
- Inserimento
- Cancellazione
- Albero binario bilanciato
- Alberi di ricerca binari autobilanciati
- Multimap ADT
- Interfaccia di Multimap ADT
- Implementazione di multimappa

Slide 7 2023 - Standard Template Library (STL)

- Standard Template Library (STL)
- Componenti STL
- Modello di base

- Contenitori STL
- Iteratori STL
- Algoritmi STL
- Iteratori
- Categorie di iteratori
- Contenitori STL
- Restrizioni sui tipi contenuti
- Inizializzazione dei contenitori
- Sequenze
- Sequenze: alcuni metodi
- vector
- Modi di attraversare un vector
- list
- vector vs. list
- deque
- Quando utilizzare quale contenitore di sequenze
- Contenitori di sequenze riassunti
- Contenitori associativi
- set
- map
- Chiavi e comparatori
- Contenitori associativi riassunto
- pair
- Altri contenitori

Slide 8 2023 - Namespace,Eccezioni,Puntatori intelligenti,Funtori,Lambda

- Ambiti di visibilità
- Spazi dei nomi: namespace
- using
- Spazi dei nomi annidati
- Spazi dei nomi anonimi o senza nome
- Gestione degli errori
- Cosa sono le eccezioni
- Come utilizzare le eccezioni
- Flusso delle eccezioni
- Cattura dell'eccezione
- Come funzionano le eccezioni
- Lancio di una eccezione
- Quando usare le eccezioni
- Gestione delle eccezioni
- Cattura di più eccezioni
- Dove catturare un'eccezione
- noexcept
- Gestione delle risorse
- RAII / SBRM
- Puntatori intelligenti
- Puntatore grezzo e puntatore intelligente
- unique_ptr
- Puntatore tradizionale vs unique_ptr
- make_unique

- std::move
- unique_ptr e STL
- unique_ptr e nullptr
- shared_ptr
- make_shared
- shared_ptr e STL
- Problema con shared_ptr
- weak_ptr
- Logica come parametro di una funzione
- Puntatori di funzione (Callbacks)
- Polimorfismo e funzioni virtuali
- Operatore di chiamata di funzione
- Funtori
- Espressioni lambda
- Sintassi delle espressioni lambda

Slide 9 2023 - File Eseguibili e passaggi Codice → Eseguibile

- Dal codice ai programmi
- Dal codice ai programmi: pre-processamento (pre-processing)
- Dal codice ai programmi: compilazione (colpiling)
- Dal codice ai programmi: assemblaggio (assembling)
- Dal codice ai programmi: collegamento (linking)
- Collegamento statico e dinamico
- Formato dei file eseguibili
- Formato eseguibile e collegabile
- ELF: Sezioni principali

- Formato PE
- Registri X86
- Registri X86-64
- Gestione della memoria
- Allocazione della memoria
- Allineamento dei dati
- Segmenti di memoria
- Lo stack
- Struttura dello stack (per x86)
- Stack frame
- L'heap
- Funzioni di gestione della memoria
- Debug
- GDB: il debugger del progetto GNU

Slide 10 2023 - Programmazione GUI

- librerie GUI
- Qt
- wxWidgets
- xWidgets e CodeBlocks
- Esempio di Hello world
- Creazione tramite wizard
- Creazione tramite wizard e GUI Builder

Slide 11 2023 - Java

Slide 12 2023 - Python