Sommario

[Modellazioni: teoria per capirle 4](#_Toc140586471)

[Librerie 4](#_Toc140586472)

[Modificatori 4](#_Toc140586473)

[Best practices generali 4](#_Toc140586474)

[Costruttori 5](#_Toc140586475)

[Costruttori di default 5](#_Toc140586476)

[Lista di inizializzazione 5](#_Toc140586477)

[Tempo di vita delle variabili 6](#_Toc140586478)

[Distruttori 6](#_Toc140586479)

[\*this 6](#_Toc140586480)

[Costruttori di copia 6](#_Toc140586481)

[Costruttore di copia profonda 6](#_Toc140586482)

[Assegnazione profonda 7](#_Toc140586483)

[Distruzione profonda 7](#_Toc140586484)

[Scrittura metodi inline 8](#_Toc140586485)

[Recap puntatori 8](#_Toc140586486)

[Overloading operatori 9](#_Toc140586487)

[Operatore di assegnazione (=) 9](#_Toc140586488)

[Operatore di stampa (<<) 9](#_Toc140586489)

[Operatore di somma (+) 10](#_Toc140586490)

[Operatore di uguaglianza (==) 10](#_Toc140586491)

[Operatore di confronto minore o maggiore ( < o > ) 11](#_Toc140586492)

[Operatore di subscripting ([ ]) 11](#_Toc140586493)

[Operatori prefisso e postfisso (++) 11](#_Toc140586494)

[Operatore di selezione di membro (->) 12](#_Toc140586495)

[Operatore di dereferenziazione (\*) 13](#_Toc140586496)

[Classi container 13](#_Toc140586497)

[Iterator 13](#_Toc140586498)

[Const iterator 13](#_Toc140586499)

[Vector/List 14](#_Toc140586500)

[Classi eccezione 14](#_Toc140586501)

[Template 15](#_Toc140586502)

[Classi amiche – friend 15](#_Toc140586503)

[Operatore di stampa con template 16](#_Toc140586504)

[Ereditarietà e conversioni 18](#_Toc140586505)

[Virtual 18](#_Toc140586506)

[Polimorfismo 21](#_Toc140586507)

[Classe Astratta (Virtuale Pura) 22](#_Toc140586508)

[Classe concreta 22](#_Toc140586509)

[Costruttori virtuali 22](#_Toc140586510)

[Costruttori nelle classi derivate 22](#_Toc140586511)

[Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteCostruttore di copia nelle classi derivate 22](#_Toc140586512)

[Assegnazione nelle classi derivate 23](#_Toc140586513)

[Distruttori virtuali 23](#_Toc140586514)

[Distruttore per classi derivate 23](#_Toc140586515)

[RTTI 23](#_Toc140586516)

[Typeid 23](#_Toc140586517)

[Tipo polimorfo 23](#_Toc140586518)

[Classe polimorfa 23](#_Toc140586519)

[Dynamic cast 23](#_Toc140586520)

[Const cast 24](#_Toc140586521)

[Ereditarietà multipla 25](#_Toc140586522)

[Gerarchia classi I/O 26](#_Toc140586523)

[C++ 11 26](#_Toc140586524)

[C++ 17 27](#_Toc140586525)

[Ridefinire il comportamento come quello di default 28](#_Toc140586526)

[Esercizio da esame con i template 29](#_Toc140586527)

[Stampa di una parola precisa 30](#_Toc140586528)

[Gerarchie 31](#_Toc140586529)

[Cosa Stampa 33](#_Toc140586530)

[Sottotipi 37](#_Toc140586531)

[Funzioni 38](#_Toc140586532)

# Modellazioni: teoria per capirle

## Librerie

* Prima cosa da fare è importare <iostream> e using namespace std
  + Se non volete importare quest’ultimo, basta fare lo scoping (cioè, due volte due punti)
  + Una cosa del tipo std::vector pippo, std::string ciao
  + Le stringhe possono essere importate senza fare lo scoping usando semplicemente #include <string>
  + Il puntatore nullo o nullptr fa parte dello std; si può usare “0” per indicare nullptr, oppure std::nullptr
* Se serve usare le liste, occorre usare #include <list>
* Se serve usare i vettori, occorre usare #include <vector>
* Se serve usare l’uguaglianza di tipo dinamico, cioè il typeid, serve usare

#include <typeinfo>

* Per maggiori info sulla libreria I/O, consultare la sezione apposita

## Modificatori

* Se un oggetto ha un valore fisso (ad esempio, una tassa, una penalità, un tot mensile), tendenzialmente, la variabile è static
  + Esempio di utilizzo
    - Dentro la classe 🡪 unsigned tipo variabile
    - Fuori dalla classe 🡪 tipo Classe::variabile = valore
* Se non vengono fatte modifiche all’oggetto di invocazione, comunemente chiamati side effects, si marca l’oggetto, la variabile o la funzione intera come const
  + Banalmente, significa che alla variabile che ritornate o usate NON state facendo assegnazioni, somme, calcoli. L’oggetto NON viene toccato.
* Altrimenti usiamo:
  + public: visibilità pubblica
  + protected: visibilità alle sole sottoclassi
  + private: visibilità della singola classe
  + const: se la variabile è costante, quindi non cambia mai valore. Qui la intendo come modificatore, con significato diverso da quanto scritto sopra
    - In un metodo costante di una classe , il puntatore this è di tipo const C\*

## Best practices generali

* Dichiararla come classe e mettere il punto e virgola alla fine delle graffe
  + class Topolino
* Inserire i parametri come private
  + Se i parametri sono numerici, tendenzialmente, se sono dei numeri positivi, consigliato metterli unsigned int, poi dipende dal contesto. Altrimenti, vanno bene anche float oppure double
  + Potrebbe tranquillamente andare bene anche protected nel caso in cui volete che dei parametri siano visibili solo alle sottoclassi
* Inserire dei metodi di *get* per tutti i campi privati (mettendo const perché non vengono fatte modifiche all’oggetto di invocazione)

## Costruttori

L'ordine dei campi dati è determinato dall'ordine in cui appaiono nella definizione della classe C.

1. Per ogni campo di tipo non classe (ovvero tipo primitivo o derivato), viene allocato in memoria lo spazio per contenere un valore di tipo ma viene lasciato indefinito.

2. Per ogni campo di tipo classe viene invocato il costruttore di default di

3. Viene eseguito il codice del corpo del costruttore

Si ricordi che i costruttori agiscono nello stesso ordine di invocazione. Se le classi sono A, B e C,(assumendo che i costruttori siano ordinati in questo modo) i costruttori agiscono nell’ordine A, B e C.

## Costruttori di default

* Costruttore di default classico: permette di inizializzare gli oggetti; tendenzialmente, può essere omesso (viene chiamato quando il costruttore è assente
  + Esempio di utilizzo
    - nome\_classe();
* Costruttore di default ridefinito: permette di dare un valore di default ai parametri presenti (cioè, la classe ammette dei parametri ed occorre costruirli)
  + Esempio di utilizzo
    - La classe ha due valori int numero, double virgola
    - nome\_classe(int num, double virg) : numero(num), virgola(virg)

## Lista di inizializzazione

* Dopo la firma del metodo (cioè, nome e parametri) è definita come *lista di inizializzazione*, che si marca con i due punti. In essa vanno inizializzati i parametri correttamente prima dell’esecuzione.
  + Box(int i) : m\_width(i), m\_length(i), m\_height(i)
* Se i valori vengono inizializzati per almeno un parametro, necessariamente devono essere inizializzati tutti i successivi. Ciò viene fatto comunemente nelle classi “container”
  + nodo(const T& t = T(), nodo\* p=nullptr, nodo\* n=nullptr) : info(t), prev(p), next(n) {}

## Tempo di vita delle variabili

E’ l’intervallo di tempo in cui la variabile viene mantenuta in memoria durante l’esecuzione del programma. Possono esserci:

* Variabili di classe automatica: deallocate quanto termina il blocco di esecuzione
* Variabili di classe statica: eliminate solo a fine programma
* Variabili dinamiche: sono eliminabili solo con delete

## Distruttori

Viene invocato in particolare nei seguenti casi al termine di una funzione con questo ordine:

1. variabili locali funzione
2. oggetto anonimo ritornato come risultato della funzione non appena sia stato usato
3. parametri passati per valore alla funzione al suo termine

Occorre usare la tilda come segno per definire un distruttore:

* + Esempio di utilizzo
    - ~nome\_classe();

Si ricordi che i distruttori agiscono nell’ordine contrario rispetto alla loro costruzione. Se le classi sono A, B e C,(assumendo che i distruttori siano ordinati in questo modo) i distruttori agiscono nell’ordine C, B e A.

## \*this

Il puntatore this viene utilizzato in una classe per fare riferimento a sé stessa ed è un puntatore a un'istanza della sua classe, disponibile per tutte le funzioni membro non statiche. È spesso utile quando si restituisce un riferimento a sé stesso.

Viene normalmente usato nell’assegnazione profonda, per controllare che NON ci siano interferenze in memoria o il problema dell’aliasing. Utile sapere che c’è anche per capire i Cosa Stampa.

## Costruttori di copia

* Costruttore di copia standard: esso permette di inizializzare un oggetto dello stesso tipo classe e di dargli un valore come tipo
  + Esempio di utilizzo
    - nome\_classe(const nome\_classe& variabile);
  + Riferimento costante perché NON si ha condivisione di memoria
  + Esso viene invocato automaticamente
    1. se oggetto viene dichiarato e inizializzato con oggetto della stessa classe
    2. se oggetto passato per valore come parametro attuale in una funzione
    3. quando una funzione ritorna tramite return un oggetto

## Costruttore di copia profonda

Copio TUTTI i campi dell’oggetto di invocazione.

Sintassi 🡪 nome\_classe(const nome\_classe& parametro){}

* + Se il campo è un puntatore o un vettore, tendenzialmente devo scorrerlo con un ciclo oppure con una ricorsione e copiarmi TUTTO ciò che ha.
    - Normalmente, qui controllo se tutti i campi esistono e li costruisco
      * Esempio di utilizzo con controlli estesi
        + Vettore(const T& x=T()){

if(\_size == 0) a = nullptr;

else a = new T[\_size];

for(int i=0; i<\_size; ++i) a[i]=x; }

* + - * Esempio di utilizzo con controlli compatti e assegnazioni in linea
        + Vettore(const T& x=T()): a(\_size == 0 ? nullptr : new T[\_size]) {

for(int i=0; i<\_size; ++i) a[i]=x; }

Di seguito le strade comuni alternative.

* + Per copiare gli oggetti di una classe (di solito, dei puntatori) di solito utilizzo il metodo di clonazione polimorfa clone()
    - Esempio di utilizzo
      * SmartP(const SmartP& s):p(s.p->clone()) {}
  + Posso anche definire un metodo esterno copia per scorrere tutti i campi che ho e copiare tutto l’oggetto
    - Esempio di utilizzo
      * C(const C& b):campo(copia(b.campo)){}
    - E il metodo copia
      * nodo\* bolletta :: copia(nodo\* p){

if(!p) return 0;

else return new nodo(p->info, copia(p->next);

}

## Assegnazione profonda

Sintassi 🡪 nome\_classe operator=(const nome\_classe& parametro){}

Data l’assegnazione ; dove è un oggetto di qualche classe . Supponiamo che l’assegnazione sia ridefinita internamente alla classe come metodo, perciò:

* Il primo operando viene assunto come oggetto di invocazione dell’assegnazione;
* Il secondo operando invece è un parametro formale di tipo . Questo parametro sarà dichiarato costante e per riferimento in modo da evitare che venga effettuata inutilmente una copia del secondo operando

Perciò ; restituisce (che permette di usare anche ;

## Distruzione profonda

Cancella TUTTI i campi presenti nell’oggetto di invocazione.

Sintassi 🡪 ~nome\_classe(){}

È possibile ridefinire il distruttore in modo che esso effettui una distruzione profonda degli oggetti, deallocando anche la memoria a cui puntano i campi puntatore, in modo che, se un oggetto x include un campo dati p che punta alla testa di una lista, con la distruzione profonda di x si deallocherà tutta la lista puntata da p.

Normalmente quello che si fa è cancellare tutto un contenitore o un puntatore:

* Esempio di utilizzo
  + ~Vettore() { delete[] a;}
* Come per sopra, può essere utile definire un metodo ausiliario che distrugge iterativamente o ricorsivamente i campi dell’oggetto
  + Esempio di utilizzo
    - bolletta :: ~bolletta(){ distruggi(first); //first punta alla testa della lista }
  + E il metodo distruggi
    - void bolletta :: distruggi(nodo\* p){ if(p){ distruggi(p->next); delete p; } }

## Scrittura metodi inline

Occorre fare una distinzione importante, a livello di compilazione e scrittura dei metodi.

Normalmente, negli esami, tutto viene scritto inline, cioè viene dichiarata la firma del metodo (cioè, nome e parametri) e poi viene scritto subito il corpo del metodo.

Altrimenti, per separazione del codice e suo migliore ordine:

* Crearsi il file header (.h) contenente le definizioni e il main
* Crearsi il file.cpp contenente il corpo di tutti i metodi

## Recap puntatori

* Dichiaro l’array vuoto
  + int\* array;
* Dichiaro l’array con X elementi
  + int\* array = new int[50];
* Accedo direttamente ad un membro della classe SENZA usare puntatori (operatore punto od operatore di selezione di membro)
  + class Base{int var1; ..} Base b; b.var1;
* Accedo direttamente ad un membro della classe CON l’uso di puntatori (operatore freccia od operatore di accesso ad un membro di classe)
  + class Student{int marks; .. } Student \*s; .. s->marks;

## Overloading operatori

Posso sovraccaricare un operatore come metodo oppure come funzione esterna.

1. Non si possono cambiare

* posizione (prefissa/infissa/postfissa)
* numero operandi
* precedenze e associatività

2. Tra gli argomenti deve essere presente almeno un tipo definito da utente

3. Gli operatori "=", "[]" e "->" si possono sovraccaricare solo come metodi interni

4. Non si possono sovraccaricare gli operatori ".", "::", "sizeof", "typeid", i cast e "? :"

5. Gli operatori "=", "&" e "," hanno una versione standard (assegnazione, address of e comma)

### Operatore di assegnazione (=)

Passi logici:

1. controllo che non ci siano interferenze in memoria
2. pulisco la memoria da quanto presente in precedenza
3. assegno tutti i campi
4. ritorno \*this

C& operator=(const C& b){

if(this != &b){

this.campo=copia(b.campo); //copio tutti i campi

}

return \*this;

}

### Operatore di stampa (<<)

Normalmente, viene sempre dichiarata come funzione *esterna* alla classe in oggetto (per evitare di avere errori di ogni tipo).

Se non voglio utilizzare metodi pubblici per leggere parti private della classe e darle in output devo dichiarare una *relazione di amicizia* (friend) tra il prototipo dell'output e la classe.

Sintassi 🡪 ostream& operator<<(ostream& os, const C& c)

* Esempio di utilizzo esteso (per maggiori dettagli con i template, consultare la sezione apposita)

class Date

{

int mo, da, yr;

public:

Date(int m, int d, int y)

{mo = m; da = d; yr = y;}

friend ostream& operator<<(ostream& os, const Date& dt);

};

ostream& operator<<(ostream& os, const Date& dt)

{

os << dt.mo << '/' << dt.da << '/' << dt.yr;

return os;

}

int main()

{ Date dt(5, 6, 92);cout << dt; }

### Operatore di somma (+)

L’operatore permette di sommare i componenti della classe; normalmente, si passa un riferimento e si eseguono le operazioni sulle variabili singole.

Sintassi 🡪 nome\_classe operator+(const nome\_classe & n);

* Esempio esteso

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Le analoghe considerazioni valgono per l’operatore meno (-), che andrà sostituito e valgono i ragionamenti all’opposto.

### Operatore di uguaglianza (==)

Questo è un operatore di confronto, come tale ritorna un booleano.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

Le analoghe considerazioni valgono per l’operatore di disuguaglianza (!=), che andrà sostituito e valgono i ragionamenti all’opposto.

### Operatore di confronto minore o maggiore ( < o > )

Questo viene usato tendenzialmente come operatore di confronto lessicografico, cioè confrontiamo tutti i caratteri delle stringhe e stabiliamo in base alle posizione dei loro caratteri e sui caratteri stessi se una parola viene prima dell’altra, come nel dizionario.

Rimane un operatore di confronto e ritorna un booleano.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

### Operatore di subscripting ([ ])

Permette di accedere ad un preciso membro di un array. Tendenzialmente gli viene passato un intero oppure un iteratore per determinare la posizione di un elemento preciso.

Immagine che contiene testo, Carattere, schermata, design

Descrizione generata automaticamente

Con iteratore:

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

### Operatori prefisso e postfisso (++)

Gli operatori di incremento e decremento rientrano in una categoria speciale perché esistono due varianti di ciascuno:

* Preincremento e postincremento
* Predecremento e postdecremento

Quando si scrivono funzioni di operatori sovraccaricati, può essere utile implementare versioni separate per le versioni prefix e postfix di questi operatori. Per distinguere le due versioni, si osserva la seguente regola:

* La forma prefisso dell'operatore è dichiarata esattamente come qualsiasi altro operatore unario; la forma postfisso accetta un argomento aggiuntivo di tipo int.

Esempio esteso:

// increment\_and\_decrement1.cpp

class Point

{

public:

// Declare prefix and postfix increment operators.

Point& operator++(); // Prefix increment operator.

Point operator++(int); // Postfix increment operator.

// Declare prefix and postfix decrement operators.

Point& operator--(); // Prefix decrement operator.

Point operator--(int); // Postfix decrement operator.

// Define default constructor.

Point() { \_x = \_y = 0; }

// Define accessor functions.

int x() { return \_x; }

int y() { return \_y; }

private:

int \_x, \_y;

};

// Define prefix increment operator.

Point& Point::operator++()

{

\_x++;

\_y++;

return \*this;

}

// Define postfix increment operator.

Point Point::operator++(int)

{

Point temp = \*this;

++\*this;

return temp;

}

// Define prefix decrement operator.

Point& Point::operator--()

{

\_x--;

\_y--;

return \*this;

}

// Define postfix decrement operator.

Point Point::operator--(int)

{

Point temp = \*this;

--\*this;

return temp;

}

int main()

{

}

### Operatore di selezione di membro (->)

Ritorna il riferimento del valore dell’oggetto puntato.



### Operatore di dereferenziazione (\*)

Ritorna il valore dell’oggetto puntato.



## Classi container

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

Un buonissimo esempio di queste classi sono gli esercizi *SmartP, Coda, Array* e similari.

In tutti i casi, sono NECESSARIE le seguenti classi *annidate* (dentro la classe container).

## Iterator

Si usa quando serve accedere agli L-valori del contenitore (i loro valori)

Le classi di tipo iteratore ci si aspetta che:

1. Dichiarino un oggetto per scorrere il contenitore e una variabile per capire se si ha passato la fine dell’array (past-the-end)
2. Costruttore
3. Metodi di accesso all’inizio e alla fine dell’iteratore (begin – end)
4. Overloading operatori incremento e decremento postfisso
5. Overloading operatori \* e ->
6. A volte anche l’operatore []

## Const iterator

Si usa quando serve accedere agli R-valori del contenitore (riferimenti costanti; quindi, tipo, il metodo che stiamo usando non fa side effects e possiamo usare il const iterator).

Le classi di tipo iteratore costante ci si aspetta che:

1. Dichiarino un oggetto per scorrere il contenitore e una variabile per capire se si ha passato la fine dell’array (past-the-end)
2. Costruttore
3. Metodi di accesso all’inizio e alla fine dell’iteratore (begin – end)
4. Overloading operatori incremento e decremento postfisso
5. Overloading operatori \* e ->
6. A volte anche l’operatore []

## Vector/List

Sono i due tipi di container da noi sempre usati in fase d’esame, specie il primo perché più efficiente.

Occorre dichiarare #include<vector> per usarlo (parallelamente, #include<list>).

Di questo si ricordi che:

* Per cancellare un elemento in una singola posizione, viene usato il metodo *erase*
  + c.erase(it);
* Per cancellare tutti gli elementi, si usa il metodo *clear*
  + c.clear();
* Per inserire un qualsiasi elemento, di solito, usiamo *push\_back*
  + c.push\_back(\*it);

Esistono vari modi di scrivere i cicli:

* Uso senza auto e con gli iteratori
  + for(list<const GalloFile\*>::const\_iterator cit = g.begin(); cit != g.end(); ++cit)
* Uso con auto e con gli iteratori
  + for(auto it = g.begin(); it != g.end(); ++it)

## Classi eccezione

Per queste classi, è easy; basta un costruttore possibilmente con una stringa.

class Eccezione{

private:

string errore;

public:

Anomalia(string e) : errore(e){}

};

Poi chiamata con:

throw Eccezione(“Errore”);

## Template

È la descrizione di un modello che il compilatore può usare per generare automaticamente istanze particolari di una classe che differiscono per il tipo di alcuni membri propri della classe.

Deve esserne definito un tipo; normalmente, si può anche usare *typename* per indicare un tipo generico. La dichiarazione del tipo precede ogni dichiarazione della classe parte di template oppure di un metodo che contiene un tipo template. Si consideri che T è un tipo generico, utilizzato per indicare una sorta di placeholder e dire “può essere qualsiasi tipo”.

La dichiarazione più generica è:

Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamente

I template possono avere dei valori di default come segue:

Immagine che contiene testo, Carattere, bianco, calligrafia

Descrizione generata automaticamente

### Classi amiche – friend

In C++ una funzione definita friend di una classe ha la possibilità di accesso alla parte privata e protetta della classe di cui è stata dichiarata amica. La regola è usarle se proprio ne si ha bisogno e solo se proprio non si possa farne a meno.

Esistono vari tipi di amicizie:

* Friend non template (la classe B, la funzione test() e il metodo A::fun() sono friend di tutte le istanze del template di classe C)

Immagine che contiene testo, Carattere, schermata, calligrafia

Descrizione generata automaticamente

* Friend associato (Se avessi bisogno di accedere ad un campo privato con tipo diverso da quello dell'istanziazione allora dovrei dichiarare un'amicizia NON associata)

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

* Friend non associato

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

L’amicizia NON viene ereditata, NON è simmetrica e NON è transitiva.

### Operatore di stampa con template

Se non voglio utilizzare metodi pubblici per leggere parti private della classe e darle in output devo dichiarare una *relazione di amicizia* (friend) tra il prototipo dell'output e la classe.

I tre approcci si differenziano per chi si dichiara amico della funzione e per come la si implementa.

1. Dichiarare tutte le istanze del template come friend

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

Con questo approccio si apre inutilmente la propria particolare istanziazione D<T>, dichiarando amiche tutte le istanziazioni di operator<<. Cioè, std::ostream& operator<( std::ostream &, const D<int>& ) ha accesso a tutti gli interni di D<double>.

* Dichiarare solo una particolare istanziazione dell'operatore di inserimento come amico.

D<int> può gradire l'operatore di inserimento quando applicato a sé stesso, ma non vuole avere nulla a che fare con std::ostream& operator<( std::ostream&, const D<double>& ).

Questo può essere fatto in due modi: il modo più semplice è quello, che consiste nell'inlining dell'operatore, una buona idea anche per altre ragioni:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

In questa prima versione, non si crea un operatore<< templatizzato, ma piuttosto una funzione non templatizzata per ogni istanziazione del template Test. Anche in questo caso, la differenza è sottile, ma è sostanzialmente equivalente ad aggiungere manualmente: std::ostream& operator<( std::ostream&, const Test<int>& ) quando si istanzia Test<int>, e un altro overloading simile quando si istanzia Test con double, o con qualsiasi altro tipo.

* Senza inlining del codice e con l'uso di un modello

Qui è possibile dichiarare una singola istanziazione del modello come amico della propria classe, senza aprirsi a tutte le altre istanziazioni:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

## Ereditarietà e conversioni

* Normalmente, l’ereditarietà è pubblica, quindi
  + Class Pippo: public Topolino
* Se la classe “non deve permettere la costruzione pubblica dei suoi sottooggetti” occorre usare protected

Parole chiave:

* Sottotipo 🡪 significa che il tipo di X:
  + o è uguale al tipo di Y;
  + oppure che X è un sottotipo diretto o indiretto di Y.
* Sottotipo proprio 🡪 significa che X è un sottotipo di Y diverso da Y stesso.
* Sottooggetto: oggetto della classe derivata

Distinzione chiave per i cosa stampa:

* Tipo statico: quello a sinistra della dichiarazione, quindi il tipo determinato dal compilatore che è staticamente derivabile dal codice sorgente
* Tipo dinamico: il tipo determinato a runtime, tendenzialmente il tipo a destra dell’uguale

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

Si noti che e sono tipo statico e dinamico qui.

Se avessimo

allora avremmo D come tipo statico ed E come tipo dinamico.

Concretamente significa che, se vengono chiamati metodi virtuali o polimorfi, preferiamo se possibile E, altrimenti se non viene sovrascritto, preferiamo il tipo statico D.

### Virtual

Questa parola chiave permette di determinare quale metodo chiamare a tempo di esecuzione; se un metodo è definito con la parola chiave *virtual* allora significa che si controlla il tipo dinamico del puntatore; se questo tipo ha un metodo che è scritto nello stesso identico modo, allora, essendo dichiarato *virtual* viene preferito quello, secondo l’overriding.

L'overriding di un metodo è l'abilità di una sottoclasse di implementare diversamente un metodo già dichiarato ed implementato precedentemente in una sua super-classe. Questa nuova implementazione ha lo stesso nome, parametri e tipo di ritorno (const incluso) del metodo nella super-classe. La versione del metodo che verrà eseguita dipenderà dall'oggetto che la invoca. Il tipo di ritorno al più può essere un sotto-oggetto del tipo di ritorno precedente.

Se viene modificata la lista dei parametri allora si fa una ridefinizione non un overriding.

#include <iostream>

**using** **namespace** std;

**class** base {

**public**:

**virtual** **void** print() { cout << "print base class\n"; }

**void** show() { cout << "show base class\n"; }

};

**class** derived : **public** base {

**public**:

**void** print() { cout << "print derived class\n"; }

**void** show() { cout << "show derived class\n"; }

};

**int** main()

{

    base\* bptr;

    derived d;

    bptr = &d;

    // Virtual function, binded at runtime

    bptr->print();

    // Non-virtual function, binded at compile time

    bptr->show();

**return** 0;

}

L’output è così definito:

Immagine che contiene testo, Carattere, schermata, Elementi grafici

Descrizione generata automaticamente

Si ha overriding della funzione *print*, in quanto ridefinita correttamente con stessa firma dalla classe derivata; si noti che, anche se non c’è il virtual davanti, dato che la stessa segnatura, il tipo dinamico determina l’esecuzione della sottostante.

Termini chiave:

* static binding/early binding/binding statico, avviene a compile time e vuol dire che tutte le informazioni rispetto ad una classe sono già conosciute a compile time e sa già con certezza quale tipo andare ad invocare per certo

Si consideri il codice seguente, in cui la funzione sum() è sovraccaricata per accettare due e tre argomenti interi. Anche se esistono due funzioni con lo stesso nome all'interno della classe ComputeSum, la chiamata di funzione sum() si lega alla funzione corretta a seconda dei parametri passati a tali funzioni. Questo binding viene effettuato staticamente in fase di compilazione.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // C++ program to illustrate the concept of static binding  #include <iostream>  using namespace std;    class ComputeSum  {      public:        int sum(int x, int y) {          return x + y;      }        int sum(int x, int y, int z) {          return x + y + z;      }  };    int main()  {      ComputeSum obj;      cout << "Sum is " << obj.sum(10, 20) << endl;      cout << "Sum is " << obj.sum(10, 20, 30) << endl;        return 0;  } |

**Output:**  
  
Sum is 30  
Sum is 60

* dynamic binding/late binding/binding ritardato, solitamente realizzata ed ottenuta tramite puntatori e/o riferimenti a runtime, che significa che a seconda del contesto di invocazione e della ridefinizione dei metodi in quell’esatto momento (quindi ritarda fino a tempo di compilazione), decide cosa è meglio chiamare. Ciascun riferimento virtuale viene mantenuto all’interno della cosiddetta vtable, tenendo corrispondenza di ognuno di questi

Consideriamo il codice seguente, in cui abbiamo una classe base B e una classe derivata D. La classe base B ha una funzione virtuale f(), che viene sovrascritta da una funzione della classe derivata D, cioè D::f() sovrascrive B::f().

Consideriamo ora le righe 30-34, dove la decisione su quale funzione della classe sarà invocata dipende dal tipo dinamico dell'oggetto puntato da basePtr. Questa informazione può essere disponibile solo in fase di esecuzione e quindi f() è soggetta al binding dinamico.

// C++ program to illustrate the concept of dynamic binding

#include <iostream>

using namespace std;

class B

{

    public:

    // Virtual function

    virtual void f() {

        cout << "The base class function is called.\n";

    }

};

class D: public B

{

    public:

    void f() {

        cout << "The derived class function is called.\n";

    }

};

int main()

{

    B base;

    D derived;

    B \*basePtr = &base;

    basePtr->f();

    basePtr = &derived;

    basePtr->f();

    return 0;

}

**Output:**  
  
The base class function is called.  
The derived class function is called.

### Polimorfismo

Il fatto di poter chiamare un metodo al posto di un altro in fase di esecuzione fa in modo che l’esecuzione di un codice possa assumere *più forme*, da cui la parola polimorfismo.

Ciò avviene tramite l’uso citato di tipi dinamici, dei virtual e dei successivi.

Si consideri inoltre che ogni tanto si parla di puntatori superpolimorfi; questi non sono altro che delle sottoclassi di altre sottoclassi, che in ogni momento di invocazione possono assumere più forme a seconda delle chiamate.

### Classe Astratta (Virtuale Pura)

Una classe astratta (o virtuale pura) è una classe che non può essere istanziata in quanto è marcata come astratta oppure specifica metodi astratti (o virtuali).

Come caratteristiche:

1. Almeno un metodo virtuale puro (dichiarazione virtual con = 0 prima del ;).

2. Non si possono costruire oggetti.

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

### Classe concreta

Una classe concreta è una classe che può essere istanziata.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

## Costruttori virtuali

Si noti che il nome della sezione descrive LE REGOLE che vengono usate in fase di costruzione degli oggetti con l’ereditarietà e per richiamare l’attenzione; MAI andare a definire un costruttore come virtual.

### Costruttori nelle classi derivate

### Immagine che contiene testo, schermata, Carattere Descrizione generata automaticamenteCostruttore di copia nelle classi derivate

Immagine che contiene testo, schermata, Carattere, informazione

Descrizione generata automaticamente

### Assegnazione nelle classi derivate

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

## Distruttori virtuali

Questi permettono di distruggere gli oggetti di una classe e anche i suoi sottooggetti (quelli delle sottoclassi)

### Distruttore per classi derivate

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

## RTTI

Il C++ permette di determinare il tipo dinamico di un puntatore o riferimento a tempo di esecuzione tramite gli operatori typeid e dynamic\_cast.

### Typeid

L’operatore typeid, definito nella libreria *typeinfo*, permette di determinare il tipo di una qualsiasi espressione a tempo di esecuzione. Se l’espressione è un riferimento o un puntatore polimorfo dereferenziato, allora typeid ritornerà il tipo dinamico di questa espressione.

Permette di determinare se “il tipo dinamico di un’espressione è esattamente uguale al tipo dinamico di un’altra espressione); utilizzo frequente con:

if(typeid(\*p) == typeid(q)) cout << “It works”;

* Se la classe non contiene metodi virtuali allora typeid restituisce il tipo statico del riferimento o del puntatore de-referenziato.
* I puntatori devono essere de-referenziati altrimenti RTTI torna il tipo statico.

### Tipo polimorfo

Un tipo T è polimorfo se è un tipo classe che include almeno un metodo virtuale.

### Classe polimorfa

* Se voglio creare una classe base astratta senza definire inutilmente un metodo virtuale puro che non mi serve, posso definire un distruttore virtuale puro
* Se voglio rendere una classe polimorfa, basta aggiungere alla classe base della gerarchia un distruttore virtuale (mettendo semplicemente virtual)

### Dynamic cast

Per consentire la conversione dinamica da un tipo ad un altro sullo stesso livello di gerarchia oppure verso un tipo ad un livello più basso, occorre usare dynamic\_cast

* tipo\_in\_cui\_convertire variabile =

< tipo\_in\_cui\_convertire\*>(&oggetto\_da\_convertire)

* Esempio di utilizzo
  + PushButton\* pb= dynamic\_cast<PushButton\*>(ab);

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

Se il cast non va a buon fine, la libreria std prevede il bad\_cast come tipo errore per la conversione dinamica. Lo si ricordi per gli esercizi.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Normalmente dynamic\_cast viene usato per effettuare il downcasting (quasi sempre), AL MASSIMO per conversioni allo stesso livello di gerarchia.

* Usare il downcasting solo quando necessario.
* Non fare type checking dinamico inutile (molto dispendioso e causa overhead)
* Usare metodi virtuali nelle classi base al posto del type checking dinamico se possibile.

### Const cast

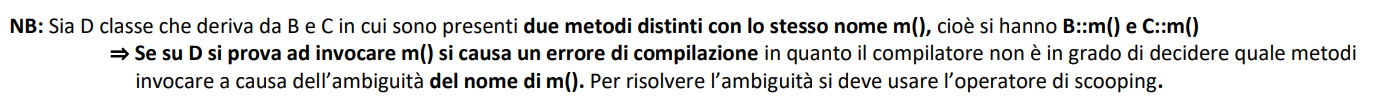
* Se l’oggetto è di tipo costante e occorre togliere il const, occorre usare const\_cast
  + Esempio di utilizzo
    - AbstractButton\* ab = const\_cast<AbstractButton\*>(\*it);
    - In questo caso, la variabile è di tipo AbstractButton non costante
* Se invece occorre fare una conversione di tipo dinamico e poi togliere il costante (di solito si fa perché ci viene data come parametro una variabile marcata const), allora si può usare in coppia con il dynamic\_cast
  + Esempio di utilizzo
    - fstream \*f=dynamic\_cast<fstream\*>(const\_cast<ios\*>(\*cit));

## Ereditarietà multipla

Una classe può essere derivata da più di una classe base, in questo caso ogni oggetto della classe derivata contiene un sottooggetto per ognuna delle classi base da cui deriva.

L’ordine di ereditarietà parte da destra e va verso sinistra.

Spesso è causa di problemi di ambiguità, come nel caso si abbia una classe che eredita due metodi con almeno lo stesso nome dalle sue classi padre, nel momento in cui si chiama il metodo nella classe derivata in compilazione viene dato un errore dovuto ad ambiguità.



Una gerarchia di classi può essere molto complessa, può capitare che una classe derivi da due classi base le quali a loro volta derivano, direttamente o indirettamente, da una stessa classe. In questo caso si parla di ereditarietà a diamante, che produce ambiguità e spreco di memoria.

Immagine che contiene testo, linea, diagramma, Carattere

Descrizione generata automaticamente

Risolvo usando la derivazione virtuale; dichiaro virtuale la derivazione sulle due classi che derivano dalla punta del diamante in questo modo posso avere un unico sotto-oggetto della punta in ogni oggetto della classe che chiude il diamante.

class A {

...

};

class B: virtual public A {

...

};

class C: virtual public A {

...

};

class D: public B, public C { //ho un puntatore al singolo sottooggetto nella classe derivata D

...

};

# Gerarchia classi I/O

Immagine che contiene schermata, diagramma, testo, linea

Descrizione generata automaticamente

Concretamente parlando:

* La classe <ios> è compresa in #include <iostream>
* Si noti che <istream> ed <ostream> sono già presenti in <iostream>
* Si noti che per usare <ifstream> ed <ofstream> occorre includere <fstream>
* Si noti che per usare <istringstream> ed <ostringstream> occorre includere <stringstream>

# C++ 11

Parole chiave VERAMENTE utili:

* *auto* per capire automaticamente il contesto della variabile
* *default* per avere disponibile la versione standard del costruttore o distruttore di default

A() = default;

virtual ~A() = default;

* *override* che specifica precisamente che si sta compiendo un overriding (virtuale con stessa firma identica) di un certo metodo e per ricordarselo

virtual void m(int) override {}

* *final*, che descrive l’ultimo *overriding* compiuto su un metodo e ottimizza la compilazione determinando che la virtualizzazione termina con una certa classe.

virtual void m(int) final {}

# C++ 17

Parole chiave VERAMENTE utili:

* *for\_each* usando la libreria *algorithm* per fare dei for in maniera seria

std::for\_each(i.begin(), i.end(), [] (int x) {cout << x << ",";});

* i cicli *for range:based*

for(auto i : vettore) {

# Ridefinire il comportamento come quello di default

* Operatore di assegnazione

In questo caso, semplicemente, andiamo ad utilizzare anche l’operatore di assegnazione delle sottoclassi dirette, assieme ai propri campi.

Immagine che contiene testo, schermata, schermo, Carattere

Descrizione generata automaticamente

* Costruttore di copia:

#include <iostream>

#include <vector>

using namespace std;

class A{

private:

    virtual void f() const = 0;

    vector<int\*>\* ptr;

};

class D: virtual public A{

private:

    int z;

    double w;

};

class E: public D{

private:

    vector<double\*> v;

    int \*p;

    int& ref;

public:

    void f() const {}

    E(): D(), v(), p(), ref() {}

    E(E e): D(e), v(e.v), p(e.p), ref(e.ref) {}

    //ridefinizione del costruttore di copia di E

    /\*

    Si considerino le precedenti definizioni. Ridefinire (senza usare

    la keyword "default") nello spazio sottostante il costruttore di copia della classe E

    in modo tale che il suo comportamento coincida con quello del costruttore di copia standard di E.

    \*/

    E(const E& e): D(e), v(e.v), p(e.p), ref(e.ref) {}

};

In questo caso, semplicemente, andiamo ad utilizzare anche il costruttore di copia delle sottoclassi dirette, assieme ai propri campi.

# Esercizio da esame con i template

In questo tipo di esercizio, è fondamentale capire quale delle due classi è amica e corrisponde allo stesso tipo listato; non c’è niente di più. L’esempio cardine di questa tipologia è il seguente.

Immagine che contiene testo, schermata, cartone animato, design

Descrizione generata automaticamente

# Stampa di una parola precisa

Questo tipo di esercizio è affrontabile andando PIANO; un esempio è andare a vedere tutte le relazioni di ereditarietà del caso e quindi capire tutto quello che succede nel programma con calma.

Esempio modello:

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

Immagine che contiene testo, schermata, diagramma

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, tipografia

Descrizione generata automaticamente

# Gerarchie

Qui siamo sulla parte easy; studiare bene i termini relativi alla parte polimorfa e capire bene come costruire le classi e le loro gerarchie. Spesso, infatti, si tratta di trabocchetti verbali (es. costruzione pubblica dei sottooggetti, ridefinire comportamenti come quelli standard, concreto, astratto). Nel file c’è già tutto.

Esempio calco:

Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamente

class A{

public:

virtual ~A()

};

class B : virtual public A {

public:

virtual void f() const = 0;

};

class C : public B {

public:

void f() const {}

};

class D : virtual public A {

protected:

D();

}

class E : public C, public D {

E& operator=(const E& e){

C::operator=(e);

D::operator=(e);

return \*this;

}

}

# Cosa Stampa

Un esercizio super frequente da esame; di seguito alcuni consigli per affrontarli bene.

* Occhio ai const, specie nelle stampe che presentano un ritorno di uno \*this; in quel caso, spesso, non compila
* Occhio agli errori a runtime, provocati generalmente da una stampa ripetuta (tipo, in un metodo, va in loop chiamando sempre m(), per dirne una); il prof li definisce “scherzetti”

Un esempio tra i tanti presenti:

#include <iostream>

using namespace std;

class B {

public:

B() {cout<<"B() ";}

virtual ~B() {cout<<"~B() ";}

virtual void f() {cout << "B::f "; g(); j();}

virtual void g() const {cout <<"B::g ";}

virtual const B\* j() {cout<<"B::j " ; return this;}

virtual void k() {cout<<"B::k "; j(); m(); }

void m() {cout <<"B::m "; g(); j();}

virtual B& n() { cout<<"B::n "; return \*this;}

};

class C: virtual public B {

public:

C() {cout<<"C() ";}

~C() {cout<<"~C() ";}

virtual void g() const override {cout << "C::g ";}

virtual void m() {cout <<"C::m "; g(); j();}

void k() override {cout <<"C::k "; B::n();}

B& n() override {cout <<"C::n "; return \*this; }

};

class D: virtual public B {

public:

D() {cout<<"D() ";}

~D(){ cout<<"~D() ";}

virtual void g(){cout <<"D::g ";}

const B\* j() {cout <<"D::j "; return this;}

void k() const {cout <<"D::k "; k();}

void m() {cout<<"D::m "; g(); j(); }

};

class E: public C, public D {

public:

E() {cout<<"E() ";}

~E(){cout<<"~E() ";}

virtual void g() const{cout << "E::g ";}

const E\* j(){cout<<"E::j "; return this;}

void m() {cout <<"E::m "; g(); j();}

D& n() final{cout <<"E::n "; return \*this;}

};

class F: public E{

public:

F() {cout<<"F() ";}

~F() {cout<<"~F() ";}

F (const F& x): B(x) {cout<<"Fc ";}

void k() {cout<<"F::k "; g();}

void m() {cout<<"F::m "; j();}

};

int main() {

B \*p1 = new E();

B \*p2 = new C();

B \*p3 = new D();

C \*p4 = new E();

const B \*p5 = new D();

const B \*p6=new E();

const B \*p7=new F();

F f;

cout<<endl;

cout<<"STAMPE"<<endl;

F x;cout<<endl;

//stampa B() C() D() E() F()

C \*p=new F(f);cout<<endl;

//stampa C() D() E() Fc

p1->f();cout<<endl;

//stampa B::f E::g E::j

p1->m();cout<<endl;

//stampa B::m E::g E::j

//(p1->j())->k();

//NON COMPILA "error: passing ‘const B’ as ‘this’ argument discards qualifiers"

//(dynamic\_cast<const F\*>(p1->j()))->g(); //RUNTIME ERROR

p2->f();cout<<endl;

//stampa B::f C::g B::j

p2->m();cout<<endl;

//stampa B::m C::g

(p2->j())->g();cout<<endl;

//stampa B::j C::g

p3->f();cout<<endl;

//stampa B::f B::g D::j

p3->k();cout<<endl;

//stampa B::k D::j B::m B::g D::j

(p3->n()).m();cout<<endl;

//B::n B::m B::g D::j

(dynamic\_cast<D&>(p3->n())).g();cout<<endl;

//stampa B::n D::g

p4->f();cout<<endl;

//stampa B::f E::g E::j

p4->k();cout<<endl;

//stampa C::k B::n

(p4->n()).m();cout<<endl;

//stampa E::n B::m E::g E::j

//(p5->n()).g();cout<<endl; //NON COMPILA "error: passing ‘const B’ as ‘this’ argument discards qualifiers"

//(dynamic\_cast<E\*>(p6))->j();cout<<endl; //NON COMPILA "cannot dynamic\_cast ‘p6’ (of type ‘const class B\*’) to type ‘class E\*’"

(dynamic\_cast<C\*>(const\_cast<B\*>(p7)))->k();cout<<endl; //stampa F::k E::g

delete p7;cout<<endl;

//stampa ~F() ~E() ~D() ~C() ~B()

}

# Sottotipi

Ogni tanto compaiono questi esercizi; sono molto simili alla stampa di una parola precisa sostituendo le classi che permettono la stampa di una precisa lettera. Letteralmente, si segua il flusso di esecuzione andando PIANO e tendenzialmente partire dalle deduzioni ovvie (se va male un cast di un tipo, di certo non è suo sottotipo, etc.).

Esercizio calco:

Immagine che contiene testo, schermata, Parallelo, diagramma

Descrizione generata automaticamente

# Funzioni

Questo tipo di esercizio racchiude funzioni legate alla classe *ios* oppure alla famiglia di classi della libreria Qt. Per farli giusti basta letteralmente leggere le richieste e con calma svolgerne ciascuna. DI fatto, la prima parte spesso è composta da una descrizione ad alto livello di metodi di libreria e poi la definizione della funzione che vi viene chiesta.

Esempi con *ios*:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Esempio calco basato su Qt:

Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente

#include <iostream>

#include <vector>

#include <list>

class QWidget{

public:

virtual ~QWidget();

};

class QLabel{

public:

void setLineWidth(int);

void setWordWrap(bool);

};

class QLCDNumber{

public:

void setDigitCount(int);

};

class QFrame{

public:

int heightDefault();

};

class QSplitter : public QFrame{

public:

QSplitter();

};

using namespace std;

list<QFrame\*> fun(vector<QWidget\*>& v){

list<QFrame\*> ret;

for(vector<QWidget\*>::iterator it=v.begin(); it!=v.end(); it++){

QLabel\* l = dynamic\_cast<QLabel\*>(\*it);

if(l){

l->setLineWidth(8);

l->setWordWrap(false);

}

QLCDNumber\* n = dynamic\_cast<QLCDNumber\*>(\*it);

if(n){

n->setDigitCount(3);

}

QFrame\* f = dynamic\_cast<QFrame\*>(\*it);

if(f && !dynamic\_cast<QSplitter\*>(f) && f->heightDefault()<10)

ret.push\_back(f);

}

return ret;

}