Capitolo 2

Assegnazione standard: C& operator=(const C&); per istruzione C a; a = b;

Costruttore di copia standard: C(const C&); 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

Comportamento di un costruttore: class C{ dato x1,...,xn C(Tipo1,...,Tipon){...}

1. per ogni campo dati xi di tipo Ti non classe (ossia tipo primitivo o derivato) viene allocato uno spazio di memoria per ottenere valore Ti e viene lasciato indefinito valore
2. per ogni campo dati xi di tipo classe Ti viene costruito mediante una invocazione del costruttore di default Ti( )
3. infine viene eseguito il corpo del costruttore

Lista inizializzazione di un costruttore: class C{ x1,...xk C(T1,...Tn):xi1(...),...xij(...){...}

1. ordinatamente per ogni campo dati xi(con i tra 1 e k) viene richiamato un costruttore:
    a. esplicitamente con chiamata costruttore xi(...) nella lista di inizializzazione
    b. implicitamente(se non compare nella lista) al costruttore di default xi( );

    ATTENZIONE: nell'ordine di dichiarazione non nell'ordine della lista di inizializzazione e dando la precedenza a copia di oggetti.
2. quindi esegue il codice del costruttore

Capitolo 3

Operazione Profonda: profonda: ossia ogni volta che si copia un puntatore deve essere effettuata la copia anche dell'oggetto puntato, attenzione è quindi costosa in termini di memoria.

Assegnazione profonda:

```
C& operator=(const C& b){
        if(this != &b){
                this.campo=copia(b.campo);
        }
        return *this;
}
```

Costruttore di copia profonda:

```
C(const C& b):campo(copia(b.campo)){}
```

Tempo di vita Variabili di classe:

1. AUTOMATICA (Stack) definite in un blocco allocate in stack quando esecuzione raggiunge la loro definizione e deallocate quando termina blocco in cui sono definite.
2. STATICA (Memoria Statica(campi dati statici, variabili globali, variabili statiche corpo funzione) vengono deallocate solo al termine del programma
3. DINAMICHE (Heap) area di memoria dinamica gestita dal programmatore, sono allocate quando viene eseguito operatore new e deallocate quando viene invocato l'operatore delete

Distruttore: 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

Ordine di distruzione parametri per valore è inverso all'ordine di costruzione.

Cast/Conversioni: il C++ oltre alla conversione come tipo1 a= (tipo1) b; con b di tipo != tipo1, delle tipologie di casting diverse ed esplicite:

- static_cast si basano su informazioni note a tempo di compilazione
- const_cast permette di convertire un puntatore/riferimento ad un tipo "const T" a un tipo "T", rimuovendo quindi attributo const
- dynamic_cast "tipo dinamico" di punt o rif non è noto a tempo di compilazione ma a tempo d'esecuzione. → nel capitolo 6

Funzioni/Classe amiche: Una funzione si dice "friend" di una classe, se è definita in un ambito diverso da quello della classe, ma può accedere ai suoi membri privati. Per ottenere ciò, bisogna inserire il prototipo della funzione nella definizione della classe (non importa se nella sezione privata o pubblica), facendo precedere lo specificatore friend.
Lo stesso si può fare pure per una classe come: friend class iteratore;

Dichiarazione incompleta di classi: class D; class C{ D*p;....} class D{...}

Puntatori smart: permette di usare puntatori intelligenti deallocati solo se nessun riferimento li raggiunge e gestisce modifiche senza sprechi di memoria ottenuto con aumento valore int riferimento nel costruttore di copia, assegnazione mentre il distruttore diminuisce di uno riferimento e se questo diventa pari a zero viene allora fatta la delete del valore puntato dal puntatore altrimenti si perde solo il puntatore che si sta distruggendo.

Capitolo 4

Template di Funzione: programmazione Generica permette di scrivere algoritmi i cui algoritmi non sono definiti su un tipo specifico che verrà quindi indicato successivamente.
template <class T>

T min(T ,T)
min<int>(int,double) NON FUNZIONA
min<int>(int,double) OK

Se richiamo la funzione min e assegno il suo valore di ritorno a un tipo diverso di T non avrò errori verrà solo fatto un casting.
Sono ammesse quattro tipologie di conversioni:

1. conversione da l-valore in r-valore(in particolare da T& a T)
2. da array a puntatore cioè da T[] a T*
3. conversione di qualificazione costante da T a const T
4. conversione da r-valore a riferimento costante ossia da r-valore di tipo T a const T&

Se il processo di deduzione degli argomenti non porta ad una istanziazione univoca si ottiene un errore di compilazione.
Quando un compilatore trova template di funzione si limita a memorizzare una rappresentazione interna di tale definizione senza compilare nulla e poi agisce in due modalità alternative:

- Compilazione per separazione rimosso da C++11 export
- Compilazione per inclusione (unico modello dopo C++11) dove le definizioni sono messe in un unico file header

Template di Classe: vista come esempio una coda è un tipo di ADT naturalmente templetizzabile con:
template <class T>
class Queue{...} → Queue<int> qi; Queue<bolletta>qb;
Con due tipologie: 1. parametri di tipo 2. parametri valore. I template classe sono quindi istanziati solo se si trovano istruzioni che allocano template per un dato parametro e i metodi dei template di classe solo quando sono invocati specificatamente.

Template e friend: le classi templatizzate possono utilizzare le amicizie in tre modi:

1. classe template con amici friend per cui questi amici hanno accesso a tutte le istanze della classe
2. classeB/metodoM template con classe template C che rende amiche verso C solamente una singola istanza di B e di M non di tutte.
3. template di classe B e funzioni M NON associate con insieme di parametri per il template disgiunti da quelli per C

Attenzione a membri statici in classi templatizzate: sono allocati pur essendo statici solo se la specifica tipologia viene utilizzata nell'invocazione del template e se il dato statico viene utilizzato.

Capitolo 5

classi contenitore: come vettori, liste, mappe sono tutti contenitori per altri tipi di dati e possiedono quindi tutti i seguenti metodi (alcuni da definire sempre):

- C::iterator
- C::const_iterator
- C::size_type ossia integrale senza segno indica la distanza tra due iteratori
- C(const C&)
- C& operator=(const C&)
- ~C() ridefinizione costruttore

- C operator* per dereferenziare puntatore o iteratore
- size_type size() ritorna dimensione contenitore
- bool empty()
- size_type max_size() massima dimensione contenitore
- operator ==
- operator<

Iteratori: come visto sono già inclusi nelle classe contenitori, servono per generalizzare i puntatori permettendo di iterare su un intervallo di oggetti e dereferenziando un iteratore otterremo l'oggetto puntato.
Le classi contenitore possiedono due metodi per la corretta inizializzazione degli iteratori

1. C::iterator begin() punta al primo elemento
2. C::iterator end() iteratore past-the-end

Mentre le classi iterator hanno:

- C::iterator& operator++ sposta iterator su elemento successivo
- C::iterator operator++(int)
- C::iterator& operator--() e operator---(int)

Sequenze: è contenitore con i suoi elementi memorizzati in ordine lineare stretto determinato dall'utente del contenitore, con le seguenti funzionalità principali:

- C c(n,t) e C c(n)
- c.insert(it,t) e c.insert(it,n,t) inserisce elemento t in c prima dell'elemento it
- c.erase(it) e c.erase(it1,it2)

Inoltre liste e vettori hanno:

- void push_back(t)
- void pop_back( )

Mentre le liste pure questi metodi complementari a quelli sopra indicati:

- void push_front(t)
- void pop_front( )

Vettori: vector, accesso casuale
Liste: liste doppiamente collegate con list e singolarmente collegate con slist.

Capitolo 6

Ereditarietà: ogni oggetto della classe derivata è utilizzabile anche come oggetto della classe base con una conversione implicita automatica che prende il sotto oggetto B dall'oggetto della classe derivata D.
Classe derivata D è sottotipo classe base B, mentre classe base B è supertipo di D.

Tipo statico e dinamico: esistono due tipi sia di puntatori che di riferimenti: quelli statici o quelli dinamici.
Statici non permette la modifica a run-time sono del tipo della compilazione quindi non è possibile una conversione da B => D mentre con quelli Dinamici sarà possibile farlo anche se rimane puramente astratto perché un puntatore o riferimento in memoria è sempre di un solo tipo non di più tipi contemporaneamente.

Accessibilita: class D : "tipo derivazione" class B

Membro \ Derivazione	public	protected	private
private	inaccessibile	inaccessibile	inaccessibile
protected	protetto	protetto	privato
public	pubblico	protetto	privato

Attenzione Amicizie Friend non sono Ereditate!

Name hiding rule: sia D una classe derivata da una classe base B, sia m( ) nome di metodo sovraccaricato nella classe B e se viene ridefinito in D nasconde tutte le versioni sovraccaricate di m( ) disponibili in D, ciò è chiamato _.
Vale sempre per 3 ridefinizioni del metodo mi:

1. stessa segnatura e lista di parametri e tipo di ritorno
2. stessa lista parametri ma diverso tipo di ritorno
3. diversa lista dei parametri

Ma tramite lo scoping posso togliere la funzione nascosta potendola richiamare con sempre con B => D tramite B::b ossia il membro b definito in B, lo stesso vale per i metodi.

Attenzione puntatore o riferimento sceglie sempre metodo del suo tipo statico, vedremo in seguito come scegliere il metodo del tipo dinamico.

Costruttore: un costruttore di una classe D derivata da B viene eseguito in questa sequenza, vale anche per quello di copia:

1. viene sempre e comunque invocato per primo un costruttore della classe B esplicitamente quando appare nella lista di inizializzazione oppure implicitamente il costruttore di default di B quando la lista non incluse una invocazione esplicita
2. successivamente viene eseguito il costruttore "proprio" di D ossia vengono costruiti prima i suoi campi e poi eseguito il corpo del costruttore

Altro esempio avendo classi Z => C => D quando creo un oggetto i costruttori partono però da Z poi C poi D creano una sorta di piramide dove alla base si trova Z e in cima troviamo D.

Assegnazione: assegnazione std di classe D derivata da B invoca assegnazione classe B sul sottoggetto e successivamente esegue l'assegnazione ordinatamente membro a membro dei campi dati propri di D invocando le rispettive assegnazioni

Distruttore: (inverso del COSTRUTTORE sequenza operazioni), (distruzione viene eseguita in maniera inversa rispetto ai costruttori) il distruttore di una classe D derivata direttamente da B chiama implicitamente il distruttore(std o ridefinito) della classe B per distruggere il sottoggetto di B soltanto dopo l'azione del distruttore standard propria di D cioè la distruzione dei dati propri di D nell'ordine inverso a quello di costruzione.
Se lo definisco in D prima viene eseguito quello poi i campi di D e infine quello di B.

Ereditarietà e Template:

1. Classe base template e Classe derivata da istanza classe base

```
template <class T>
class base{...};

class derivata : public <int> {...};
```

2. Classe base non template e Classe derivata template

```
class base{...};

template <class T>
class derivata : public base {...};
```

3. Classe base e Classe derivata entrambi template

```
template <class T>
class base{...};

template <class Tp>
class derivata : public <Tp> {...};
```

Metodi virtuali: utilizzando la keyword virtual prima del metodo lo rendiamo virtuale permettendo di effettuare il late binding legato al parametro dinamico del puntatore o riferimento. Metodo virtuale m in B rimane virtuale in tutta la gerarchia di classi derivate.
Attenzione se metodo m virtuale e richiamato su oggetto in cui il metodo virtuale non compare nella lista delle funzioni delle classi statiche non compila.
Nella classe derivata da B ridefinendo un metodo non possiamo cambiare il tipo ritornato a parità di parametri tranne nel caso il tipo di ritorno sia un puntatore o riferimento alla classe attuale derivata D.
Se cambio i parametri della funzione questa risulta essere una nuova funzione indipendente da quella virtuale anche se il nome è identico.

Vtable: Late Binding implica un **Overhead** del 50% nel 96 ora è circa 15% in più il tempo oltre allo spazio maggiore per la **vtable** ossia una tabella con gli indirizzi di tutti i metodi virtuali della classe C che conterrà almeno un metodo virtuale.
Per questa classe C ogni suo oggetto istanziato possiede un puntatore a funzione (detto vpointer) alla vtable di C quindi la selezione per la chiamata polimorfa a run-time è effettuata tramite l'accesso a queste strutture aggiuntive.

Distruttori virtuali: classe B e D sua derivata permettono di richiamare il distruttore corretto (ossia di D) del tipo dinamico di una classe derivata D richiamato su un puntatore di tipo statico della classe base B, non richiamando il distruttore di D tranne nel caso in cui definendo virtuale il distruttore di B lo saranno pure tutti quelli delle sue **classi derivate**.
Spesso si dichiara il distruttore virtuale e si lascia il corpo vuoto rendendo virtuali i costruttori standard.

Metodo virtuale puro: in alcuni casi dovremo definire classi puramente virtuali come B ossia delle classi base astratte di cui non dovremo mai creare un oggetto o usare oggetti. Svolge la funzione da "interfaccia comune" per le classi derivate di B. Oggetti di una classe base astratta sono **segnalati** come errori dal compilatore.

```
class B {
...
  virtal void G() = 0;
...
};
```

Per definire un metodo virtuale senza il CORPO in C++ bisogna mettere il marcatore =0 alla fine della sua dichiarazione. Una classe D derivata da B si dice **concreta** se implementa tutti i metodi virtuali della classe B.

Identificare tipo a run-time con operatori RTTI (su classi polimorfe!): un tipo T è polimorfo se include almeno un metodo virtuale e lo saranno tutte le sue classi derivate.

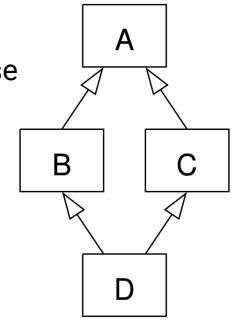
1. typeid permette di determinare il tipo di una qualsiasi espressione a tempo d'esecuzione se espressione è tipo polimorfo ritorna il tipo dinamico di quella espressione. Attenzione che typeid:
    a. se la classe non è polimorfa restituisce tipo statico.
    b. su puntatore non dereferenziato ritorna il tipo statico
    c. ignora sempre attributo const
2. **dynamic_cast** invece permette di convertire puntatori o riferimenti classe base B polimorfa a classe D sua sottoclasse. Ponendo dynamic_cast<D*>(p) con tipo dinamico di p=E* allora:
    a. se E è derivato da D allora la conversione andrà a buon fine ritorna 1
    b. se E non è sottotipo di D allora la conversione fallirà generando un puntatore nullo 0

    B è tipo polimorfo, D =< B: posso fare:
    B* => D* pure con B& => D&
    Downcasting - "dall'alto verso il basso": ossia dalla base alla derivata andando contro la conversione implicita.
    Safe downcasting dove si gestisce in modo controllato la conversione safe o safe cast gestendo gli errori di conversione che si verificano a run time.

    D* => B* pure con D& => B&
    Upcasting - "dal basso verso l'alto": ossia da classe derivata a classe base

Ereditarietà multipla: una classe può essere derivata da più di una classe base
class D : public C1, public C2{...} se metodo presente sia in C1 e C2 su oggetto di D devo utilizzare operatore di scoping altrimenti errore di compilazione.

Derivazione virtuale: risolve il problema introdotto da un ereditarietà a diamante quindi nel caso seguente previene la creazione doppia della classe A qualora sia creato un oggetto D che avrà pure problemi d'ambiguità usando metodi di A non sapendo quale delle due A generate utilizzare.



Si risolve questo problema usando keyword virtual prima della protezione della derivazione classe madre.
class A{...} class B : virtual public A{...} class C : virtual public A{...}
class D : public B, public C {...}
In questa gerarchia A è detta base virtuale per classi B e C.
In caso di accessi classi padre diverse per B e C esempio B privato e C pubblica prevale l'apertura di C ma se con scoping si richiama metodo di A da B:: è illegale essendo inaccessibile invece richiamando metodo da D si passa per C.
Costruttori classe derivata con classi virtuali multiple:

1. per primi sono eseguiti solo una volta tutti i costruttori delle classi virtuali che si trovano nella gerarchia di derivazione di D per ognuna di queste si arriva alla classe base virtuale e si prosegue con metodo("left to right top-down order")
2. vengono richiamati i costruttori delle superclassi dirette non virtuali
3. infine si esegue il corpo del costruttore di D dove sono costruiti i campi propri di D.

Attenzione se in B o C viene richiamato costruttore A questa chiamata è ignorata se è già stata costruita la classe base virtuale A.
Con schema elimino prima solo classi virtuali totalmente o che parzialmente totali poi tutte le dirette.

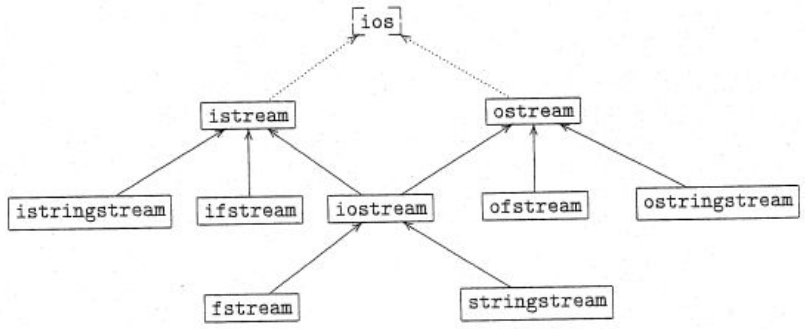
Capitolo 7

Eccezioni:

- **Il costrutto try**
  La parola-chiave try introduce un blocco di istruzioni.
  try{ res = fun(39); }
  Le istruzioni contenute in un blocco try sono "sotto controllo": in esecuzione, qualcuna di esse potrebbe generare un errore. Nell'esempio, la funzione fun potrebbe chiamare un'altra funzione e questa un'altra ancora ecc... , generando una serie di pacchetti che si accumula sullo stack. L'area dello stack che va da un un blocco try in su è detta: exception stack frame e costituisce l'insieme di tutte le istruzioni controllate.
- **L'istruzione throw**
  Dal punto di visto sintattico, l'istruzione throw è identica all'istruzione return di una funzione (e si comporta all'incirca nello stesso modo):
  throw espressione;
- **Il gestore delle eccezioni: costrutto catch**
  La parola-chiave catch introduce un blocco di istruzioni che ha lo stesso formato sintattico della definizione di una funzione, con un solo argomento e senza valore di ritorno.
  catch (tipo argomento) { ..... blocco di istruzioni ..... }

Capitolo 8

Gerarchie classi I/O:



Capitolo 9

Utilizzo Qt Base

Capitolo 10

C++11 ha introdotto nuove funzioni e strumenti anche per semplificare sviluppo:
Lambda espressioni: o funzione anonima consente di definire una funzione a livello locale dove viene invocata
[capture list variabili prese] (lista parametri) ->return-type {corpo della funzione}

Riferimenti rvalue: T&& variabili riferimento rvalue possono essere solo associati a rvalues, come oggetti temporanei e valori di tipi predefiniti.

Inferenza automatica di tipo: possibile dichiarazione con simultanea inizializzazione permettendo di dichiarare variabili con keyword auto come iteratori di vettori.

Inizializzazione uniforme: introduce una singola inizializzazione con parentesi graffe nel quali delimitare la lista di valori.

Keyword default e delete: =default permette di generare l'implementazione di default per la funzione inoltre si può pure fare l'opposto ossia non rendere tali funzioni disponibili tramite la sintassi =delete, utile per evitare la copia di oggetti che potremmo mettere quindi impedire facilmente "bannando" costruttore di copia per la classe specifica.

Overriding esplicito: aggiunti altri due modificatori, override per dichiarare esplicitamente quando una funzione virtuale deve necessariamente essere un overriding e si può inoltre pure usare il nuovo modificare final che rendere proibito effettuare derivazioni dall'ovveriding di m() nelle classe derivate da questa.

Nullptr: questa keyword sostituisce la macro per il preprocessore NULL per prevenire problemi con piattaforme diverse.

Chiamate di costruttori: un costruttore può finalmente richiamare un altro costruttore come si poteva fare in Java detta delegation a un costruttore già definito della stessa classe con uso di argomenti di default per quello richiamato o ottenuti da valori passati.