1_Namespace

Namespace

```
namespace nome_namespace {
        struct nome_struct {...};
        class nome_classe{...}; //solo dichiarazione
}
```

Alias:

- `namespace nome1 = nome_namespace;` ridefinisce `nome_namespace` a `nome1` permettendo l'uso del namespace attraverso nome1::nome_dichiarazione. `nome1` è di fato un **alias**.

Direttiva d'uso:

- `using namespace nome_namespace;` apre il cancello e permette l'utilizzo di tutti i metodi e tipi del namespace senza scooping operator (::).
  ex. `using namespace std;`

Dichiarazione d'uso:

- `using nome_namespace::nome_dichiarazione` permette di usare `nome_dichiarazione` (funzione, tipo ecc) proprio di namespace senza utilizzare lo scooping operator per quello specifico `nome_dichiarazione`.

Attenzione:

- tipi con lo **stesso nome** in ***namespace diversi***, ***sono diversi per il compilatore*** come tipi perciò se provo assegnare da un namespace un tipo con nome `tipo1` ad un altro tipo `tipo1` *in un altro namespace* ottengo un errore in compilazione.

2_Strings

Strings

```
#include <string>
using namespace std;
int main(){
        char *s = "stringa alla c\0";
        char *s1 = "stringa alla c"; //equivalente ad s
        string st("stringa alal c++"); //creata con costruttore di copia
        string str = "stringa alla c++";
        st.size(); //metodo della classe string
        string st1; //costruttore di default (dichiara stringa e la
lascia vuota)
string st2(st1); //costruttore di copia di string
}
```

- size(), metodo di *string* che ritorna la lunghezza della stringa senza terminatore \0

  Classi

  Costruttore di copia

  Costruttori

3_Abstract Data Type

ADT

L'ADT viene definito distinguendo nettamente la sua interfaccia, divisa in:

1. Operazioni sui dati
2. Implementazione interna *(operazioni)*:
   - come sono conservate le informazioni.
   - come vengono manipolate le informazioni.

ADT = Valori + Operazioni.

Interfaccia costituita da metodi pubblici o Operazioni proprie.

Rappresentazione interna dell ***ADT*** **INACCESSIBILE**.

- ADT esempio: tipo primitivo *int*.
- `struct` non rispetta il concetto di ADT, manca completamente l'information hiding.

4_Classi

Classe

Concetto di classe, implementa gli Abstract Data Type

- ***Dichiarazione*** dei membri della classe:

  - Campi dati (variabili):

    1. Privati(di default), solitamente i dati sono sempre privati.
    2. Pubblici.
    3. Protetti.
    4. Statici.

  - Metodi (funzioni interne alla classe, solo dichiarate):

    1. Privati(di default) usati di solito come supporto per l'implementazione di altri metodi nella classe.
    2. Pubblici.
    3. Protetti.
    4. Statici.
    5. Virtuali.

- ***Implementazione*** o ***Definizione*** della classe

  - Implementazione dei metodi (definisco come agisce una funzione)

> In una classe i metodi invocati hanno un puntatore ***this*** di tipo C* che PUNTA all'oggetto d'invocazione in memoria (*this è l'oggetto di invocazione
> Ex. (*this).sec, this->sec per la classe orario; this è il puntatore ad un oggetto di tipo orario, de-referenziandolo si può accedere a campi, anche privati e ai metodi)

> ***Durante l'implementazione*** di un metodo di una classe, ***ho accesso a tutti i campi*** dati, **anche privati** della classe.

Attenzione:

> `this` è una keyword riservata non utilizzabile nella dichiarazione del metodo.

Tipo di classi

- concreta.
- astratta.

Relazioni:
Costruttore di copia
Distruttore
Container
Rule of Three

5_Costruttori

Costruttori

```
class orario{
public:
    orario(); //1
    orario(int, int); //2
    orario(int =0, int =0); //3
}
```

1. costruttore di default
2. costruttore a 2 parametri
3. costruttore a 0, 1, 2 parametri (con valori di default)+ conversione di tipo ***int => orario***
    - Explicit

- Può usare anche una lista di inizializzazione.

Comportamento del costruttore

Sia C una classe con campi dati $x_1, x_2, \ldots, x_k$ di qualsiasi tipo.

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

Nella classe C suppongo di definire un costruttore come segue:
$C(tipo_1, \ldots, tipo_n)$ $\{//codice\}$
Il comportamento del costruttore è:

1. Per ogni campo x_j di ***tipo non classe*** T_j (ovvero tipo primitivo o derivato), viene allocato in memoria lo spazio per contenere un valore di tipo T_j ma viene lasciato indefinito.
2. Per ogni campo x_i di ***tipo classe*** T_i viene invocato il **costruttore di default** di $T_i()$
3. Viene eseguito il codice del ***corpo del costruttore*** ({//codice})

> I punti 1 e 2 vengono eseguiti per tutti i campi dati $x_1, \ldots, x_k$ seguendo il loro ordine di dichiarazione in C.

> In breve: Prima costruisco i campi dati seguendo il loro ordine di dichiarazione poi eseguo il corpo del costruttore.

Costruttori nelle classi derivate

Costruttore presente

La lista di inizializzazione di un costruttore di una classe ***D derivata direttamente da B*** può contenere invocazioni di costruttori per **campi dati propri di D** e **costruttore per la classe base B**.
L'esecuzione del costruttore di D avviene come:

1. Sempre invocato per primo il costruttore della classe base B o **esplicitamente o implicitamente** il costruttore di ***default di B se non esplicitato*** il costruttore di B nella lista di inizializzazione.

   > Se presenti "nonni" o "bisnonni" non vengono richiamati i costruttori, al massimo si richiama quello della classe B da cui discende direttamente.

2. Viene eseguito il costruttore proprio di D ovvero vengono costruiti i dati propri di D nel modo usuale.
3. viene eseguito il corpo del costruttore

Costruttore assente

Nel caso di assenza di costruttori per la classe derivata è al solito disponibile il costruttore di default standard di D che :

1. Richiama il costruttore di default di B
2. Si comporta come il costruttore di default standard proprio di D ovvero richiama il costruttore di default dei campi dati di D nel loro ordine di dichiarazione nella classe D.

Costruttore in presenza di basi virtuali

Struttura a diamante $B \leq A, C \leq A, D \leq B$ e $D \leq C$

1. **Per primi** vengono chiamati una sola volta i **costruttori delle classi base virtuali** nella gerarchia di derivazione di D. Potendoci essere più classi base virtuali nella gerarchia di D, la ***ricerca delle classi base virtuali nella gerarchia*** procede seguendo l'ordine ***da sinistra a destra e dall'alto verso il basso***.
2. Una volta invocati i costruttori nella gerarchia di derivazione di D vengono chiamati i ***costruttori delle super-classi dirette non virtuali di D***. Questi ultimi non richiamano eventuali costruttori di classi virtuali già chiamati al passo 1.
3. Viene ***eseguito il costruttore proprio di*** D, ovvero vengono inizializzati i campi dati propri di D nell'ordine di dichiarazione e poi viene eseguito il corpo del costruttore di D

NB:

> **Se non esplicitate le chiamate ai costruttori nei punti 1 e 2 vengono automaticamente inserite dal compilatore nella lista di inizializzazione del costruttore di D**. Si tratta di chiamate implicite ai costruttori di default.

```
class A {
public:
        A() { cout << "A ";}
};

class B {
public:
        B() { cout << "B ";}
};

class C : virtual public A {
public:
        C() {cout << "C ";}
};

class D : public B {
public:
        D() {cout << "D ";}
};

class E : public C, virtual public D {
public:
        E(){ cout << "E ";}
};

int main() { E e; } //stampa: A B D C E
```

6_Const Sega

Const mantra

Scrivere tutto ciò che è possibile come costante.

Oggetti costanti

Oggetti dichiarati ***costanti*** possono essere utilizzati sono ***con metodi*** dichiarati ***costanti***.

Metodi costanti

Per metodi dichiarati costanti, il compilatore verifica che nella sua implementazione non ci siano istruzioni che possano provocare side effect sull'oggetto di invocazione.

> In un metodo costante di una classe C, il puntatore `this` è di tipo `const C*`

Recap:

```
int x=2;
int y=3;
int& a=x; //legale, alias
int& a1=2; //ILLEGALE scannato se si fa
a=5;
a=y; //legale, assegnamento
int* p=&x;
*p=5;
p=&y //legale
int * const p1=&x;
*p1=5; //legale cambio valore ad un puntatore costante
p1=&y; //ILLEGALE cambio indirizzo ad un puntatore costante
int & const r = x; //ILLEGALISSIMO un riferimento costante??? non esiste
(una volta dichiarato il riferimento rimane quello, non può essere
cambiato verso un altra variabile)
const int* p2 = &x; //puntatore a tipo costante
*p2=5; //ILLEGALE cambio valore a un puntatore A costante
p2=&y //legale cambio indirizzo ad un puntatore a costante
const int *const p3 = &x; //puntatore costante a tipo costante
*p3=5; //ILLEGALE
p3=&y; //ILLEGALE
```

```
const int& r1 = x; //riferimento a tipo costante
r1=5; //ILLEGALE
r1=y; //ILLEGALE
const int &r2 = 4; //legale
r2=5; //ILLEGALE
r2=y; //ILLEGALE
const int& const r3=2; //ILLEGALE
```

Esempi per tipo:

```
int x=2;
int& a = x; //alias
int& a1=2; //ILLEGALE scannato se si fa
a=5;
a=y; //legale
```

```
int x=2;
int* p=&x;
*p=5;
int y=3;
p=&y //legale
```

```
int x=2;
int * const p=&x;
*p=5; //legale
int y=3;
p=&y; //ILLEGALE
```

```
int x=2;
int & const r = x; //ILLEGALISSIMO un riferimento costante??? non esiste
(una volta dichiarato il riferimento rimane quello, non può essere
cambiato verso un altra variabile)
```

```
int x=2;
const int* p = &x; //puntatore a tipo costante
*p=5; //ILLEGALE
int y=3;
p=&y //legale
```

```
int x=2;
const int *const p = &x; //puntatore costante a tipo costante
*p=5; //ILLEGALE
int y=3;
p=&y; //ILLEGALE
```

```
int x=2;
const int& r = x; //riferimento a tipo costante
r=5; //ILLEGALE
int y=3;
r=y; //ILLEGALE
```

```
const int& r = 4; //legale
r=5; //ILLEGALE
int y=3;
r=y; //ILLEGALE
```

```
const int& const r=2; //ILLEGALE
```

```
void fun(const int& r);
int x=2;
fun(x); //legale
fun(4); //legale
```

```
const int& fun() {return 4; //legale}

fun() = 5; //ILLEGALE
```

```
void fun_ref(const int& r);
void fun_ptr(const int* p);
int x=2;
fun_ref(x);
fun_ptr(&x);
fun_ref(4); //legale
fun_ptr(&4); //ILLEGALE
```

Ex.

```cpp
#include <iostream>
using namespace std;

int& fun(int& x){x+=2; return x;}

int main(){
 int y =2;
 cout << fun(y) << fun(y)+2;
 return 0;
}*/ //stampa 4 e 8, prima ritorna y che è aumentato di 2 da fun (y=4), poi torna y a cui è stato sommato 2 in fun(y=6) ci somma 2 e quindi stampa 8
```

7_Static type

Static type

La dichiarazione static è **sufficiente** farla **al momento della dichiarazione** del tipo o del metodo.

Tipo statico

Valore comune tra tutti gli oggetti della classe.

> Un campo statico viene **inizializzato all'esterno della classe** ed è sempre richiesta.

> È presente un' ***unica copia in memoria*** di ciascun ***campo dati statico***.

> Un campo dati dichiarato `static const` se è di ***sola lettura***, deve essere inizializzato al momento della dichiarazione.

Metodo statico

> Non è necessario un oggetto di invocazione per metodi statici.

> Non ha nessun senso il `this` (non ho oggetto di invocazione).

```
const orario inutile;
cout << inutile.OraDiPranzo().Ore()
```

diventa:

```
class orario{
public:
    static orario OraDiPranzo(); //basta nella dichiarazione
    static const int Secondi_di_un_Ora = 3600; //se solo in lettura
dichiaro il tipo statico costante
};

orario orario::OraDiPranzo() {
    return orario(13,15);
}

cout << orario::OraDiPranzo().Ore()
```

8_Explicit

Explicit

Aggiungendo ***explicit*** come prefisso ad un costruttore, richiede di specificare il costruttore per effettuare la conversione ***tipo => C***.

Riepilogando:

- ogni volta che ho una conversione ***tipo => C***, devo esplicitare `C c = C(t)` oppure `C c(C(t))` dove `C(t)` è il costruttore ad un parametro della classe ***C***, poi viene chiamato il costruttore di copia. ***t*** può essere il tipo specificato nella dichiarazione del costruttore oppure un tipo di cui sia disponibile la conversione per promozione data da linguaggio al tipo nella dichiarazione.
- `C c = 4` da errore, non viene implicitamente chiamato il costruttore sull' ***int***

Esempio:

```cpp
#include <iostream>
using namespace std;

class C{
public:
    int value;
    explicit C(int x): value(x) {std::cout << "funziona\n" << value <<
endl; };
    C(const C&) {cout << "Cc\n"; }
};


int main()
{
        C x = C('A'); //1
}
```

```cpp
#include <iostream>
using namespace std;

class C{
public:
    int value;
    explicit C(int x): value(x) {std::cout << "funziona" << endl <<
value; };
        C(const C&) {cout << "Cc\n"; }
};

class Cx{
public:
    char c = 'A';
    operator C() {return C(c);}//esplicito il costruttore in quanto
                //è presente explicit (+ conversione implicita
char=>int)
};

int main()
{
    Cx y;
    //C x = y; non funziona.
    //Conversion from ‘Cx’ to non-scalar type ‘C’ requested
    C x = y; //2
}
```

1. stampa funziona 65 Cc promozione implicita da linguaggio char=>int costruzione temporaneo anonimo da A=65 ascii table, costruzione di copia

dal temporaneo anonimo a x.

2. stampa funziona 65 Cc Cc conversione Cx=>C con operator C(), successivamente chiamo costruttore ad un parametro su c conversione char=>int quindi temporaneo anonimo di tipo C costruzione di copia per il valore di return e invocazione c. di copia per x da temporaneo anonimo ritornato.

Costruttore di copia
Conversioni implicite
Costruttori
Conversione Classe a tipo

9_Conversione Classe a tipo

Conversione C=>T

`operator T() {return ...; }` il return deve essere dello stesso tipo di T.

Esempio:

```
class orario{
public:
        operator int() {return sec; };
        ...
};

int x = o;
```

Classi

10_Overloading operatori

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 virgola

L'operatore virgola separa delle espressioni (i suoi argomenti) e ritorna il valore solamente dell'ultima espressione a destra, mentre tutte le altre espressioni sono valutate per i loro effetti collaterali.

Why?

```
int main() {
  int a = 0, b = 1, c = 2, d = 3, e = 4;
  a = (b++, c++, d++, e++);
  cout << "a = " << a << endl; // stampa 4
  cout << "c = " << c << endl; // stampa 3
}
```

Operator =

Interno

```
C& operator=(const C& c){
if (this != &c) //controllo che non sia nel caso c = c
        {
        //pulizia heap dell'oggetto puntato da this e copia (se
```

```
        //necessario profonda) da c all'oggetto puntato da this
        return *this;
        }
}
```

operator= profondo

per classi derivate

```
D& operator=(const D& d){
if (this != &d)
        {
        this->B::operator=(d) //dove B è la classe da cui D deriva
direttamente
        //assegnazioni necessarie per i campi dati della classe C
        //return *this;
        }//da rivedere non sicuro il return
}
```

Operator <<

Per risolvere l'output vado *esternamente*:

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

```
//firend std::ostream& operator<<(std::ostream& os, const C& c) nella
classe se
//non voglio usare metodi pubblici della classe per l'output

std::ostream& operator<<(std::ostream& os, const C& c){
        //return os << o.Ore() << ':' << o.Minuti() << ':' <<
o.Secondi();
        return os << c.metodo_o_tipo;
}       //sempre esterna, altrimenti mi si sarebbe incasinato il modo di
scrivere
        //l'output, ora posso scriverlo nel modo classico
```

Operator+

Vado esterno e uso i metodi pubblici della classe per risolvere la somma:

```
C operator+(const C& a, const C& b){
        //...
```

```
        return C;
}//in questo modo viene chiamato il costruttore se gli operandi non sono
di tipo C per fare se possibile una conversione di tipo
```

Altrimenti problemi come:

```
class C{
public:
        int x;
        C(int y=0): x(y) {};
        C operator+(const C&) const; //nella pate pubblica della classe
};

int main(){
        C orario::operator+(C o) const; //operatore+ interno
        C::orario(int o); //costruttore ad un parametro
        C t(), s;
        s=t+4; //OK conversione implicita int=>orario
        s=4+t;  //errore non posso chiamare un metodo su un oggetto
diverso dal tipo
                        //della classe di invocazione del metodo, non è
amessa conversione
                        //implicita
}
```

Operator ==

É interno alla classe

```
bool C::operator==(const C& c) const {
        return ...;
}
```

Metodo interno alla classe quindi vado a metterlo nella parte pubblica e lo definisco nel file .cpp con l'operatore di scoping.

Operator[]

É interno alla classe

iteratore di subscripting.

Operator++ (post e pre)

É interno alla classe

Operator++ prefisso:

```
T& operator++() {
        ...
        return *this;
}
```

Operator++ postfisso:

```
T operator++(int) {
        T aux = *this;
        ...
        return aux;
}
```

Operator*

Sia definita una **classe nodo** in C con campo **info** di tipo T e campo next di tipo $nodo*$ e una classe iteratore con campo privato **punt** di tipo $nodo*$. Operator* è un metodo di iteratore.

```
T& operator*() const {return punt->info; }
```

Operator->

Sia definita una **classe nodo** in C con campo **info** di tipo T e campo next di tipo $nodo*$ e una classe iteratore con campo privato **punt** di tipo $nodo*$. Operator-> è un metodo di iteratore.

```
T* operator->() const {return &(punt->info); }  //al puntatore tornato
viene

//applicata l'operatore

//-> ordinario
```

> operator* e operator->, sono metodi costanti, ma sono compatibili anche con tipi di ritorno non costanti, infatti, **this diventa un puntatore a costante**, quindi **punt è costante ma l'oggetto puntato no**! Perciò dereferenziando punt posso modificare l'oggetto puntato.

11_Costruttore di copia

Costruttore di copia

Il costruttore di copia C(const C&) viene invocato in automatico quando:

- Un **oggetto** viene ***dichiarato e inizializzato*** **ad un altro oggetto** della ***stessa classe***:

```
orario adesso(14,30);
orario copia = adesso; //cc
orario copia1(adesso); //cc

orario copia1;
copia1 = adesso; //È un'assegnazione non una copia! Solo
inizializzazione
```

- Un **oggetto** viene **passato per valore** come parametro di una funzione

```
orario orario::Somma(orario o) const {
        orario aux;
        aux.sec = (sec + o.sec) % 86400;
        return aux;
};

ora = ora.Somma(DUE_ORE_E_UN_QUARTO);
```

- Quando una funzione ritorna per valore tramite return, un oggetto

```
//come prima
orario Somma(orario o) const {
        ...
        return aux;//tornato per valore
}
```

Ottimizzazione da compilatore

Il compilatore ***g++*** di default compie l'**ottimizzazione** di **omettere** la creazione di un **oggetto temporaneo** utilizzato **per** la **creazione di un oggetto dello stesso tipo**.

> Non completamente deterministico quindi a fini didattici da escludere con `g++ --fno-elide-constructors`

Costruttore di copia standard:

$C\left(const\ C\&\ obj\right)\ :\ x_1\left(obj.x_1\right),\ldots,\ x_k\left(obj.x_k\right)\{\}$

Ridefinizione del costruttore di copia

Copia profonda

12_Make

Make

Compilazione automatizzata con make e makefile (qmake per il progetto)

```
# Commento: variabile CC è il comando che invoca il compilatore
CC=g++
# CCFLAGS: flags per il compilatore
CCFLAGS=-Wall -march=x86-64

my_prog : main.o kbd.o video.o help.o print.o
$(CC) $(CCFLAGS) -o my_prog main.o kbd.o video.o \\
help.o print.o

main.o : main.cpp config.h
$(CC) $(CCFLAGS) -c main.cpp -o main.o

kbd.o : kbd.cpp config.h
$(CC) $(CCFLAGS) -c kbd.cpp -o kbd.o

video.o : video.cpp config.h defs.h
$(CC) $(CCFLAGS) -c video.cpp -o video.o

help.o : help.cpp help.h
$(CC) $(CCFLAGS) -c help.cpp -o help.o

print.o : print.cpp
$(CC) $(CCFLAGS) -c print.cpp -o print.o

clean :
rm *.o
echo "pulizia completata"
```

il comando clean pulisce i file nella directory corrente, in questo caso tutti i file (*) con estensione .o

Il formato è

> Target: dependencies...
> Command

Target

Di solito è il nome dell'eseguibile o del file oggetto da ricompilare, oppure può anche essere un'azione (es. clean) che è un comando proprio del programma: `$make clean`

Dependencies

Usate in input per generare l'azione target e solitamente sono multiple. Solitaente file o azioni da cui dipende il target

Command

Comando da eseguire, può essere più d'uno e solitamente viene applicato alle dependencies.

13_Direttive di compilazione condizionale

Direttive di compilazione

```
#ifdef identificatore
        text
#endif

#ifndef identificatore 1
        text1
#endif

#elif defined identificatore2
        text2
#endif
```

Defined verifica se un simbolo è stato definito o meno

```
#ifdef identificatore

#ifndef identificatore
```

Equivalgono a:

```
#if defined identificatore

#if !defined identificatore
```

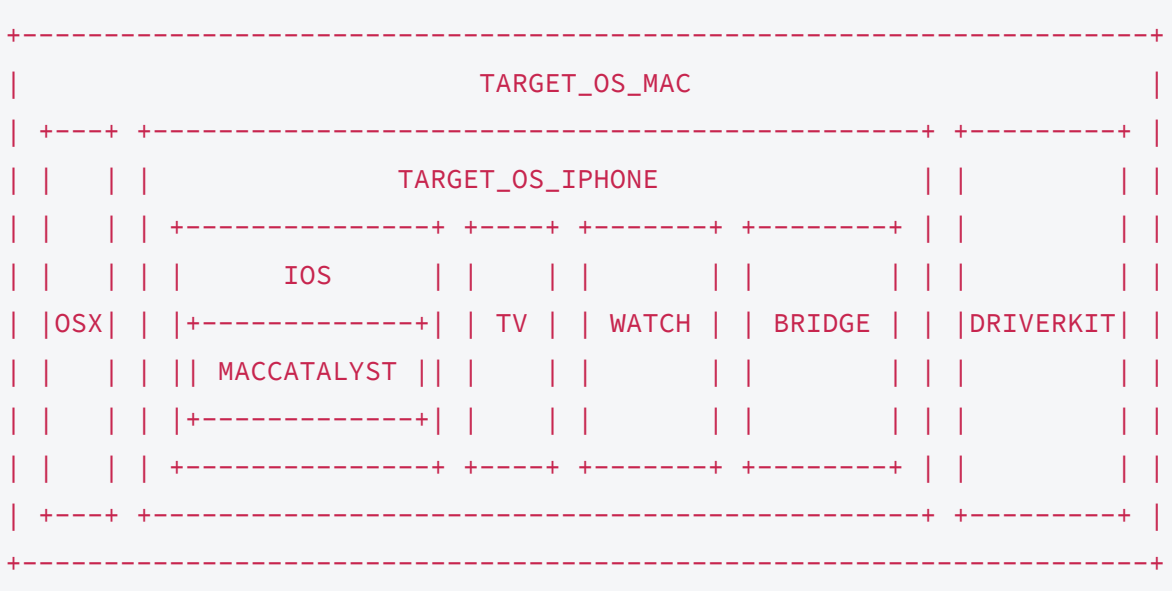
Esempio:

```
//file orario.h
#ifndef ORARIO_H
#define ORARIO_H
class orario{
        ...
};
#endif
```

Come verifico il SO?

Apple (OS X and iOS)

```
//apple, ios or iphone
#ifdef __APPLE__ //or __MACH__
#ifdef TARGET_OS_TV
#endif
```



ex: `#if TARGET_OS_any_on_the_table`

Windows/mingw

```
//windows 32bit
#ifdef _WIN32
        text
#endif

//windows 64bit
#ifdef _WIN64
        text1
#endif
//
#ifdef __CYGWIN__
        text2
#endif
```

CYGWIN

```
//mingw 32bit
#ifdef __MINGW32__
        text
#endif
//mingw 64bit
#ifdef __MINGW64__
        text1
#endif
//visual C++
#ifdef __MINGW32__
        text2
#endif
```

Per verere i define di gcc: `gcc -dM -E - <NUL:`

Di più

Linux

```
#ifdef __linux__
        text
#endif
```

Per verere i define di gcc: `gcc -dM -E - </dev/null`

Android

```
#ifdef __ANDROID__
        text
#endif
```

Unix

```
#ifdef (unix) || defined(__unix__) || defined(__unix)
#define PREDEF_PLATFORM_UNIX
        text
#endif
```

FreeBSD

```
#ifdef __FreeBSD__
        text
#endif
```

14_Lista di inizializzazione

Lista di inizializzazione

In un costruttore anche di copia posso avere una lista di inizializzazione del costruttore.

Sia C una classe con campi dati $x_1,\ x_2, \dots,\ x_k$ di qualsiasi tipo, un costruttore con lista di inizializzazione per i campi dati $x_{i1},\ x_{i2}, \dots,\ x_{ij}$ è definito tramite:

$$C(T_1, \dots, T_n)\ :\ x_{i1}\ (\dots), \dots,\ x_{ij}\ (\dots)\ \{//codice\};$$

Il comportamento del costruttore è:

1. ***In modo ordinato*** per ogni campo dati $x_i\ (1 \leqslant i \leqslant k)$ viene richiamato un costruttore:
   - Esplicitamente tramite una chiamata ad un costruttore $x_i\ (\dots)$ definita nella lista di inizializzazione.
   - Implicitamente (non appare nella lista di inizializzazione) tramite una chiamata al costruttore di default $x_i\ ()$.
2. Viene eseguito il codice del corpo del costruttore ({//codice})

> Si parla di costruttore e costruttore di copia anche per campi dati di tipo non classe (primitivi o derivati): posso includere inizializzazioni anche per questi tipi di dato.

> La chiamata implicita al costruttore di default standard per un campo dati di tipo non classe, al solito, alloca spazio in memoria ma lascia indefinito il valore.

> **L'ordine in cui vengono invocati i costruttori**, implicitamente o esplicitamente, **è sempre determinato dalla lista ordinata dei campi dati nella classe C, qualsiasi sia l'ordine delle chiamate nella lista di inizializzazione**

NB:

Se un campo dati é di **tipo costante**, è **riferimento** oppure **puntatore costante**, **necessita di essere inizializzato**.

- Un **campo dati di tipo costante** se non lo inizializzo tramite lista di inizializzazione, avrà un valore casuale oppure un valore di default nel caso sia un oggetto. Questo valore non è più modificabile eccetto il caso in cui usi un const_cast.
- Un **riferimento** deve essere associato ad una variabile, non posso fare `int& i;` ma per forza `int& i = variabile_che_tipa;` perciò deve essere inizializzato.
- Un **puntatore** costante necessita di un indirizzo di inizializzazione altrimenti è illegale. `int* const p` è illegale, `int* const p = indirizzo` va bene.

Nota:

> Nella lista di inizializzazione se vengono creati temporanei anonimi, per l'inizializzazione della variabile viene chiamato il costruttore di copia per assegnare il temporaneo anonimo alla variabile. Es: C() : d(D(5)) ... {}; ta per int => D e Dc per copia su d.

15_Container

Definizione di contenitore

Un ***contenitore*** è una classe, una struttura dati, o un Abstract Data Type le cui istanze sono collezioni di altri oggetti. In altre parole sono usate per contenere oggetti in un modo organizzato utilizzando specifiche regole di accesso. La dimensione del contenitore dipende dal numero di oggetti contenuti.

Caratteristiche dei container

Le classi di tipo container ci si aspetta che implementino metodi per:

- Creare container vuoti (constructor)
- Inserire oggetti nel container
- Eliminare oggetti dal container
- Eliminare ***tutti*** gli oggetti dal container (clear)
- Accedere agli oggetti nel container
- Avere il numero di oggetti nel container

> A volte i container sono implementati assieme ad un iteratore.

Contenitori in STL: string, list, vector, map, set, ...

16_Aliasing

Fenomeno dell'interferenza o aliasing

Cause:

1. Condivisione di memoria.
2. Funzioni con side effects.

Definizione:
Il termine Aliasing descrive una situazione in cui la posizione in memoria di un dato può essere acceduta da diversi nomi simbolici nel programma. Perciò, modificare un dato attraverso uno dei nomi significa modificarlo anche per tutti gli altri alias, che può essere un fattore che il programmatore non si aspetta.

17_Metodi copia e distruggi

Deep Copy

- Definizione di 2 metodi aggiuntivi:

Nella parte privata della classe C di cui fare copie profonde: (nodo è una classe nella parte privata di C con costruttore/i, capo **info** e campo **next**)

```
class C {
public:
        C();
private:
        class nodo {
        public:
                nodo();
                nodo(const T&, nodo*) //T è il tipo di dato del campo
info
                T info;
                nodo* next;
        };

        static nodo* copia(nodo*);
        static void distruggi(nodo*);

        nodo* first;
};

C::nodo::nodo() : next(0) {}    /*senza definire l'inizializzazione di t
do per scontato che T abbia un costruttore di default */

C::nodo::nodo(const T& t; nodo *n) : info(t), next(n) {}

C::C() : first(0) {}
```

Implementati come:

RIcorsivamente:

```
C::nodo* C::copia(nodo* n){
        if(!n) return 0; //se la lista è vuota torno 0 (e quindi se sono
alla fine)
        else
        return new nodo(n->info, copia(n->next));       /*ritorna
l'indirizzo di un nuovo nodo con campo info uguale a info del nodo
```

```
corrente e campo next l'indirizzo tornato dalla chiamata ricorsiva sul
nodo successivo*/
}

void c::distruggi(nodo* n){
        if(n){
                distruggi(n->next); //vado fino all'ultimo nodo
                delete n; //elimino a cascata i nodi dall'ultimo
        }
}
```

Iterativamente:

```
C::nodo* C::copia(nodo* n){
        if(!n) return 0;

        nodo* primo = new nodo(n->info, 0);
        nodo* q = primo;

        while(n->next){
                n = n->next; //avanzo sul nodo successivo
                q->next = new nodo(n->info, 0); /**/
                q = q->next;
        }
        return primo;
}

void C::distruggi(nodo* n){
        if(n){
                nodo* q = n;
                while(n){
                        n = n->next;
                        delete q;
                        q = n;
                }
        }
}
```

Se devo anche implementare operator= allora:

```
class C{
public:
        ...
        static nodo* copia(nodo*);
        static void distruggi(nodo*);
        C& operator=(const C&);
        ...
```

```
};

C& C::operator=(const C& c){
        if(this != &c){ //se b = b non devo cancellare nulla!
                distruggi(first); //pulizia heap
                first = copia(c.first); //copia profonda
        }
        return *this; //anche nel caso in cui faccia b = b devo tornare
qualcosa!
}
```

Ridefinizione del costruttore di copia:

```
class C{
public:
        ...
        static nodo* copia(nodo*);
        static void distruggi(nodo*);
        C(const C&);
        ...
};

C::C(const C& c) : first(copia(c.first)) {}
```

Distruttore

Rule of Three

18_Tempo di vita di una variabile

Tempo di vita di una variabile

1. Variabili di classe automatica (Call stack)
2. Variabili di classe statica (Memoria statica)
   - campi dati statici (P2)
   - variabili globali (P1, almost deprecated)
   - variabili statiche in corpo di funzione (bad practice) per esempio una chiamata ricorsiva deve avere variabili condivise
3. Variabili dinamiche (Heap)

19_Distruttore

Distruttore

Metodo che viene invocato automaticamente quando l'oggetto viene distrutto.

È sempre disponibile il distruttore standard:

Invoca il "distruttore" per tutti i campi dati della classe nell'ordine inverso alla loro dichiarazione (nella classe).
Una volta che il distruttore ridefinito finisce viene invocato il distruttore standard!!!

Il distruttore viene invocato:

- Per gli **oggetti di classe statica**, al termine del programma(all'uscita dal main())
- Per gli **oggetti di classe automatica** definiti in un blocco (come una funzione), all'uscita del blocco in cui sono definiti. Parametri formali di una funzione
- Per gli **oggetti dinamici** (allocati sullo heap) quando viene invocato l'operatore **delete** su un puntatore ad essi.
- Per gli oggetti che sono **campi dati** di un oggetto x, quando x viene distrutto.
- Gli **oggetti con lo stesso tempo di vita**, tipicamente oggetti definiti nello stesso blocco, oppure oggetti statici di una classe, vengono distrutti nell'ordine inverso in cui sono stati creati.

Viene invocato in particolare:

1. Sui **parametri** di una funzione **passati per valore** all'uscita di una funzione.
2. Sulle **variabili locali** di una funzione all'uscita di una funzione.
3. **Sull'oggetto anonimo ritornato per valore** da una funzione non appena esso sia stato usato(Se qualche ottimizzazione prolunga il tempo di vita dell'oggetto anonimo ritornato significa che non è stato invocato il distruttore).

> Per clang e g++ l'ordine è 2, 3, 1. (locali, ritorno, parametri)

Comportamento:

Supponiamo che una **lista ordinata dei campi dati** di una classe C sia $x_1, \ldots, x_n$. Quando viene distrutto un oggetto di tipo C viene invocato in automatico il distruttore della classe C, standard o ridefinito nel modo seguente:

1. Viene eseguito, **se esiste**, il **corpo del distruttore della classe** C
2. Vengono **richiamati i distruttori per i campi dati nell'ordine inverso alla loro lista di dichiarazione**($x_n, \ldots, x_1$). Per campi dati (primitivi o derivati) viene rilasciata la memoria(distr. "standard" dei tipi non classe), mentre per i tipi classe viene invocato il distruttore standard oppure quello ridefinito.

> Il distruttore standard ha il corpo vuoto, quindi si limita a richiamare i distruttori per i campi dati di C in ordine inverso alla lista di dichiarazione

> In breve: Prima eseguo il corpo del distruttore, poi in ordine inverso rispetto all'ordine di dichiarazione chiamo i distruttori per i campi dati.

Posso anche implementare un distruttore per distruggere una lista intera senza necessità di definire un metodo distruggi, quindi in modo più elegante.

In nodo:

```
class C{
private:
        class nodo{
        public:
                nodo();
                nodo(const T&, nodo*);
                ~nodo();
                T info;
                nodo* next;
        };
        nodo* first;
public:
        ...
        ~C();
        ...
};

C::nodo::~nodo(){
        if(next) //posso anche non metterlo
        delete next;    //delete chiama il distruttore di next,
                                        //quindi vado di nodo in nodo in
modo ricorsivo
}

//e il distruttore di C senza il quale non inizierebbe nulla

C::~C() {
        if(first) delete first; //chiama distruttore di nodo che va poi
in modo
```

```
                                                    //ricorsivo su
tutti i nodi successivi a first
}
```

> Posso evitare di inserire il controllo per il nodo nullo nei disruttori di puntatori in quanto da standard C++0x (C++11) viene specificato "the value of the operand of delete may be a null pointer value."

Distruttore per classi derivate

Quando ho delle classi derivate, nel momento in cui sfrutto il polimorfismo e l'allocazione degli oggetti nella heap, incorro in un problema di garbage nel caso in cui si faccia la `delete` di un puntatore con tipo dinamico rappresentante una sottoclasse del tipo del puntatore. Ciò va a de-allocare la memoria del tipo statico del puntatore ma la parte implementata dalla classe derivata va a formare garbage.

```
D* pd = new D;
B* pb = pd;
delete pb; // garbage enorme, devo virtualizzare il distruttore
```

```
class B {
        ...
        virtual ~B();
        ...
};

class C : public B {
        ...
        ~C(); //C è virtuale, mantiene la dichiarazione virrtual fatta
dalla base
}
```

Distruttore virtuale puro

Implementato per la definizione di una classe astratta senza avere la necessità di dichiarare un metodo virtuale puro inutile.

```
class Base {
public:
        virtual ~Base() = 0; //=0 definisco distruttore virtuale puro
};

Base::~Base() {} //definizione del distruttore obbligatoria
```

```
int main(){
        Base b; //cannot declare b of abstract type Base
        Base* p = new Base; //cannot allocate an object of abstract type
base
}
```

20_Rule of three

Rule of Three

Quando una classe modifica uno o più dei seguenti probabilmente deve esplicitare i restanti:

- Distruttore
- Costruttore di copia
- Operatore di assegnazione

21_Information hiding

Nascondere ulteriormente la parte privata

La classe viene divisa in 2 pezzi, C_handle e C_privata, in questo modo si può avere una definizione completamente nascosta dei campi dati privati della classe C.

```
//file C_handle.h
class C_handle{
public:
        //parte pubblica
private:
        class C_private; //classe dichiarata in modo incompleto
        C_privata* punt; //puntatore a una classe dichiarata in modo
```

```
incompleto
};
```

```
//file C_handle.cpp
class C_handle::C_privata{
public: //altrimenti non vedo i campi dalla classe C_handle
        //parte privata nascosta dal source code
}
```

> Se nella parte C_privata non metto pubblici i campi allora dalla classe C_handle non ci potrò accedere.

> In questo modo potrei causare dei problemi di interferenza, devono essere quindi ridefiniti operatore di assegnazione, costruttore di copia e distruttore. Per esempio come in metodi copia e distruggi.

Nota:

In una classe C se viene dichiarata un'amicizia con la classe D, questa funge anche da dichiarazione incompleta di D.

Attenzione:

> Rischio di interferenze tra puntatori sullo heap. Necessario ridefinire CC, Distruttore e Assegnazione. Rule of Three

---

22_Relazione di amicizia

Relazioni di amicizia

Premessa:

Essere cauti con l'utilizzo di `friend`, utilizzarlo solo nel caso non si possa farne a meno.

> L'utilizzo di friend va contro il principio dell' incapsulamento.

In c++ una funzione definita amica di una classe, ha la **possibilità di accesso alla parte privata e protetta della classe** di cui è stata dichiarata amica.

Vedi ridefinizione operatore output.

Amicizie in un template

Friend non template

- Template di classe C con classe o funzione friend non template.

Attenzione!

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

```
class A { ... int fun(); ... };

template<class T>
class C {
        friend int A::fun();
        friend class B;
        friend bool test();
}
```

Friend associato

- Template di classe C con template di classe A o funzione friend associato

```
template<class T>
class C {
private:
        T t;
public:
        C(const T&);
        friend void f_friend<T>(const C<T>&);
        //amicizia associata al tipo di istanziazione T
};

template<class T>
C<T>::C(const T& x) : t(x) {}

template<class T>
```

```
void f_friend(const C<T>& c) {
        cout << c.t << endl; //per amicizia
}

int main(){
        C<int> c1(1); C<double> c2(2.5);
        f_friend(c1); //stampa 1
        f_friend(c2); //stampa 2.5
}
```

Nota

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

Friend non associato

- Template di classe C con template di classe o di funzione friend non associato

```
template<class T>
class C {
        template<class Tp>        //amicizia con
        friend int A<Tp>::fun();  //template di metodo

        template<class Tp>        //amicizia con
        friend class B;           //template di classe

        template<class Tp>        //amicizia con
        friend bool test(C<Tp>);  //template di funzione
};
```

Esempio

```
template <class T>
class C {

        template <class V>
        friend void fun(const C<V>&);

private:
        T t;
public:
        C(const T& y ) : x(y) {}
};

template <class T>
```

```
void fun(const C<T>& t) {
        cout << t.x << " "; //per amicizia associata
        C<double> c(3.1);
        cout << c.x << endl; //per amicizia NON associata
}

int main() {
        C<int> c(4);
        C<string> s("pippo");
        fun(c); //stampa 4 3.1 istanziaz. implicita
        fun(s); //stamoa pippo 3.1 istanziaz. implicita
}
```

Amicizia per ereditarietà

L'amicizia non si eredita!

23_Classi annidate

Classi annidate

Ci sono delle differenze sostanziali tra i diversi standard sulle regole tra classi annidate:

Da C++11:

Una classe annidata è membro della stessa, perciò ha gli stessi permessi di accesso di ogni altro membro.

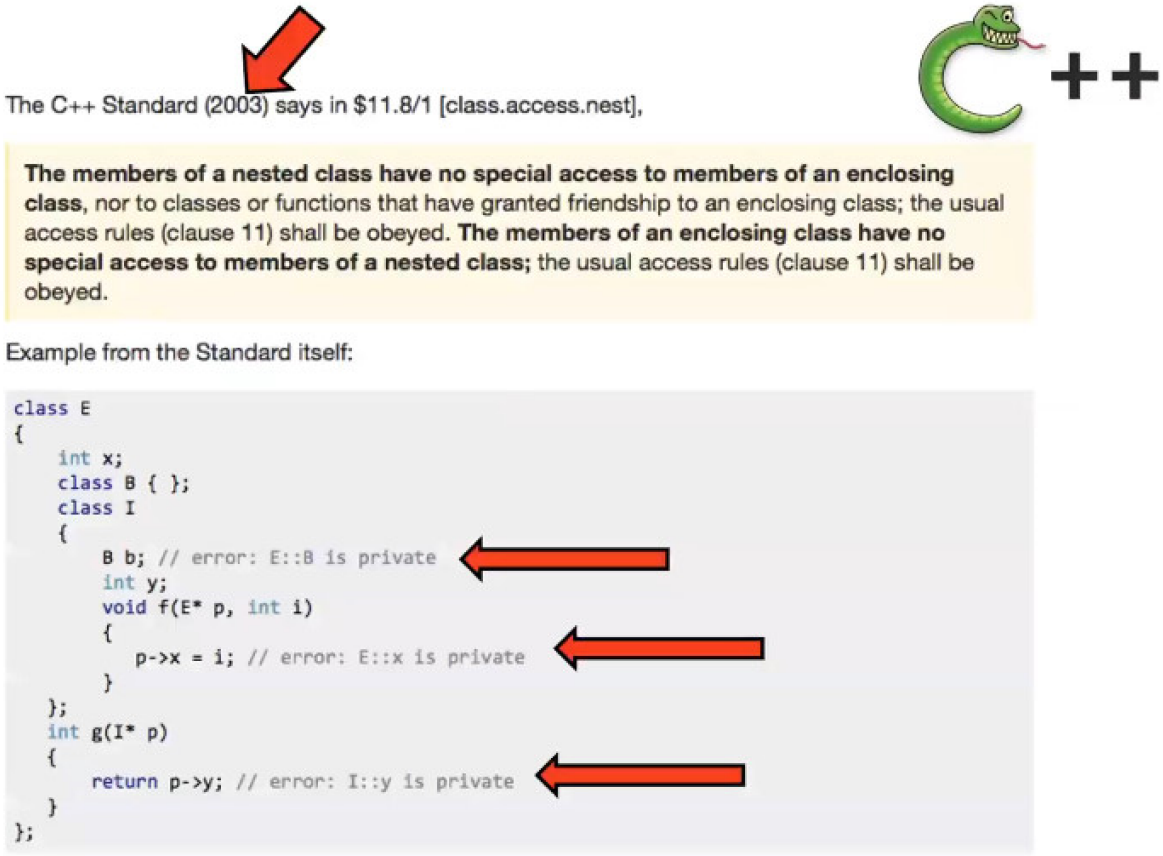
```
class E{
        int x;
        class B {};

        class I{
        B b;                            //OK: E::I può accedere E::B
        int y;
        void f(E* p, int i) {
                p->x = i;               //OK: E::I può accedere E::x
                }
        };

        int g(I* p) {
```

```
                return p->y;    //error:: I::y is private, g non può
accedere alla parte
                                                //privata di I, in
quanto g non è friend e non è metodo
                                                //di I
        }
};
```

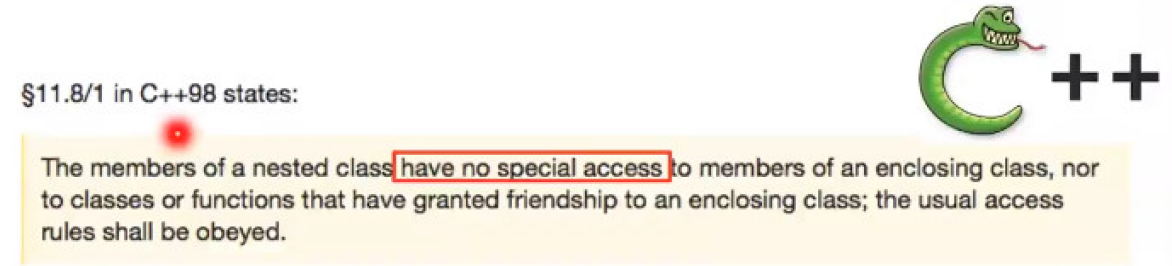
> La relazione di amicizia **non è simmetrica** e **non è transitiva**.

Per standard precedenti:

- Standard (2003):



- C++98



24_Iteratori

Iteratori

Funzionalità per scorrere e accedere a degli elementi di una collezione.

> Spesso strettamente collegato al container.

Nota:

> Non sono necessari costruttori per la classe iterator in quanto già presenti nella classe padre.

Esempio:

```
//file contenitore.h
class contenitore {
//friend class iteratore; per versioni c++ < c++03
private:
        class nodo{
        public:
                int info;
                nodo* next;
                nodo (int x, nodo* p) : info(x), next(p) {}
        };
        nodo* first;
public:
        class iteratore{
                friend class contenitore;
                private:
                        contenitore::nodo* punt; //nodo puntato
dall'iteratore
                public:
                        bool operator==(const iteratore& i) const {
                                return punt == i.punt;
                        }
                        bool operator!=(const iterarore& i) const {
                                return punt != i.punt;
                        }
                        iteratore& operator++() { //operator++ prefisso
                                if(punt) punt = punt->next;
                                return *this;
                        }
                        iteratore operator++(int) { //operator++
```

```
postfisso
                                iterator aux = *this;
                                if(punt) punt = punt->next;
                                return aux;
                        }
                };
        iteratore begin() const;
        iteratore end() const;
        int& operator[] (const iteratore&) const;

        contenitore(): first(0){}
        void aggiungi_nodo(int x) {first = new nodo(x, first); }
};
```

```
//fie contenitore.cpp
contenitore::iteratore contenitore::begin() const {
        iteratore aux;
        aux.punt = first; //per amicizia accedo a punt
        return aux;
}

contenitore::iteratore contenitore::end() const {
        iteratore aux;
        aux.punt = 0; //per amicizia
        return aux;
}

int& contenitore::operator[](const contenitore::iteratore& it) const{
        return it.punt->info; //per amicizia, nessun controllo su
it.punt
}
```

> operator[] ha come parametro di invocazione un oggetto di classe contenitore e passato per riferimento a costante un iteratore che è tra le parentesi quadrate.

Classe iteratore i cui oggetti rappresentano degli indici ai nodi della classe contenitore, è una classe annidata per poter accedere ai campi privati.

25_Smart pointer

Smart Pointer

Cosa sono?

Sono dei puntatori speciali i quali hanno un interfaccia pubblica che permette l'utilizzo degli smart pointer come puntatori ordinari ma presentano delle funzionalità aggiuntive, tipo gestione automatica della memoria (nessuna necessità di preoccuparsi per il junk) oppure il controllo dei limiti (bounds checking).

26_Conversioni di tipo

Conversioni di tipo

- Conversioni **implicite** (coercions)

  > Un espressione e si dice convertibile implicitamente a T sse T può essere inizializzato di copia da e:
  >
  > ```
  > T t=e; //e è convertibile implicitamente sse il programma compila
  > ```

- Conversioni **esplicite**
- Conversioni **predefinite dal linguaggio**
- Conversioni **definite dall'utente**
- Conversioni **con/senza perdita di informazione** (narrow/wide conversions)

Operatori di conversione esplicita

- `static_cast` permette una conversione narrow
- `const_cast` permette una conversione da const a non const
- `reinterpret_cast` reinterpreta a basso livello la sequenza di bit puntati dal puntatore (peicoloso)
- `dynamic_cast` utilizzato per il down-casting nelle classi derivate(polimorfe).

Possibili conversioni implicite

Conversioni implicite safe:

```
T& => T //non viceversa (non va bene int& x=5)
T[] => T* //int[2] a={3,1}; int* p = a;
T* => void* //puntatore generico: int* p=&x; void* q=p;
T => const T //int x=5; const int y=x;
const NPR => NPR //non puntatore o riferimento (in particolare C* const
=> C*)
                                    //const int x = 5; int y=x;
                                    //int* const p=&z; int* q=p;
T* => const T* //int* p=&x; const int* q=p;
T => const T& //int x=4; const int& r=x;

//tra tipi primitivi (conversioni senza perdita di info (wide))
bool => int
float => double => long double
char => short int => int => long
unsigned char => ... => unsigned long
```

Possibili conversioni esplicite

static_cast:

```
//(con perdita di informazioni) narrowing
double d = 3.14;
int x = static_cast<int>(d);
//wide conversion (coercion)
char c = 'a';
int x = static_cast<int>(c);
//conversione T* => void*
void* p;
p=&d;
//per la conversione di void* serve uno static_cast
double* q = static_cast<double*>(p)
```

const_cast:

```
const int i=5;
int* p = const_cast<int*> (&i);

void F(const C& x) {
        x.metodoCostante();
        const_cast<C&>(x).metodoNonCostante();
}
```

```
int j=7;
const int* q=&j; //OK, cast implicito
```

reinterpret_cast: (pericolo)

```
Classe c;
int* p = reinterpret_cast<int*>(&c);
const char* a = reinterpret_cast<const chat*>(&c);
string s(a);
cout << s;
```

27_Standard Library

Standard Library

Smart Pointers

Ne esistono di diversi tipi e implementano tutti quanti una gestione automatica sicura verso le eccezzioni della vita di un oggetto.

> - unique_ptr (c++11) Smart pointer con unico proprietario dell'oggetto
> - shared_ptr (c++11) Smart pointer con proprietari condivisi per l'oggetto
> - weak_ptr (c++11) Smart pointer con riferimento debole ad un oggetto gestito da shared_ptr

Boost library

Queue

Template da standard

Lo standard C++ prevede la definizione di:

- Template di classi collezione: contenitori.
  - Presentano ***tutti***:
    - il metodo `insert`, `size`, `empty` e `erase`
    - La possibilità di utilizzare `==`(ovviamente anche `!=`)

    - Per ***contenitori sequenziali*** inoltre:
        - I metodi `push_back`, `pop_back`, `front` e `back`
        - Gli operatori booleani `<` e `<=`
- Template di funzione: algoritmi generici.

Vector

Contenitore semplice ed efficiente presente nella STL, implementato come template di classe che generalizza template dinamici.

Caratteristiche

1. Supporta l'*accesso casuale in tempo costante*.
2. *Inserimento e rimozione in coda in tempo lineare* ammortizzato(in media).
3. *Inserimento e rimozione arbitraria in tempo lineare* ammortizzato(in media).
4. **Capacità variabile dinamicamente** (se pieno raddoppia).
5. **Gestione automatica della memoria**

Dichiarazione

```
template <class T, class Alloc = allocator<T>> class vector;
```

> Template di classe con due parametri di tipo di cui uno con parametro di default

Uso

Dichiarazione

- Alla c

```
int a[10]; //array
```

- Stile STL

```
vector<int> v(10); // costruttore ad 1 argomento vector(size_type) con costruzione di default per gli elementi
```

Costruttori

- `vector(size_type)` costruisce un vector i cui elementi sono inizializzati con il costruttore di default per il tipo del vector.

```
vector<int> v(10);
```

- `vector(size_type n, const T& t)` costruisce tutti gli elementi di copia a partire da un valore iniziale t (da C++11).

```
vector<int> v(10, 0);
```

- Sequenza di valori.

```
vector<int> v(5) = {-1, 5, -7, 0, 12};
```

- Template di costruttore, inizializzo vector con segmento di vector o array.

```
int main(){
        int ia[20];
        vector<int> iv(ia, ia+6);
        cout << iv.size() << endl; //size 6
        vector<int> iv2(iv.begin(), iv.end()-2);
        cout << iv2.size() << endl; //size 4
}
```

Operatori e metodi

- Operatore di indicizzazione operator`[]`;

```
int n = 5;
vector<int> v(n);
int a[n] = {2,4,5,2,-2};
for (int i=0; i<n; i++)
        v[i] = a[i] + 1;
```

- Metodo `size()` e `capacity()`, $v.size() <= v.capacity()$ sempre, size dice il numero di elementi e capacity il numero contenibile prima di un'espansione;

```
vector<string> v(10), w;
cout << v.size() << " " << v.capacity(); //10 10
vector<string> u(v); //costruzione di copia
w = u;               //assegnazione
```

- `empty()` disponibile per tutti i contenitori STL, true sse il contenitore su cui è stato chiamato il metodo è vuoto;
- `void push_back(const T&)` inserimento in coda attraverso costruttore di copia;
- `void pop_back()` rimuove l'ultimo elemento;
- `iterator insert(iterator position, const T& val)`
  - Inserire un singolo elemento in posizione fornita da un iteratore.
- `void insert(iterator position, size_type n, const T& val)`
  - Inserire n volte un elemento a partire dalla posizione fornita da un iterator.
- Template `template<class InputIterator> void insert (iterator position, InputIterator first, InputIterator last)`
  - Inserire a partire dalla posizione fornita da un iterator una parte di vettore compatibile dalla posizione puntata dal primo iterator alla posizione precedente quella puntata dal secondo.
- `iterator erase(iterator position)` oppure `iterator erase(iterator first, iterator last)` cancellano l'elemento puntato da position, oppure cancellano gli elementi compresi tra first e last escluso;
- `T& front()` accedo al primo elemento sia per const che non;
- `T& back()` accedo all'ultimo elemento sia per const che non;
- `iterator begin()` e `iterator end()` metodi applicabili su un vector tornano un iteratore che punta al primo o ultimo elemento di un contenitore STL;

```
vector<string> s;
vector<string>::iterator it;
for (it = s.begin(); it!=s.end(); it++)
        cout << *it << endl;
```

Classi di vector

- `vector<T>::iterator` e `vector<T>::const_iterator` si usa `iterator` se si necessita l'accesso agli elementi come lvalue, mentre `const_iterator` se si deve fare un accesso in sola lettura (solo rvalue).
  - su `vector<T>` è possibile usare `iterator begin()` e `iterator end()` metodi che tornano un iteratore
  - su un iteratore in STL posso sempre usare `*`, `++` pre e post-fisso, `--` pre e post-fisso e `==`.

- Per un iteratore di vector, list e deque posso anche andare avanti e indietro di un numero arbitrario di elementi

```
vector<T>::iterator i, j;
int k;
i += k;
i -= k;
j = i+k;
j = i-k;
i < j;
i > j;
i <= j;
i >= j;
```

> **NB:** Un `const vector<T>` necessita di un const_iterator altrimenti da errore

List

Implementata come una lista doppiamente linkata.

Pro/Contro

Meno efficiente di vector ma permette di eseguire in tempo costante (O(1)) operazioni di inserimento e rimozione in posizione arbitraria.

Deque

È una Double Ended Queue e ciò permette di unire dei vantaggi di lista e vector, solitamente implementato come buffer circolare o più array dinamici di dimensione fissa in cui ogni elemento è individuato da una chiave, ordinate nell'insieme.

Vantaggi

1. Accesso indicizzato efficiente per lettura e scrittura.
2. Inserimento ed eliminazione agli estremi efficienti come per una list.

Set

Insieme contenente un occorrenza per ogni valore nel set, chiave rappresentata dal valore dell'elemento.

MultiSet

Chiave data dal valore dell'elemento ma possibili occorrenze multiple dell'elemento.

Map

Elementi formati da coppia chiave valore, per accesso e valore associato alla chiave. Le chiavi sono distinte.

MultiMap

Come la Map ma può contenere più elementi con la stessa chiave.

28_Template di funzione

Template di funzione

A cosa servono?

Servono per implementare la programmazione generica, dove un algoritmo viene scritto sulla base di tipi che verranno specificati più tardi e che saranno inizializzati conseguentemente.

Cosa sono?

Sono una caratteristica del linguaggio C++ che permette a funzioni e classi di lavorare con tipi generici. In questo modo si permette ad una funzione o classe di lavorare su molti tipi differenti senza la necessità di riscriverla ogni volta per ciascun tipo.

Parametri di un template

- **Parametri di tipo**: si possono istanziare con un tipo qualsiasi
- **Parametri valore** di qualche tipo: si possono istanziare con un valore costante del tipo indicato

> **Un template non è codice compilabile**: istanziazione **implicita** o **esplicita** di un template di funzione

Istanziazione implicita

> **Processo di *deduzione degli argomenti*** di un template nella ***istanziazione implicita***, dove **il tipo di ritorno non si considera mai** (perché posso anche non utilizzare il tipo di ritorno)

```
template <class T >
T min(T a , T b) {
        return a<b? a : b;
}
double d; int i, j;
...
d=min(i,j); //istanzia int min(int,int) e usa la conversione implicita
int=>double
```

L'**algoritmo di deduzione degli argomenti di un template** esamina tutti i parametri attuali passati al template di funzione da sinistra verso destra. Se si trova uno stesso parametro T del template che appare più volte come parametro di tipo, l'argomento del template dedotto per T da ogni parametro attuale deve essere esattamente lo stesso.

```
int main (){
        int i; double d,e;
        ...
        e = min(d,i); //non compila, si deducono 2 diversi argomenti del
template, double e int
}
```

Per l'**istanziazione implicita** sono **ammesse solo 4 tipologie di conversioni** dal tipo dell'argomento attuale al tipo dei parametri del template:

1. Conversione da lvalue a rvalue, T& =>T
2. Da array a puntatore, T[]=>T*
3. Conversione di qualificazione costante, T=>const T
4. Conversione da rvalue a riferimento costante, rvalue di tipo T=>const T&

```
template <class T> void E(T x) {...};
template <class T> void F(T* p) {...};
template <class T> void G(const T x) {...};
template <class T> void H(const T& y) {...};

int main(){
        int i=6; int& x=i;
        int a[3] = {4,2,9};
        E(x);  // 1: istanzia void E(int)
        F(a);  // 2: istanzia void F(int*)
        G(i);  // 3: istanzia void G(const int)
        H(7);  // 4: istanzia void H(const int&)
}
```

Instanziazione esplicita

Per l'**istanziazione esplicita** sono ammesse **tutte le conversioni implicite di tipo** per i parametri attuali del template di funzione

```
int main() {
        int i; double d, e;
        e = min<double>(d,i); //compila, conversione implicita
int=>double
}
```

Parametri valore

Se un **parametro di una funzione è un riferimento ad un array statico**, la dimensione dell'array è parte integrante del tipo del parametro.

```
int min(int (&a) [3]){...}

int ar[4] = {4, 2, 5, 7};
cout << min(ar); // non compila
```

```
template <class T, int size> //parametro di tipo + parametro valore
T min(T (&a) [size]) {...}

int main(){
        int ia[20];
        orario ta[50];
        ...
        cout << min(ia);
        cout << min(ta);
        //oppure
        cout << min<int, 20>(ia);
        cout << min<orario, 50>(ta);
}
```

Modello di compilazione del template

1. La definizione del template deve sempre essere visibile all'utilizzatore del template
2. Il file header del template deve includere dichiarazione e definizione (No information hiding).
3. Per evitare istanze multiple di template istanziati allo stesso tipo faccio la dichiarazione esplicita di istanziazione del template di funzione ad un certo tipo e utilizzo la flag `-fno-implicit-templates` che non fa produrre codice al compilatore a meno che non sia una istanziazione esplicita del template.

```
//file min.h
template <class T>
T min(T a, T b){
        return a<b ? a : b;
}
```

Dichiaro esplicitamente un istanziazione a int e a orario del template sopra citato in modo da forzare il compilatore a generare il codice dell'istanza del template relativa al tipo *int* e al tipo *orario*.

```
//file usedTemplates.cpp
#include "min.h"
template int min(int, int);
template orario min(orario, orario);
```

```
//file main.cpp
#include "min.h"
#include "orario.h"
int main(){
        cout << min(9,3) << min(3*16, 50-2);
        cout << min(orario(4), orario(4,5,6));
}
```

```
g++ -fno-implicit-templates -c main.cpp
g++ -fno-implicit-templates -c usedTemplates.cpp
g++ main.o usedTemplates.o
```

29_Template di classe

Template di classe

Solo istanziazione esplicita

Nel caso si vogliano utilizzare delle strutture fisse per dei tipi diversi risulta necessario scrivere due definizioni distinte della classe con nomi diversi.

Parametri template di classe

- Parametri di tipo.
- Parametri valore.
- Parametri tipo/valore con valore di default.

```
template <class tipo = int, int siza = 1024>
class Buffer {
        ...
};
```

```
Buffer<> ib; //Buffer<int, 1024>
Buffer<string> sb; //Buffer<string, 1024>
Buffer<string, 500> sbs; //Buffer<string, 500>
```

Dichiarazione o definizione di un template

Nella dichiarazione o implementazione del template possono comparire sia nomi di istanze di template di classe sia nomi di template di classe

```
template <class T>
int fun (Queue<T>& qT, Queue<string> qs); //template di classe e istanza
del template di classe
```

Definizione esterna di un metodo di un template di classe

```
template <class T>
class Queue {
        ...
public:
        Queue();
        ...
};

//definizione esterna
template <class T>
Queue<T>::Queue() : primo(0), ultimo (0) {}
```

Istanziazione

Un'istanza viene generata solo quando è necessario.

Istanziazione non necessaria

```
template <class T> class Queue; //dichiarazione incompleta

void Stampa(cunst Queue<int>& q){
        Queue<int>* qpi = const_cast< Queue<int>* >(&q); //non viene
fatta nessuna istanziazione in quanto non è necessaria la
rappresentazione dell'oggetto per il riferimento o per il puntatore.
}
```

Istanziazione necessaria

```
template <class T> class Queue {
        //definizione della classe necessaria
};

void Stampa(Queue<int> q){ //per valore, costruttore di copia istanziato
        Queue<int> qi; //si genera l'istanza Queue<int> per la
costruzione di default
}
```

```
template <class T> class Queue {
        //definizione della classe necessaria
};

void Stampa(Queue<int> q){ //per valore, costruttore di copia istanziato
        Queue<int> pqi = const_cast< Queue<int>* >(&q);
        pqi++; //deve istanziare Queue<int> in quanto deve calcolare la
sizeof(Queue<int>) di cui occorre incrementare il puntatore pqi
        ...
}
```

Esempio

queue per int e string.

```
class QueueInt{
public:
        Queue();
        ~Queue();
        bool empty() const;
        void add(const int&);
        int remove();
private:
```

```
        ...
};
```

```
class QueueString{
public:
        Queue();
        ~Queue();
        bool empty() const;
        void add(const string&);
        string remove();
private:
        ...
};
```

Posso templatizzare

```
template <class T>
class QueueItem{
public:
        QueueItem(const T&);
        T info;
        QueueItem* next;
};

template <class T>
class Queue{
public:
        Queue(); //non Queue<T>
        ~Queue();
        bool empty() const;
        void add(const T&);
        T remove();
private:
        QueueItem<T>* primo;
        QueueItem<T>* ultimo;
};
```

Così posso dichiararne di tipi diversi con ***inizializzazione esplicita***

```
Queue<int> qi;
Queue<bolletta> qb;
Queue<string> qs;
```

Posso rendere private la parte di QueueItem

```
template <class T>
class QueueItem{
        friend ostream& operator<< <T> (ostream&, const Queue<T>&);
        friend class Queue<T>; //relaz di amicizia associata per
privatizzare
private:
        QueueItem(const T&);
        T info;
        QueueItem* next;
};
```

Ed implementare per esempio l'operatore di output

```
template <class T>
ostream& operator<<(ostream& os, const Queue<T>& q) {
        os << "(";
        QueueItem<T>* p = q.primo; //amicizia Queue
        for (; p!=0; p=p->next)    //amicizia QueueItem (come sopra)
                os << *p << " ";       //operator<< per QueueItem (da
fare)
        os << ")" << endl;
        return os;
}
```

Campi dati statici

Un campo dati statico istanziato in un template di classe ha valore comune tra tutti gli oggetti di una certa istanza della classe

```
template <class T>
class Queue {
private:
        static int contatore;
}
```

I campi statici vengono inizializzati esternamente con:

```
template <class T>
int Queue<T>::contatore = 0; //inizializzo il campo dati statico
```

NB

> Un campo dati statico non effettivamente utilizzato non viene istanziato quindi non inizializzato dalla definizione del template.

```
class A{
public:
        A(int x=0) {cout << x << "A() ";}
};

template<class T>
class C {
public:
        static A s;
};
template<class T>
A C<T>::s=A(); //stampa 0A()

int main() {
        C<double> c;
        C<int> d;
        C<int>::s = A(2);
}

//stampa: 0A() 2A() non 0A() 0A() 2A() perchè non uso lo static di
C<double>
```

Classi annidate in template di classe

Template di classe annidato associato

Posso annidare QueueItem in Queue come segue

```
template <class T>
class Queue{
private:
        //template di classe annidato associato all'istanza T
        class QueueItem {
        public:
                QueueItem(const T& val);
                T info;
                QueueItem* next;
        };
        ...
}
```

> QueueItem<T> è detto tipo implicito, perché non è un tipo completamente definito ma dipende implicitamente dai parametri di tipo di Queue<T>

Tipi e template impliciti in template di classe

```
template <class T>
class C {
public:
        class X {}; //template di classe annidata associato potrebbe
utilizzare il parametro di tipo T

        class D {
        public:
                T x;
        };          //template di classe annidata associato, usa il
parametro di tipo T

        template <class U>
        class E {
        public:
                T x;
                U y;
                void fun1() {return;}
        }; // template di classe annidata non associato, usa T del
template contenitore e usa il suo parametro di tipo U

        template <class U>
        void fun2() {
                T x; U y;  return;
        } // template di metodo di istanza non associato
}; // i due class U sono tipi differenti relativi alla classe E e al
metodo fun2


template <class T>
//C<T>::D è un uso di un tipo che dipende effettivamente dal parametro T
void templateFun(typename C<T>::D d){

        //C<T>::X è un uso di un tipo che dipende comunque dal parametro
T
        typename C<T>::X x;

        //C<T>::D è un uso di un tipo che dipende effettivamente dal
parametro T
        typename C<T>::D d2 = d;

        // (1) E<int> è un uso del template di classe annidata che
dipende dal parametro T
```

```
        // (2) C<T>::E<int> è un uso di un tipo che dipende dal
parametro T
        typename C<T>::template E<int> e;
        e.fun1();

        //c.fun2<int> è un uso del template di funzione che dipende dal
parametro T
        C<T> c;
        c.template fun2<int>();
}
```

30_Ereditarietà

Ereditarietà

Uso

1. Estensione
2. Specializzazione
3. Ridefinizione
4. Riutilizzo di codice

Terminologia

Classe Base B e classe derivata D
Sottoclasse D e super-classe B
Sottotipo D e super-tipo B

Relazione is-a

Induce il subtyping caratteristica fondamentale dell'ereditarietà.

Subtyping

Ogni oggetto della classe derivata è utilizzabile come oggetto della classe base.

Subtyping: $Sottotipo\ D \Rightarrow Supertipo\ B$

Per oggetti

$D \Rightarrow B$ estrae il sotto-oggetto, ovvero prendo il super-tipo B scartando la nuova parte implementata da D.

Trasformo oggetto dataora in orario con costruttore di copia di orario che estrae il sotto-oggeto ($dataora \Rightarrow orario$)

```
class dataora : public orario {};

int F(orario o) {...}
dataora d;
int i = F(d);
```

Per puntatori e riferimenti

Creo puntatori e riferimenti polimorfi:

- Copio tutto l'indirizzo dell'oggetto, andando a visualizzare l'oggetto con informazione specifica del sottotipo come il suo super-tipo (non perdendo informazioni).

  - $D^* \Rightarrow B^*$

```
class D : public B {};

D d; B b;
D* pd= &d;
B* pb= &b;
pb = pd;  //conversione D* => B*
```

  - $D\& \Rightarrow B\&$

Attenzione:

> Se ho puntatori o riferimenti con tipo dinamico diverso dal tipo statico devo ridefinire il distruttore! Altrimenti rischio che per oggetti sulla heap puntati ad esempio da un puntatore con tipo statico uguale alla base e tipo dinamico uguale ad una delle sue classi derivate, crea garbage in quanto la delete elimina solo il sotto-oggetto senza eliminare il resto della classe derivata.

Tipi

Tipo statico

- Ottenuto al momento della dichiarazione del puntatore o del riferimento (tipo T*/& a cui ho dichiarato puntatore/riferimento).

- Uguale nel tempo.

Tipo dinamico

- Tipo dell'oggetto puntato/riferito in un dato istante **a tempo di esecuzione**.

Interfaccia per subtyping

Derivazioni protette e private non supportano l'ereditarietà di tipo (subtyping)
la relazione is-a è supportata solo dalla derivazione pubblica

Un membro protetto è accessibile dalla classe ma non dall'esterno (idea: un membro privato ereditabile). Posso accedere ai campi protected dall'esterno se li richiamo dal sotto-oggetto.

```
class B {
protected:
        int i;
        void protectedPrintB() const {cout << ' ' << i;}
public:
        void printB() const {cout << ' ' << i;}
};

class D : public B{
        double z;
public:
        stampa() {
                cout << i << ' ' << z; //OK
        }


        static void stampa(const B& b, const D& d){
                cout << ' ' > b.i; //Illegale B::i è protetto (sono
nella classe D non nella classe B, è come se accedessi dall'esterno)
                b.printB(); //OK
                b.protectedPrintB(); //Illegale B::protectedPrintB() è
protetto

                cout << ' ' << d.i; //OK i è membro protetto del sotto-
oggetto ereditato
                d.printB(); //OK
                d.protectedPrintB(); //OK
        }
};
```

L'ereditarietà privata si usa quando si vuole ereditare l'implementazione di una classe ma non la sua interfaccia. Ovvero eredito solo dei metodi che posso utilizzare sul sotto-oggetto ereditato

Classe Base con derivazione

- Pubblica
  - Membri pubblici della base rimangono pubblici
  - Membri protetti rimangono protetti
  - Membri privati rimangono inaccessibili
- Protetta
  - Membri pubblici diventano protetti
  - Membri protetti lo rimangono
  - Membri privati rimangono inaccessibili
- Privata
  - Tutti i membri della base diventano privati

I membri protected violano l'information hiding.

Ereditarietà privata vs relazione di composizione has-a

Relazione has-a

```
class Motore {
private:
  int numCilindri;
public:
  Motore(int nc): numCilindri(nc) {}
  int getCilindri() const {return numCilindri;}
  void accendi() const {
    cout << "Motore a " << getCilindri() << " cilindri acceso" << endl;}
};

Class Auto {
private:
  Motore mot; // Auto has-a Motore come campo dati
public:
  Auto(int nc = 4): mot(4) {}
  void accendi() const {
    mot.accendi();
    cout << "Auto con motore a " << mot.getCilindri() << " cilindri accesa" << endl;
  }
};
```

Ereditarietà privata

```
class Motore {
private:
  int numCilindri;
public:
  Motore(int nc): numCilindri(nc) {}
  int getCilindri() const {return numCilindri;}
  void accendi() const {
    cout << "Motore a " << getCilindri() << " cilindri acceso" << endl;}
};

Class Auto: private Motore { // Auto has-a Motore come sottoogetto
public:
```

```
  Auto(int nc = 4): Motore(nc) {}
  void accendi() const {
    Motore::accendi();
    cout << "Auto con motore a " << getCilindri() << " cilindri accesa" << endl;
  }
};
```

Ereditarietà privata vs relazione di composizione has-a

Similarità

1) In entrambi i casi un oggetto Motore "contenuto" in ogni oggetto Auto

2) In entrambi i casi, per gli utenti esterni, Auto* non è convertibile a Motore*

Differenze

1) La composizione è necessaria se servono **più motori** in un auto (a meno di usi limite di ereditarietà multipla)

2) Ered.privata può introdurre **ereditarietà multipla** (problematica) **non necessaria**

3) Ered.privata **permette ad Auto** di convertire Auto* a Motore*

4) Ered.privata permette **l'accesso alla parte protetta** della base

Recap

```
class C {
public:
        void f() {cout << "C::f\n";}
};

class D: public C {
public:
        void f() {cout << "D::f\n";}
};

class E: public D {
```

```
public:
        void f() {cout << "E::f\n";}
};

int main() {
        C c; D d; E e;
        C* pc = &c; E* pe = &e;
        c = d; //OK D => C
        c = e; //OK E => C
        d = e; //OK E => D
        C& rc=d; //OK D => C
        D& rd=e; //OK E => D
        pc->f(); //OK
        pc = pe; //OK E* => C*
        rd.f(); //OK
        c.f(); //OK
        pc->f(); //OK
}
```

31_Name Hiding Rule e ridefinizione di un metodo

Name hiding rule e ridefinizione di un metodo

Ridefinizione di un metodo

Per i metodi ereditati attraverso una relazione di sub-typing se il ***metodo viene ridefinito*** allora si perdono tutte le versioni sovraccaricate del metodo disponibili nella super-classe. Diventano quindi in seguito a una ridefinizione del metodo accessibili attraverso l'operatore di scoping `::`

```
class A {
public:
        void met(){cout << "metA " << endl; }
};

class B : public A {
public:
        void met(int) { cout << "metB " << endl; }
};

int main() {
        B b;
```

```
        b.met(); //ilegale, non ho più il metodo met() definito in A ma
solo la sua ridefinizione
        b.A::met(); //OK
}
```

```
class B {
protected:
        int x;
public:
        B() : x(2) {}
        void print() {cout << x << endl; }
};

class D: public B{
private:
        double x; //ridefinzione del campo dati x
public:
        D() : x(3.14) {}
        //ridefinisco print()
        void print() { cout << x << endl; } // D::x
        void printAll() { cout << B::x << " " << x << endl; }
};

int main() {
        B b; D d;
        b.print(); //stama 2
        d.pirnt(); //stampa 3.14
        d.printAll(); //stampa 2 3.14
}
```

```
class C {
public:
        void f(int x) {}
        void f() {}
};

class D: public C{
private:
        int y;
public:
        void f(int x) { f(); y=3+x; } //illegale "no matching function
for D::f()" se uso "C::f()" invece va bene
};
```

32_Overriding metodi virtuali

Overriding metodi virtuali

> Una volta dichiarato un metodo come `virtual` non è più necessario nelle classi derivate specificare la keyword.

```
class A {
        ...
        virtual void metodo(); //metodo virtuale
        ...
};
```

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.

NB:

> Se nome e parametri sono uguali al metodo di cui fare l'override ma il tipo di ritorno cambia, il compilatore torna un errore.

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

Overriding con tipo di ritorno covariante

```
virtual T1* m(...); // T2 <= T1 (T2 sottoclasse)

T2* m(...)
```

> Non è possibile cambiare tipo di ritorno di un metodo virtuale se non con un tipo sottoclasse di un altro.

```
class B {
public:
        virtual int f() { cout << "B::f()\n"; return 1; }
        virtual void f(string s) { cout << "B::f(string)\n"; }
        virtual void g() { cout << "B::g()\n"; }
};

class D1 : public B {
public:
        void g() { cout << "D1::g()\n"; } //overriding metodo virtuale
non sovraccaricato
};

class D2 : public B {
public:
        int f() { cout << "D2::f()\n"; return 2; } //overriding metodo
virtuale sovraccaricato
};

class D3 : public B {
public:
        //!void f() { cout << "D3::f()\n"; } //NON COMPILA non posso
modificare il ripo di ritorno
};

class D4 : public B {
public:
        int f(int) { cout << "D4::f()\n"; return 4; } //lista degli
argomenti modificata, ridefinizione non overriding
};
```

```
class X {};
class Y : public X {};
class Z : public X {};

class B {
        X x;
public:
        virtual X* m() { cout << "B::m() "; return &x; }
};

class C : public B{
        Y y;
public:
        virtual X* m() { cout << "B::m() "; return &y; } //overriding
che non arricchisce il tipo di ritorno
};
```

```
class D : public B {
        Z z;
public:
        virtual Z* m() { cout << "B::m() "; return &z; } //OK overriding
covariante legale Z <= X
};

int main () {
        B b; C c; D d; B* pb = &b; X x;
        Y* py = c.m(); //illegale sto assegnando una superclasse ad una
sottoclasse, impossibile senza dynamic cast
        X* px = c.m();
        Z* pz = d.m(); //z ha arricchito il tipo di ritorno
dell'overriding di m() con tipo di ritorno Z*
        x = *(pb->m());
        pb = &d; x = *(pb->m()); //OK sottotipo
}
```

```
class B {
public:
        virtual void m(int x=0) {cout << "B::m01 ";}//il valore di
default si mantiene lungo tutta la gerarchia di tipi
};

class D : public B {
public:
        virtual void m(int x) {cout << "D::m01 ";} //x eredita il valore
di default dalla base
        virtual void m() {cout << "D::m() ";} //nuovo metodo di D non è
un overriding di B::m
};

int main() {
        B* p = new D;
        p->m(2); //stampa D::m01 e non B::m01 perchè fa il binding con
il prototipo della funzione (void m(int x=0)) ma p avendo tipo dinamico
D viene usata la funzione D::m(int)
        p->m(); //srampoa D::m01 e non D::m() perchè la funzione che fa
l'override in D ha valore di default
        p->B::m(); //stampa B::m01 e nn D::m01 in quanto blocco il late
binging con operatore di scooping
}
```

Late binding

Anche detto binding ritardato o Dynamic binding oppure dynamic lookup è l'azione di verifica che compie il compilatore a runtime per assegnare il corretto l'override di un metodo in base al tipo che lo invoca.

Questo viene implementato grazie ad un membro nascosto incluso dal compilatore nelle classi che definiscono un metodo virtuale. Il membro è un puntatore alla Virtual Method Table (VMT) che viene usata a runtime insieme al puntatore per invocare la corretta implementazione di funzione.

33_Classi astratte

Classi astratte

Una classe astratta è una classe che non può essere istanziata in quanto è ***marcata come astratta*** **oppure** ***specifica metodi astratti*** (o virtuali).
Una classe che implementa solo metodi astratti è chiamata Classe Base Astratta Pura (Pure ABC or interface in C++).
Una classe concreta è una classe che può essere istanziata.

Classe base astratta

1. Almeno un metodo virtuale puro (dichiarazione virtual con `= 0` prima del `;`).
2. Non si possono costruire oggetti.

```
class B {
public:
        virtual void f() = 0; //metodo vistuale puro
};

class C : public B { //sottoclasse astratta
public:
        void g() {cout  << "G::g() ";} //non conclude il contratto
astratto di B
};

class D : public B { //sottoclasse concreta
public:
        virtual void f() {cout << "D::f() ";}
};

int main() {
        //C c; //Illegale "cannot declare c of type C..."
        D d; //OK D è concreta
        B* p; // OK puntatore a classe astratta
        p = &d; //puntatore (super)polimorfo
```

```
        p->f(); //stampa D::f()
}
```

NB:

> Se voglio creare una classe base astratta senza definire inutilmente un metodo virtuale puro che non mi serve, posso definire un distruttore virtuale puro

---

34_RTTI

Run-Time Type Information (RTTI)

> **TD(ptr)** tipo dinamico di puntatore polimorfo ptr
> **TD(ref)** tipo dinamico di riferimento polimorfo ref

Operatori

Operatore `typeid`

Permette di determinare il tipo di una espressione qualsiasi a tempo di esecuzione. Ritorna un tipo `type_info`.

1. Se l'espressione operando di `typeid` è un ***riferimento*** ref **ad una classe che contiene** almeno **un metodo virtuale**, cioè una ***classe polimorfa***, allora `typeid` ritorna un oggetto `type_info` che rappresenta il tipo dinamico di ref.
2. Se l'espressione operando di `typeid` è un ***puntatore de-referenziato*** $*punt$ dove $punt$ è un ***puntatore ad un tipo polimorfo***, allora `typeid` ritorna un oggetto `type_info` che rappresenta il tipo T dove T^* è il tipo dinamico di $punt$.

NB:

> - 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.**

Nota:

> Necessario includere typeinfo `#inclde <typeinfo>`

Metodi:

- `bool operator==(const type_info&) const;`
- `bool operator!=(const type_info&) const;
- `const char* name() const;`

```
#include <typeinfo>
#include <iostream>

int main() {
        int i=5;
        std::cout << typeid(i).name() << endl; //stampa: i(nt)
        std::cout << typeid(3.14).name << endl; //stampa: d(ouble)
        if(typeid(i) == typeid(int)) std::cout << "Yes";
}
```

```
class A { public: virtual ~A() {} };
class B : public A {};
class D : public B {};

int main() {
        B b; D d;
        B& rb = d;
        A* pa = &b;
        if(typeid(rb) == typeid(B)) std::cout << "1";
        if(typeid(rb) == typeid(D)) std::cout << "2";
        if(typeid(*pa) == typeid(A)) std::cout << "3";
        if(typeid(*pa) == typeid(B)) std::cout << "4";
        if(typeid(*pa) == typeid(D)) std::cout << "5";
} // stampa 24
```

35_Downcasting

Downcasting

Utilizzato ***per tipi polimorfi*** B, $D \leq B$ ovvero D sottoclasse di B e ha almeno un metodo virtuale.

Il downcast consiste nella conversione per puntatori e riferimenti

- $B^* \Rightarrow D^*$
  - `dynamic_cast<D*>(p) != 0` $\Leftrightarrow TD(p) \leq D^*$ ovvero, il `dynamic_cast` torna un valore diverso da `nullptr` se e solo se il ***tipo dinamico di*** p ***è una sottoclasse di*** D ***oppure*** D ***stessa*** (ovvero è compatibile con il tipo D).
- $B\& \Rightarrow D\&$
  - Il `dynamic_cast` **se fallisce su un riferimento torna un'eccezione** di tipo `std::bad_cast` **di cui fare il catch** ed eseguire un'azione nel caso succeda.

```
class X { public: virtual ~X() {} };
class B { public: virtual ~B() {} };
class D : public B {};

#include <typeinfo>
#include <iostream>

int main() {
        D d;
        B& b = d; //upcast
        try {
                x& xr = dynamic_cast<X&>(b);
        } catch(std::bad_cast e){
                std::cout << "Cast fallito!" << std::endl;
        }
}
```

```
class B {
public:
        virtual void m();
};

class D : public B {
public:
        virtual void f(); //nuovo metodo virtuale
};

class E : public D {
        void g();
};

B* fun() { /*può ritornare B*, D*, E*, ...*/} // I primi 3 adesso, il
resto lo posso tornare nel caso in cui estendessi la gerarchia

int main() {
```

```
        B* p = fun();
        if(dynamic_cast<D*>(p)) (static_cast<D*>(p))->f(); //check sulla
fattibilità del downcast e poi cast statico alla sottoclasse (downcast)
        E* q = dynamic_cast<E*>(p); //assegno il valore ritornato dal
dynamic_cast ad un puntatore
        if(q) q->g(); //se il puntatore non è nullo allora chiamo la
funzione della sottoclasse a cui ho downcastato
}
```

Downcasting vs metodi virtuali

- Usare il **downcasting solo quando necessario**.
- Non fare type checking dinamico inutile.
- **Usare metodi virtuali nelle classi base al posto del type checking dinamico se possibile**.

> Il downcasting in generale rispetto ai metodi virtuali in una gerarchia di classi comporta bassa estensibilità del codice. Mentre con i metodi virtuali posso estendere la gerarchia a piacimento, se provo a farlo con type checking dinamico ogni volta che inserisco una nuova classe derivata devo inserire altro codice per coprire il nuovo caso(tipo) introdotto.

36_SOLID object oriented design

SOLID design

- Single responsibility
  - Una classe dovrebbe avere una singola responsabilità
- Open/Closed principle
  - Open for extension
  - Closed for modification
- Liskov substitution principle
  - Rimpiazza oggetti con istanze dei loro sottotipi senza alterare la correttezza del programma
- Interface segregation principle
  - Tante interfacce specifiche sono meglio di una generica
- Dependency inversion principle
  - Le astrazioni non dovrebbero dipendere dai dettagli

37_Eredità multipla

Eredità multipla

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**.

> Si risolve con operatore di scoping `::` oppure con una ridefinizione del metodo che avrebbe, attraverso la name hiding rule, mascherato le funzioni dei genitori.

Problema del diamante

Quando ho un'ereditarietà a diamante ho due problemi:

1. Ambiguità.
2. Spreco di memoria.

Derivazione virtuale

> Risolvo dichiarando 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. Questo grazie al compilatore che introduce il solito puntatore alla VMT per i metodi (in questo caso derivazioni) dichiarati virtuali.

```
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
...
};
```

Unique final overrider rule

Vale solo per metodi virtuali.

```
class A {
public:
        virtual void print() = 0; //metodo virtuale puro
};

class B : virtual public A {
public:
        void print() override {cout << "B ";}
};

class c : virtual public A {
public:
        void print() override {cout << "C ";}
};

class D : public B, public C {
public:
        void print() override {cout << "D ";} //se non se non faccio
questo override viene tornato un errore "no unique dinal overrider for
A::print()" ho due override prima e non so quale prendere per la classe
derivata a diamante stesso problema dell'ereditarietà multipla per
metodi non ridefiniti
};

int main() {
        D d;
        A* p = &d; //compila
        p->prnt(); //stampa: D
}
```

```
class A {
public:
        virtual void print() {cout << "A ";}
};

class B : virtual public A {
public:
        //void print() override {cout << "B ";}
```

```
};

class c : virtual public A {
public:
        //void print() override {cout << "C ";}
};

class D : public B, public C {
public:
        //void print() override {cout << "D ";} //posso anche non fare
l'override in quanto eredito un unique final overrider
};

int main() {
        D d;
        A* p = &d; //compila
        p->prnt(); //stampa: A
}
```

```
class A {
public:
        void print() {cout << "A ";}
};

class B : virtual public A {
public:
        void print() {cout << "B ";}
};

class c : virtual public A {
public:
        void print() {cout << "C ";}
};

class D : public B, public C {
public:
        //eredito 2 metodi print() non virtuali, compila ma ambiguità in
compilazione per un'invocazione di d.print()
};

int main() {
        D d;
        A* p = &d; //compila
        p->prnt(); //stampa: A
        d.print(); //Illegale chiamata ambigua
        d.B::print(); //OK stampa: B
}
```

```
class A {
public:
        void f() {cout << "A ";}
};

class B : virtual private A {}; //derivazione virtuale privata

class c : virtual public A {}; //derivazione virtuale pubbloca

class D : public B, public C {}; //prevale la derivazione publica

int main() {
        D d;
        d.f(); //OK stampa: A (sarebbe C::f()) prevale derivazione
pubblica
        d.B::f(); //illegale non compila A::f() inacessibile
}
```

Costruttore per basi virtuali.

38_IOStream

IOStream

Stream: Sequenza non limitata di celle ciascuna contenente un byte.

- Posizione delle celle in uno stream inizia da 0.
- I/O effettivo attraverso un buffer associato allo stream.

Stati di uno stream

$8 = 2^3$ stati differenti rappresentati da un intero in [0, 7] rappresentato dal campo dati **state** della classe base `ios` corrispondente al numero binario di 3 bit:
`bad fail eof`.

> `bad fail` ed `eof` rappresentano un bit di stato

- `eof==1` $\Leftrightarrow$ stream nella posizione di end-of-file.

- `fail==1` ⇔ operazione precedente sullo stream è fallita senza perdita di dati, per esempio provo a leggere un `int` dove l'input è `double`.
- `bad==1` ⇔ operazione precedente sullo stream è fallita con perdita di dati. Errore fatale/fisico per cui normalmente non si può continuare, per esempio provo ad accedere a file o rete inesistenti.

In base al valore dei bit vengono definiti 4 metodi che tornano true solo in determinate condizioni:

- `bool good() const;` ritorna true se lo stato è good (tutti i bit a 0)
- `bool eof() const;` ritorna true se ho raggiunto la fine dello stream
- `bool fail() const;` torna ture se l'operazione fallisce senza perdita di dati
- `bool bad() const;`
- `iostate rdstate() const;`
- `void clear (iostate state = goodbit);`

Istream

Fa l'overloading dell'operatore di input `operator>>` per i **tipi primitivi e gli array di caratteri**.

`cin` è un oggetto di istream.

Per esempio `operator>>(double& val)` preleva dallo istream di invocazione una sequenza di caratteri che rispetta la sintassi dei litterali di $double$ e converte la sequenza nella rappresentazione numerica di $double$ assegnandolo a val. Se la sequenza non soddisfa la sintassi prevista per $double$, l'operazione è nulla e l'istream va in uno stato di errore recuperabile: `fail==1`, `bad==0`, non vengono effettuati prelievi dallo stream e l'argomento di `operator>>` non viene modificato.

Quindi definire l'overloading di `operator>>` in una certa classe C significa dare un significato alla conversione $sequenza\ di\ byte \Rightarrow oggetto\ di\ C$. Ovvero significa fare parsing di una sequenza di byte di input secondo le regole di rappresentazione sintattica degli oggetti di C.

```
class Punto {
        friend istream& operator>>(istream&, Punto&);
private:
        double x,y;
};

istream& operator>>(istream& in, Punto& p) {
        char cc; in >> cc;
        if(cc=='q') return in; //carattere q per usicre
        if(cc != '(') {in.clear(ios::failbit); return in;}
        else {
                in >> p.x;
                if(!in.good()) {in.clear(ios::failbit); return in;}
                in >> cc;
```

```
                if (cc != ',') {in.clear(ios::failbit); return in;}
                else{
                        in >> p.y;
                        if(!in.good()) {in.clear(ios::failbit); return
in;}
                        in >> cc;
                        if (cc != ')') {in.clear(ios::failbit); return
in;}
                }
        }
        return in;
}

int main() {
        Punto p;
        cout << "Inserisci un punto nel formato (x,y) ['q' per
uscire]\n";
        while(cin.good()) {
                cin >> p;
                if(cin.fail()) {
                        cout << "Input non valido, ripetere\n";
                        cin.clear(ios::goodbit);
                        char c=0;
                        while(c!=10) {cin.get(c);}
                        cin.clear(ios::goodbit);
                }
                else cin.clear(ios::eofbit);
        }
}
```

Ostream

Fa l'overloading dell'operatore di output `operator<<` per i **tipi primitivi e array di caratteri costanti**.

`cout` e `cerr` sono oggetti di ostream.

Stream di file

Stream di file fatti attraverso oggetti delle classi `ifstream`, `ofstream`, `fstream`. Sono disponibili diversi costruttori come da documentazione.

```
fstream file("dati.txt", ios::in|ios::out); /apre il file dati.txt in
I/O
if(file.fail()) cout << "Errore in apertura\n";
```

39_Exception Handling

Exception handling

L'idea di base è avere un metodo che genera l'eccezione, finché questa non verrà gestita continuerà ad essere mandata sempre più in alto nella "catena" dell'handling, se una volta arrivati al main nemmeno questo riesce a gestirla allora viene chiamata una funzione abort che termina il programma.

```
class Ecc_Vuota {}; //definizione di una classe di accezione

telefonata bolletta::Estrai_Una() {
        if (Vuota()) throw Ecc_Vuota();
        ...
}

int main() {//in questo caso la funzione chiamante
        ...
        try { b.Estrai_Una();}
        catch (Ecc_vuota e) {
                cerr << "La bolletta è vuota" << endl;
                abort(); //in stdlib.h
        }
        ...
}
```

Una `throw` può sollevare una espressione di qualsiasi tipo.

Flusso di controllo provocato da una throw

Quando in una funzione F viene sollevata un'eccezione di tipo T tramite una istruzione `throw` inizia la ricerca della clausola `catch` in grado di catturarla.

1. Se l'espressione throw è collocata in un blocco `try` nel corpo della stessa funzione F, l'esecuzione abbandona il blocco `try` e vengono esaminate in successione tutte le `catch` associate a tale blocco.
2. Se si trova un type match per una `catch` l'eccezione viene catturata e viene eseguito il codice della `catch`. Eventualmente al termine dell'esecuzione del corpo della `catch` il controllo dell'esecuzione passa al punto di programma che segue l'ultimo blocco `catch`.
3. Se non si trova un type match per una `catch` oppure se l'istruzione `throw` non era collocata all'interno di un blocco `try` della stessa funzione F la ricerca prosegue nella funzione che

ha invocato F.

4. Questa ***ricerca top-down sul call stack*** di funzioni continua fino a che si trova una `catch` che cattura l'eccezione o si arriva alla funzione `main` nel qual caso viene chiamata la funzione di libreria `terminate()` che per default chiama la funzione `abort()` che fa terminare il programma in errore.

> **è possibile che una `catch` si accorga di non poter gestire direttamente una certa eccezione**, in questo caso può rilanciare l'eccezione alla funzione chiamante con una throw.

Attenzione:

> ***utilizzando le eccezioni è possibile che si crei garbage***, in quanto il programma può essere terminato prima della de-allocazione della memoria, effettuata dopo il codice che può sollevare eccezioni.

Clausola catch generica

Viene utilizzata per risolvere il precedente problema di garbage.

```
gestore () try {
        risorsa rs;
        rs.use();
        ... //codice che può sollevare eccezioni
        rs.release(); //non eseguita in caso di eccezione
}

catch (...) {//... è la sintassi per una catch generica di una qualunque
eccezine generata
        rs.release();
        trhow;
}
```

Match del tipo delle eccezioni

La `catch` che cattura un'eccezione di tipo E è la prima `catch` incontrata durante la ricerca che abbia un tipo T compatibile con E.

Il tipo T della `catch` è compatibile con il tipo E dell'eccezione se:

- Il tipo T è uguale al tipo E.
- Il tipo E è un sottotipo di T, ovvero:
  - E è un sottotipo derivato pubblicamente da T;
  - T è un tipo puntatore B^* ed E è un tipo puntatore D^* dove D è un sottotipo di B

    - T è un tipo riferimento $B\&$ ed E è un tipo puntatore $D\&$ dove D è un sottotipo di B
- T è di tipo `void*` ed E è un qualsiasi tipo puntatore.
- Non possono essere applicate conversioni implicite.

40_C++11

C++11

> Per abilitarlo `g++ -std=c++11` come direttiva di compilazione.

Specifica delle eccezioni deprecata

- **Run-time checking**: il *test di conformità delle eccezioni avviene a runtime* e non a compile-time, quindi non c'è garanzia statica di conformità.
- **Run-time overhead**: Run-time checking richiede al compilatore codice addizionale che può agire negativamente su alcune ottimizzazioni.
- **Inutilizzabile con i template**: in generale i parametri di tipo dei template non permettono di specificare eccezioni.

Inferenza automatica di tipo

Esempio di strong typing:

```
vector< vector<int> >::const_iterator cit=v.begin();
```

Keyword `auto` (big chad)

Per evitare strong typing come nel precedente esempio si ricorre alla **keyword** `auto`.

```
auto x=0; //x ha tipo int perchè 0 è un literale di tipo int
auto c='f'; //char
auto d=0.7 //double
auto tasse_universitarie = 2500000000000L //long int
auto y = qt_obj.qt_fun(); //y ha il tipo di ritorno di qt_fun

void f(const vector<int> &vi) {
        vector<int>::const_iterator ci=vi.begin();
        ...
} //che maroni diventa
```

```
void f(const vector<int> &vi) {
        auto ci=vi.begin();
} //mega chad auto
```

Keyword `decltype`

Determina ***staticamente*** il tipo delle espressioni.

```
int x=3;
decltype(x) y=4; //int
std::vector<int> v(1);
auto a = v[0]; //int
decltype(v[1]) b = 1; //int
```

Inizializzazione uniforme aggregata

- per array
- **per contenitori stl**

```
int* a = new int[3] {1, 2, 0};

class X {
        int a[4];
public:
        X() : a{1,2,3,4} {} //inizializzazione di campo dati array
};
```

```
std::vector<string> vs = {"first", "second", "third"};

void f(std::list<double> l);
fun({0.34, -3.2, 5, 4.0});
```

Keyword `default` e `delete`

Recap:
Per ogni classe sono disponibili le versioni standard di:

1. Costruttore di default (perso con ridefinizione di un costruttore qualsiasi)
2. Costruttore di copia
3. Assegnazione
4. Distruttore

> 2,3,4 fanno parte della Rule of 3.

```
class A {
public:
        A(int) {}
        A() = default; //se per esempio definisco un costruttore non di
default, quello di default viene nascosto, con questa dichiarazione è
possibile ripristinarlo
        virtual ~A() = default; //anzichè dover definire(implementare)
il distruttore in quanto dopo la dichiarazione è necessario farlo, posso
usare la keyword default in modo da usare quello standard, salvo
particolari esigenze per gestione del garbage in cui è necessario
ridefinirlo.
};
```

```
class NoCopy {
public:
        NoCopy& operator=(const NoCopy&) = delete; //elimino l'operatore
=
        NoCopy (const NoCopy&) = delete; //elimino il costruttore di
copia
};
```

Keyword `override` (override esplicito)

In C++11 anziché fare l'override di un metodo virtuale e dimenticarsi per strada che è un override, si può usare la keyword `override` per esplicitarlo.

```
class B {
public:
        virtual void m(double) {}
        virtual void f(int) {}
};

class D : public B {
public:
        virtual void m(int) override {} //Illegale, ricorda che puoi
cambiare solo tipo di ritorno, se cambi parametri diventa una
ridefinizione e la keyword override da errore.
        virtual void f(int) override {} //OK
};
```

Keyword `final`

Permette un'ottimizzazione di de-virtualizzazione fatta dal compilatore, che evita in blocco il late binding su derivazioni successive risparmiando tempo.

```
class B {
public:
        virtual void m(int) {}
};

class C : public B {
public:
        virtual void m(int) final {} //final overrider
};

class D : public C {
public:
        virtual void m(int) {} //illegale
};
```

Keyword `nullptr`

Può sostituire la macro `NULL` e il valore `0`.
il tipo di `nullptr` è `std::nullptr_t` convertibile **implicitamente** a qualsiasi tipo ***puntatore e bool*** non convertibile ai tipi primitivi integrali (int).

```
void f(int);
void f(char*);

int main() {
        f(nullptr); //invoca f(char*)
}
```

Chiamate a costruttori in lista di inizializzazione

```
class C {
        int x, y;
        char* p;
public:
        C(int v, int w) : x(v), y(w), p(new char [5]) {}
        C() : C(0,0) {}
        C(int v) : C(v,0) {}
};
```

Funtori (mega chad)

Un funtore è un oggetto di una classe che può essere trattato come fosse una funzione.

```
class Functor {
private:
        int x;
public;
        Functor(int n) : x(n) {}
        int operator() (int y) const {return x+y;}
};

int main() {
        Functor SommaCinque(5); //costruttore inizializza x a 5
        cout << SommaCinque(6); //stampa 11 tramite overloading di
operator()
}
```

```
class MoltiplicaPer {
private:
        int factor;
public:
        MoltiplicaPer(int x) : factor(x) {}
        int operator() (int y) const {return factor*y;}
};

int main() {
        vector<int> v = {1,2,3};
        cout << v[0] << " " << v[1] << " " << v[2]; //stampa 1 2 3
        std::transform(v.begin(), v.end(), v.begin(), MoltiplicaPer(2));
        cout << v[0] << " " << v[1] << " " << v[2]; //stampa 2 4 6
//inizializza factor a 2 e poi viene passata ogni cella di v a
MoltiplicaPer come MoltiplicaPer(v[i]) v[i] in realtà e un iteratore
dereferenziato
}
```

> `std::transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op);`

Con template

```
class UgualeA {
private:
        int number;
public:
        UgualeA(int n) : number(n) {}
        int operator() (int x) const {return x==number;}
```

```
};

template<class Functor>
vector<int> find_matching(const vector<int>& v, Functor pred) {
//dichiarazione della funzione find_matching
        vector<int> ret;
        for(auto it=v.begin(); it<v.end(); ++it)
                if(pred(*it)) ret.push_back(*it); //operator() (int) è
CONST perchè v è un const vector e auto da un const_iterator
        return ret;
}

int main() {
        vector<int> w = {1,5,1,3};
        vector<int> r = find_matching(w, UgualeA(1)); //inizializza il
template di metodo find_matching con il funtore UgualeA a sua volta
inizializzato dalla chiamata al costruttore UgualeA(int x) con x==1
        for(int i=0; i<r.size(); ++i) cout << r[i] << " ";
}
```

Lambda espressioni o closures

`[capture list] (lista parametri) ->return_type {corpo}`

- **Capture list:** elenca la lista delle variabili all'esterno della lambda espressione usate come l-valore(lettura-scrittura) o r-valore(lettura) dalla closure.

```
[] //nessuna variabile esterna catturata
[x, &y] //x per valore, y per riferimento
[&] //tutte le variabili esterne catturate per riferimento
[=] //tutte le variabili esterne catturate per valore
[&, x] //tutte le variabili per riferimento a parte x per valore
```

- parametri, return_type e corpo

```
[] ->int {return 3*3;} //lista vuota di parametri
[](int x, int y) {return x+y;} //tipo di ritorno implicito: int
[](int x, int y) ->int {return x+y;} //tipo di ritorno esplicito:
int
[](int& x) {++x;} //tipo di ritorno implicito: void
[](int& x) ->void {++x;} //tipo di ritorno esplicito: void
```

> `this` può solo essere catturato da una closure per valore perché non posso modificarlo.

```
void f(const std::vector<int>& v, int fattore) {
        std::for_each(v.begin(), v.end(),
                [fattore](int x) {std::cout << fattore*x << " ";}
                );//prende fattore in input per copia, x è un parametro
della lambda espressione dato dalla dereferenziazione dell'iteratore per
ogni campo del vettore v, tale valore viene preso per copia e
moltiplicato per il fattore.
}
```

> `std::for_each(InputIterator first, OutputIterator result, UnaryFunction F)`
> necessario `#include<algorithm>`

Esempio lamda expression e template di lambdas functor: (omega chad level)

```
class RubricaEmail {
private:
        vector<string> rub;
public:
        template<class Functor>
        vector<string> trovaIndirizzi(Functor test) const {
                vector<string> ris;
                for(auto it = rub.begin(); it != rub.end(); ++it)
//sostituibile con (auto it : rub) e poi non devo nemmeno dereferenziare
it in quanto è già l'elemento del vontenitore
                        if(test(*it)) ris.push_back(*it); //itero la
rubrica e chiamo il funtore come funzione su ogni oggetto del vettore in
questo caso di tipo stringa, se il test è positivo allora aggiungo
l'oggetto al risultato
                return ris;
        }
};

vector<string> trovaIndirizziGmail(const RubricaEmail& r) {
//dichiarazione di funzione trovaIndirizziGmail con input oggetto
RubricaEmail che sfrutta il metodo funtorizzato trovaIndirizzi
        return r.trovaIndirizzi(
                [](const string& email) { //parametro di input per il
funtore è un iteratore su const vector<string> (perchè prendo un const
RubricaEmail) dereferenziato, ovvero una stringa costante
                        return email.std::find("@gmail.com") !=
string::npos; //ritorna 1 se trova la stringa "@gmail.com" come
sottostringa di email in quanto find torna l'iteratore al primo
carattere che matcha, se invece non trova "@gmail.com" allora find torna
il tipo string::npos (nopos) e quindi il return sarà 0. Questo valore è
valutato nel metodo trovaIndirizzi, che se 1 inserisce la mail nel
```

```
risultato altrimenti no, quindi viene passato il risultato a
trovaIndirizziGmail e la magia è fattta.
                }
        );
}

vector<string> trovaIndirizziConMatch(const RubricaEmail& r, const
string& match){
        return r.trovaIndirizzi(
        [match](const string& email) {
                        return email.std::find(match) != string::npos;
                }
        );
}
```

std::shared_ptr

> necessario `#include <memory>`

```
class C {
public:
        int x;
        C(int y=0) : x(y) {std::cout << "C(" << x << ") \n";}
        ~C() {std::cout << "~C() \n";}
        C(const C& x) {std::cout << "Cc \n";}
        C& operator=(const C& x) {std::cout << "C= \n"; return *this;}
};

int main() {
        std::shared_ptr<C> p1 = std::make_shared<C>(5); //C(5)
        std::cout << "p1->x = " << p1->x << std::endl; //5
        std::cout << "p1 Reference count = " << p1.use_count() <<
std::endl; // 1

        std::shared_ptr<C> p2(p1); std::shared_ptr<C> p3(p2);
        std::cout << "p1 Reference count = " << p1.use_count() <<
std::endl; // 3
        std::cout << "p2 Reference count = " << p2.use_count() <<
std::endl; // 3
        std::cout << "p3 Reference count = " << p3.use_count() <<
std::endl; // 3
        if (p1==p3) std::cout << "p1 == p3\n"; //p1==p3

        std::cout << "Reset p1 " << std::endl; p1.reset();
        std::cout << "p1 Reference count = " << p1.use_count() <<
std::endl; // 0
        std::cout << "p2 Reference count = " << p2.use_count() <<
std::endl; // 2
```

```
        std::cout << "p3 Reference count = " << p3.use_count() <<
std::endl; // 2

        p1.reset(new C(7)); //C(7)
        std::cout << "p1 Reference count = " << p1.use_count() <<
std::endl; // 1

        p1 = nullptr; // ~C() conversione da nullptr a shared_ptr,
ccount scende a 0 e viene invocato il distruttore di C per non avere
garbage
        std::cout << "p1 Reference count = " << p1.use_count() <<
std::endl; // 0
        if(!p1) std::cout << "p1 is NULL" << std::endl; //p1 is NULL

        p2 = std::make_shared<C>(2);
        std::cout << "p1 Reference count = " << p1.use_count() <<
std::endl; // 0
        std::cout << "p2 Reference count = " << p2.use_count() <<
std::endl; // 1
        std::cout << "p3 Reference count = " << p3.use_count() <<
std::endl; // 1

        std::shared_ptr<C> p4; p4=p3
        std::cout << "p3 Reference count = " << p3.use_count() <<
std::endl; // 2
        std::cout << "p4 Reference count = " << p4.use_count() <<
std::endl; // 2
}
```

Gli shared pointer sono l'arma di dio per i puntatori e per passare probabilmente P2.

STD C++17

> range for
> structured_binding

```
#include <iostream>

#include <algorithm>

#include <map>

#include <memory>

using namespace std;
```

```cpp
int main() {

        std::vector<int> i {0,1,2,3,4,5,6,7,8,9};

    for(auto a : i) {
        a = 1;
    };

    std::for_each(i.begin(), i.end(), [] (int x) {cout << x << ",";});
//0,1,2,3,4,5,6,7,8,9,

    for(const auto& a : i) {
        //a = 1; illegale "assignment of read-only reference ‘a’"
    };
    cout << "\n";

    for(const auto a : i) {cout << a << ",";} //0,1,2,3,4,5,6,7,8,9,
    cout << "\n";

    for(auto& a : i) {
        a = 1;
    };
    for(const auto a : i) {cout << a << ",";} //1,1,1,1,1,1,1,1,1,1,
    cout << "\n";

    std::map<const int, double> j {{0, .01},{1, .1},{2, .2},{3, .3},{4,
.4},{5, .5},{6, .6},{7, .7},{8, .8},{9, .9}}; //dichiarazione const per
il primo membro non necessaria, se non specificata il tipo del primo
membro viene dichiarato const in automatico in quanto chiave

    cout << "auto keyvalue : j \n";
    for(auto keyvalue : j) {
        //keyvalue.first = 1; "error: assignment of read-only member
‘std::pair<const int, double>::first’"
        keyvalue.second = 1;
        cout << "(" << keyvalue.first << ", " << keyvalue.second << ")"
<< ",";
    };
    cout << "\n";

    for (const auto& kv : j) {cout << "(" << kv.first << ", " <<
kv.second << ")" << ","; } //(0, 1),(1, 0.1),(2, 0.2),(3, 0.3),(4, 0.4),
(5, 0.5),(6, 0.6),(7, 0.7),(8, 0.8),(9, 0.9),
    cout << "\n";

    cout << "auto& keyvalue : j \n";
    for(auto& keyvalue : j) {
        keyvalue.second = 1;
        cout << "(" << keyvalue.first << ", " << keyvalue.second << ")"
<< ",";
```

```
    };
    cout << "\n";


    for (const auto& kv : j) {cout << "(" << kv.first << ", " <<
kv.second << ")" << ","; } //(0, 1),(1, 1),(2, 1),(3, 1),(4, 1),(5, 1),
(6, 1),(7, 1),(8, 1),(9, 1),
    cout << "\n";
}
```
