Metodologie di programmazione

Programmazione Avanzata

Università degli Studi di Parma

110 pag.

Processo di compilazione

Su linux possiamo compilare(codice c++) tramite l'istruzione

```
g++ -wall -wextra nome.cc -o nomeoutput
```

Dove:

- -wall -wextra= operazioni per avere i warning di base
- -o: serve per indicare un nome per il file di output, di base sennò verrà chiamato A.out

Possiamo poi eseguire il programma con:

```
./.nomeoutput
```

Il processo di compilazione si divide in diverse fasi:

preprocessing

operazione

Si cerca e avviano le direttive di inclusione (quelle con #, ad esempio gli include)

output

unità di traduzione

fase

Possiamo vedere come output l'unità di traduzione tramite l'opzione -E del compilatore (Possiamo dire al compilatore dove cercare le varie librerie da inserire attraverso l'opzione -I /directory/...)

Esempio

Grazie a questa fase vengono inserite tutte le varie librerie che utilizziamo, come #include iostream, senza di essa non potremmo usare cout ecc

Compilazione in senso stretto

operazione

traduce il codice da noi scritto in codice macchina con tutti i vari accorgimenti di traduzione

output

codice in assembler (codice macchina)

fase

Possiamo vedere come output il codice macchina tramite l’opzione -S del compilatore

assemblaggio

operazione

traduce il codice macchina in **codice oggetto in binario** con tutti i vari accorgimenti di traduzione

output

Codice oggetto (in binario)

fase

Possiamo vedere come output il codice binario tramite l’opzione -c del compilatore

collegamento

operazione

Crea un file finale contenente riferimenti a tutte le parti create precedentemente, di conseguenza adesso abbiamo un file completo da poter eseguire

(non contiene effettivamente tutte le parti del programma, contiene dei riferimenti ad essi e quindi dove poterli trovare quando dobbiamo eseguire)

output

Programma finito

fase

é il normale comportamento del compilatore, quindi basta inserire -o per nominare il file di output e otterremo in automatico il programma finito

Per inserire diversi file oggetto al suo interno basta passarli come parametro

```
g++ file1 file2 file3 ... -o output
```

Entità

dichiarazione

La dichiarazione come suggerisce il nome è l'operazione che consiste nel dichiarare l'esistenza di una determinata entità e associarla ad uno specifico nome.

```
Struct indirizzo{};
//in questo caso si è semplicemente detto che è si definirà in futuro una struttura di nome indirizzo
```

Analogamente è una dichiarazione:

```
Int I;
```

nota bene

In questo caso non stiamo definendo nessuna struttura della nostra entità ma stiamo solo dicendo che esiste ed ha quel nome.

extern

è una parola che va inserita **prima della dichiarazione** di una entità e serve per **indicare che l'entità verrà definita in un secondo momento in altri file**, **serve quindi per dire al compilatore di utilizzarla normalmente come se fosse definita** (come detto verrà fatto in un secondo momento/luogo)

Dopo questo Extern possiamo anche dare un valore alla variabile (una sola volta per programma, sennò si causerebbero errori)

```
extern int a=5;
```

nota bene

Per le funzioni spesso viene omessa la dicitura extern

definizione

La definizione è la fase successiva alla(o che comprende anche la) dichiarazione, **infatti in questa fase si forniscono altri elementi oltre il nome**

```
Struct indirizzo{
int numero_civico;
String via;
String città;
};
```

Nota bene

A differenza della fase definita precedentemente stiamo descrivendo anche la struttura interna del nostro tipo

scopo

In genere ci sono 2 motivi per fare una dichiarazione:

- **dichiarazione senza definizione**: si fa la dichiarazione senza nessuna definizione perché si userà l'entità attraverso dei puntatori, per i puntatori essendo variabili di dimensione fissa (dato che contengono semplicemente l'indirizzo di memoria della variabile a cui puntano) non serve che venga fatta la definizione immediatamente (**un puntatore ad una entità non ancora definita viene chiamato puntatore opaco**)
- **dichiarazione pura (forward)**: viene fatta la dichiarazione di una entità qualsiasi e la si definisce in un secondo momento, in genere la definizione viene poi fatta in un altro file o dopo il main e **si usa questa tecnica per semplice pulizia del codice**
    - Fino a c++11 non era possibile fare una dichiarazione pura del tipo enumerate

template

Attraverso di essi si definisce una struttura generica di una funzione,classe o struct.
Il template è parametrico, nei suoi parametri si possono passare dei tipi, così da utilizzarli al suo interno.
Attraverso questa sua proprietà otteniamo appunto la struttura generica e non rigida

```
template <typename T>
class classe{
    T variabile;
    //così facendo abbiamo un tipo non definito a priori ma
```

```
template <typename T>
T somma(T v1,T v2){
    return v1+v2;
    //questa somma essendo con tipi passati da utente può
```

```
che lo sarà in un secondo momento tramite parametri
}
```

```
essere applicata a diveri tipi di dato
(int,float,boolean,ecc...)
}
```

scope (campo d’azione)

Definisce le parti di codice in cui una entità è visibile

regola generale

La regola generale dice che una entità è visibile solamente dopo la sua dichiarazione, non prima. (in realtà questo non è valido all'interno delle classi ma in genere nei vari blocchi di codice si)

```
------------------------qui NO-------------------------
int entità;
------------------------qui SI-------------------------
```

Ci sono diverse tipologie di scope, ognuna con diversa ampiezza

tipologie

Scope globale (o di namespace)

Quando una dichiarazione non è all’interno di una classe/funzione o in generale un blocco di codice.
é quindi visibile ovunque nel programma e **ci si può riferire ad essa attraverso lo scope operator ::**

operatore di scope

attraverso lo scope operator :: si può definire da quale blocco di scope prendere la variabile/funzione ecc

namespace

Si può anche definire un namespace, ovvero un blocco di codice con visibilità globale che contiene varie dichiarazioni di entità.

```
namespace nome_namespace{
dichiarazioni di variabili, funzioni, classi ecc
}
```

Per riferirsi ad essi poi si utilizza l’operatore detto prima.

```
nome_namespace::nome_entità
```

scope di blocco

entità dichiarate all'interno di un blocco hanno validità solo in quel blocco. (con blocco si intende funzioni/cicli/if ecc)

```
for(int i=0;i<num;i++){
    //la i è visibile solo dentro al for
}
    //qui la i non è più visibile
```

scope di classe

Indipendentemente da quanto detto prima, nelle classi possiamo usare le entità anche se le dichiariamo/definiamo dopo. (ovviamente però dobbiamo definirle in qualche punto).

```
class persona {
    void stampa(){
    cout<<nome<<cognome;
    //usiamo nome e cognome prima di averli dichiarati, nelle classi è possibile
}
String nome;
String cognome;
}
```

Possiamo poi utilizzare le funzioni o dati della classe tramite l'operatore . applicato all'oggetto

persona francesco

francesco.nome="francesco";

Le classi hanno dalla loro l'ereditarietà, infatti una classe può ereditare da un'altra e di conseguenza vede tutte le sue entità (in realtà ci sono dei modificatori di visibilità che la rendono leggermente più complicata di così "private" "public" "protected")

scope di funzione

é uno scope utilizzato per le etichette che verranno usate per i go to, ha un comportamento strano rispetto a qualsiasi altro scope visto fino ad ora, infatti non segue nessuna regola di blocco.
l'etichetta in questione è visibile in tutto il programma (sia prima che dopo la sua "dichiarazione") e quindi richiamabile da ovunque.

Il go to però è un elemento sconsigliato nella programmazione, potrebbe causare molti problemi e in genere non si utilizza, di conseguenza neanche le etichette.

enum

Per evitare problemi con gli enum sarebbe meglio utilizzare l'operatore di scope, infatti definendo i seguenti enum:

- enum prova{rosso,blu}
- enum provad{rosso,verde}

con il seguente comando "cout<<rosso" non sapremmo a quale dei due enum ci stiamo riferendo, dobbiamo specificarlo con l'operatore ::
→ "cout<<prova::rosso"

modificatori di scope

Quelli appena definiti si chiamano **scope potenziali**, ovvero quelli teorici.
Ci sono però alcuni costrutti che ne modificano la visibilità

hiding

Quando ci sono campi "annidati" è possibile che una dichiarazione di un blocco interno nasconda quella di un blocco esterno

```
Int main(){
    int a=10;
    for (int  a=0;a<5;a++){
        cout<<a<<endl;
//in questo caso con a si indica quella interna al for, non quella esterna, si è quindi soggetti alla hiding
}
cout<<a;
```

```
//stampa 10 perché intende quella esterna
}
```

Questo fenomeno capita anche durante la derivazione delle classi quando si crea una entità nella sottoclasse che però è già presente all'interno della superclasse (si va a sovrascrivere quest'ultima)

estensione

corrisponde al capitolo su using

using

Come abbiamo detto possiamo utilizzare entità di altri scope tramite l'operatore ::, ad esempio come detto prima nel namespace.

In generale se utilizziamo un namespace molte volte possiamo evitare la dicitura "nomenamespace::entità" ogni volta, per farlo andremo ad inserire la dicitura "using namespace nome_namespace" e quindi quella particolare entità sarà utilizzabile semplicemente tramite il nome (ad esempio capita spesso per il namespace std contenente funzioni molto usate come cout, cin ecc)

Possiamo poi utilizzare questa dicitura per una particolare entità del nostro namespace (se magari utilizziamo prevalentemente solo alcune entità) semplicemente inserendo alla precedente (più generale) l'operatore di scope con l'entità richiesta.
using namespace nome::entità

Ovviamente l'entità o il namespace sarà utilizzabile da dopo la dicitura inserita, non prima

```
using namespace std;
      //oppure
using namespace std::cout;

int main(){cout<<"prova";}
```

attenzione però, se richiamiamo più volte lo stesso namespace (o la stessa entità del namespace) avremo un errore perché sarà già presente.

Ad esempio:

```
sing namespace std;
sing namespace std::cout;
```

```
/darà errore perché cout è contenuto in std che avevamo già
ichiamato precedentemente, ovviamente anche il contrario
vrebbe dato errore
```

In generale i passaggi di controllo quando cercheremo un'entità saranno:

- vedo se l'entità è presente in questo blocco
- se non è presente passo a controllare il blocco superiore (a me visibili ovviamente)
- se non c'è nei blocchi visibili a me controllo le var globali
- se non c'è neanche nelle var globali guardo le direttive using

Per i motivi sopra detti è meglio evitare di mettere le direttive using in modo non controllato (ad esempio ad inizio di ogni file), potrebbero complicare le cose creando conflitti

lifetime

Il lifetime da come si può intuire indica il tempo di vita, quindi questa volta non andiamo ad analizzare il "dove" è visibile una entità ma il "quando" è visibile

Il tempo di vita di una entità è influenzato da come viene creato l'oggetto, gli oggetti in memoria possono essere creati:

- da una definizione (non basta una dichiarazione)
- creandoli tramite l'espressione new (quest'ultima li crea nella memoria dinamica, l'heap, e senza un nome)
- valutando un'espressione che crea implicitamente un oggetto (anche questo senza un nome definito)

possiamo trovare 2 punti principali nella vita di una entità:

inizio

la vita inizia dopo la sua costruzione, quest'ultima viene distinta in 2 fasi:

1) allocazione della memoria "grezza"
2) inizializzazione della memoria (se e quando prevista)

fine

Ovviamente si svolge solo per quegli oggetti che sono stati inizializzati in modo corretto, se non si svolge la fase precedente ovviamente neanche questa verrà svolta.

La vita di una entità termina con la sua distruzione, anch'essa composta in 2 fasi:

1) invocazione del distruttore (quando previsto)
2) deallocazione memoria grezza

tipi di allocazione

Esistono diversi tipi di allocazione:

allocazione statica

Allocazione fatta **non a tempo di esecuzione ma di compilazione**, di conseguenza **il ciclo di vita dell'entità in questione è uguale alla durata del programma**

Vediamo diversi elementi che hanno allocazione statica

variabili globali

Sono variabili inizializzate fuori dal main e da qualsiasi altra funzione o blocco.
Si differenziano per tempo di inizializzazione, vediamo come:

- se fanno parte della stessa unità di traduzione (file di programma) semplicemente vengono inizializzate dall'alto verso il basso
- altrimenti **se fanno parte di diverse entità di traduzione non possiamo sapere con precisione l'ordine**, questo spesso porta ad avere problemi

Membri di una classe dichiarati con la parola static

La parola **static** rende l'elemento non associato ad una particolare entità della classe ma alla classe stessa, è quindi utilizzabile senza la dichiarazione esplicita di un oggetto.
Comunque ciò ne rende il funzionamento come le variabili globali

variabili locali con dicitura static

Vengono allocate prima di iniziare l'esecuzione del main, vengono inizializzate la prima volta in cui si esegue quella riga di comando e non ridefinite ogni volta che la si "percorre"

```
void foo(){
static int a=1;
a++;
}
```

```
//la a non viene inizializzata ogni volta che viene richiamata la funzione ma solo la prima volta
```

Allocazione thread local

Diversi thread in esecuzione, le variabili vengono allocate quando il thread inizia e deallocate alla sua fine.

Allocazione automatica

Variabili locali ad un blocco di esecuzione non static, l'oggetto viene creato dinamicamente (sullo stack).
L'allocazione e deallocazione sono dietro diretto controllo del sistema, vengono distrutte appena si esce da quel blocco dove è stata inizializzata l'entità.
L'ordine di allocazione corrisponde ovviamente all'ordine di esecuzione

Allocazione automatica di temporanei

Creiamo degli oggetti temporanei (senza associare ad essi un nome per poterli richiamare) all'interno di istruzioni, ad esempio:

```
Class S{}
void foo(S s){}

void bar(){
    void foo(S(42));
//viene fatta una allocazione temporanea per l'oggetto S inizializzato tramite il costruttore al valore 42 (omettiamo questa parte perché ovviamente è un esempio)
}
```

Il ciclo di vita dipende dalla vita dell'istruzione richiamate, in questo caso foo.
Le cose cambiano se si inizia a parlare di riferimenti.
Nel caso avessimo:
Const S& s=S(42) di base il ciclo di vita terminerebbe subito, in realtà però non terminerà finchè dura il riferimento.

Allocazioni dinamica

è l'allocazione che si applica quando si usa l'espressione "new"
Es: int * pi=new int (42)

non si alloca nello stack ma nella memoria dinamica, ovvero l'heap.

deallocazione

In questo caso la distruzione dell'oggetto non avviene automaticamente, ma è sotto controllo del programmatore tramite l'istruzione "delete"

La delete va applicata al puntatore: Delete pi;
(Anche se il valore di pi non cambia lui continua a puntare sempre a quell'indirizzo che però ora sarà vuoto)

dangling pointer

E quindi verrà chiamato "**dangling pointer**", tradotto, puntatore pendente (ovvero che punta a qualcosa che non esiste)

problemi con i puntatori

ci sono diversi problemi con i puntatori:

- **use after free**: si va a scrivere/leggere nell'indirizzo di memoria puntato dopo che si era usato l'istruzione delete, deallocando quindi quel valore. (ovvero si vanno a fare operazioni su un dangling pointer)
  Dopo ciò si potrebbero generare i più svariati errori, è quindi una istruzione da evitare.
- **double free**: usare due o più volte delete sullo stesso indirizzo,ciò potrebbe causare la distruzione di una parte di memoria allocata ormai ad altri dati.
- **memory leak**: zona di memoria allocata ma non più puntata da nessun puntatore occupando memoria e portando problemi appesantendo il sistema inutilmente
- **wild pointer**: come nella "use after free" si indirizza questa volta a memoria casuale, scrivendo e leggendo dove non c'è un oggetto, o non l'oggetto inteso

behavior

Esistono vari comportamenti possibili che possono ritrovarsi dopo diverse situazioni

well defined behavior

comportamento ben definito, è il comportamento che si ottiene quando qualcosa ha un comportamento prestabilito per qualsiasi evenienza possibile

local specific behavior

i comportamenti sono ben definiti ma potrebbero variare alcune cose in base alla locazione geografica (credo solo geografica), ad esempio in base alle valute del luogo,

o il fuso orario ecc

implementation defined
l'implementatore deve dichiarare come si comporta una specifica strutture nei vari casi

unspecified behavior
ci sono modi determinati per fare le cose ma l'implementatore non è tenuto a dirci in che modo si comporta

undefined behavior
hai utilizzato il codice come non dovresti, quindi non ti prometto nulla, mi comporto nel modo che voglio

tipi

tipi fondamentali
ci sono diversi tipi fondamentali che formano un linguaggio di programmazione

tipi booleani
hanno il tipo bool, e permettono di avere 2 valori:
- true: verificato
- false: non verificato

tipi caratteri
si dividono in 2 tipologie
- narrow:
  - char: di base la char potrebbe avere segno o no, ciò dipende dal compilatore che si sta usando
  - signed char
  - unsigned char
- wide:
  - wchar_t
  - char16_t
  - char32_t

tipi interi standard con segno

- signed char
- short
- int
- long long

tipi interi standard senza segno

- unsigned char
- unsigned short
- unsigned int

tipi floating point

sono i tipi che rappresentano numeri decimali

- float
- double
- long double

suffissi

per i tipi floating point possiamo usare la notazione decimale:

- F alla fine indica float
  - 1.2345F
- se manca alla fine indica double
  - 1.234
- L alla fine indica long double
  - 1234L

si può poi usare anche la **notazione scientifica**

- float: 1.234e2F= $1.234 * 10^2 F$
- double: 1.234e2= $1.234 * 10^2$
- long double: 1.234e2L= $1.234 * 10^2 L$

tipo void

serve per indicare che una funzione non ritorna alcun valore, oppure se utilizzato come tipo di ritorno di un cast esplicito indica che il valore ritornato dall'espressione deve essere scartato

tipo nullptr_t

è un tipo puntatore con valore nullo, è convertibile implicitamente in un qualsiasi altro

tipo puntatore

char

tipo che rappresenta un carattere alfanumerico
si rappresenta inserendo tra ‘ il carattere indicato: ‘a’ ‘3’ ‘Z’

Ne fanno parte anche alcune “sequenze di escape” ad esempio ‘\n’ che indica al compilatore di andare a capo

Prefissi

Possiamo indicare per i nostri caratteri alcuni prefissi che ne definiscono il tipo preciso:

- u8’a’ : carattere a in codifica utf-8
- u’a’ : char16_t
- U’a’ : char32_t
- L’a’ : wchar_t

suffissi

Non possiamo indicare un char che abbia come tipo: “signed/unsigned char” o “short/unsigned short”.
Di base quando scriviamo un numero come carattere avrà tipo intero

Se vogliamo modificare tale comportamento possiamo utilizzare i suffissi:

- LL: long long
  - 123LL
- L: long
  - 123L
- U: unsigned
  - 123U

U ed L/LL sono combinabili per ottenere unsigned Long o unsigned long long

In caso i suffissi non vengono inseriti si attribuisce alla variabile il primo tipo tra int, long, long long che corrisponde al valore

tipi composti

sono dei tipi che son formati da varie informazioni

- t&: riferimento ad un lvalue di tipo t
- t&&: riferimento ad un rvalue di tipo t

- t*: puntatore ad una variabile di tipo t
- t[n]: array di n elementi di tipo t
- t(t1,t2,t3,…,tn): tipo funzione che restituisce un valore di tipo t
- enumerazioni
- classi/struct

stringhe

Le stringhe sono delle sequenze di caratteri, le utilizziamo in due diverse forme, stringhe della libreria strandard del c++ e cstring

cstring

Sono la tipologia di stringhe che veniva usata in c normale, ovvero appunto degli array di char, dove in ogni cella dell'array veniva inserita una lettera della stringa.

ad esempio se scrivessimo "hello" avremmo una cstring, formata da un array di 6 caratteri:

- 5 lettere
- 1 che è il carattere terminatore inserito alla fine della stringa indicato con 0

Con esse possiamo usare gli identificatori indicato nel char u, U8,U,L

stringhe grezze

una specifica rappresentazione delle stringhe che ci permette di modificare la normale sintassi utilizzando delimitatori che vogliamo noi

ci permettono quindi di andare a capo, usare gli apici a piacimento ecc ecc

Sintassi:
R"delimitatore(stringa)delimitatore"
Esempio
R"stringa(scrivo ciò che voglio e vado a capo
tranquillamente, anche usando " o ' se mi va

)stringa"

user defined literal

permette all'utente di definire dei suffissi che fanno cambiare il modo in cui viene valutata una stringa

s=“stringaprova”s
Potremmo indicare che il suffisso s faccia diventare la stringa una stringa c++

stringhe libreria standard

Non ne abbiamo parlato

tipi puntatori e riferimento

i puntatori e riferimenti sono dei tipi simili che spesso vengono confusi, andiamo a vedere le differenze

puntatori

oggetto che ha come valore un indirizzo di memoria nel quale è presente una variabile. Si riferiscono ad un oggetto, potrebbe teoricamente puntare a se stesso, ma in linea di massima abbiamo 2 oggetti diversi:

- puntatore
- puntato

Un puntatore si crea attraverso l’operatore * prima del nome
Int *P;
Il tipo del puntatore viene dato (e deve essere uguale al tipo dell’oggetto puntato) perché serve per capire come trattare le operazioni che vengono fatte sull’oggetto

dereferenziazione

Per indicare un oggetto puntato da un puntatore P dobbiamo usare l’operatore di dereferenziazione *:
Es: *P++;
in questo modo andiamo ad aumentare di 1 il valore dell’oggetto puntato

```
int a=5;
int *P=&a;
//& prima del nome di una entità serve per prenderne il suo indirizzo di memoria
*P++;
//avremo che a=6;
```

In caso il puntatore punta a classi/strutture (quindi oggetti composti) si usa l’operatore -> per indicare uno specifico pezzo della struttura a cui punta.
P->anni
Indica che stiamo utilizzando la variabile anni della struttura puntata da P (equivale a fare (*P).anni)

Al contrario per indicare proprio il puntatore useremo semplicemente il suo nome P

array e puntatori

array e puntatori sono molto legati, passare una cella dell'array equivale a passare un puntatore che punta a quello spazio di memoria.

```
int a[100];
a[5]
//equivale a usare *(a+5) e di conseguenza anche *(5+a)
```

aritmetica dei puntatori

l'aritmetica dei puntatori quando essi puntato agli array dice:
(immaginando che puntatore_array sia un puntatore ad un array)

```
puntatore_array+tot //muoviti in avanti di tot posizioni
(caselle)
tot+puntatore_array //muoviti in avanti di tot posizioni
(caselle)
```

possiamo calcolare le distanze (in celle) di due elementi di un array scrivendo

```
puntatore1-puntatore2 //dove ovviamente i due puntatori
puntano agli elementi degli array
```

versione senza puntatori	versione con puntatori

`for(int i=0; i<100; i++)`
`{`
`  //fai qualcosa con a[i]`
`}`
	`for(int*p=a; p!=a+100; p++){`
	`  //fai qualcosa con *p;`
	`}`

sono uguali, con la seconda forma però si possono generalizzare codici per array di cui non si conosce bene la dimensione utilizzando coppie di puntatori e quindi eseguendo in un certo intervallo	
`int* fine=a[150]` `for(int*p=a; p!=fine; p++){`	

```
   //fai qualcosa con *p;
}
```

riferimento

non è un oggetto vero e proprio ma una sorta di alias (nome alternativo) per accedere ad un oggetto già esistente.
Per crearlo bisogna inizializzarlo per forza

differenze

come differenze principali abbiamo:

- non può esistere un riferimento che “indica” un valore null, però può esistere un puntatore a null (esso prenderà il valore nullptr)
- Una volta creato un riferimento non possiamo modificare l’oggetto a cui riferirsi, invece il puntatore può essere modificato in modo da puntare ad un altro oggetto
- Quando facciamo un’operazione su un riferimento implicitamente la stiamo facendo sull’oggetto indicato, al contrario invece, dato che il puntatore è un oggetto diverso da quello puntato quando lavoriamo su di esso facciamo le operazioni su lui non su quello puntato
- un eventuale qualificatore aggiunto al riferimento si applica necessariamente all’oggetto riferito, e non al riferimento stesso.
  Al contrario con il puntatore parlando (come detto sopra) di 2 oggetti diversi (puntatore e puntato) i qualificatori vanno utilizzati per i due oggetti separatamente.
    - se const sta a sinistra del * si applica all’oggetto puntato
    - se const sta a destra del * si applica all’oggetto puntatore
        - int i=5; //valore modificabile
        - const int ci=5; //valore non modificabile
        - int & r_i=i; //posso modificare il valore di i usando r_i
        - const int& cr_i=i //il riferimento non potrà cambiare il valore di i
        - int & r_ci=ci //errore, non posso usare un non const per riferirmi ad un const dato che sennò proverei a modificarlo
        - const int& cr_ci=ci; //ok, accesso in sola lettura
        - int & const cr=i; //errore const può stare solo a sinistra del &
        - int * p_i; //p_i e *p_i sono entrambi modificabili
        - const int * p_ci //p_ci è modificabile ma *p_ci non lo è
        - int * const cp_i=&i; //cp_i non è modificabile ma *pc_i lo è
        - const int* const cp_ci=&i //cp_ci e *cp_ci sono entrambi non modificabili

- p_i=&i //ok
- p_ci=&i; //ok, prometto di non modificare i usando *p_ci
- cp_i=nullptr //errore perché cp_i è const, quindi non possiamo modificarne il valore
- p_i=&ci //errore: non posso inizializzare un oggetto a non const usando un indirizzo const
- p_ci=&ci //ok, perché con un qualcosa const indico qualcosa di const
- cp_ci=&ci //errore cp_ci non modificabile

somiglianze

- quando termina il tempo di vita di un puntatore viene distrutto l'oggetto puntatore ma non l'oggetto puntato, succede lo stesso con un riferimento
- esiste un qualcosa di simile ai dangling pointer con i riferimenti quando ci si riferisce a qualcosa di non più esistente

qualificatori

ne esistono di due tipi "const" e "volatile"

volatile

si usa per quei valori che vengono cambiati spesso e da diverse "parti", quindi si segnano con tale qualificatore in modo da dire "guarda che il valore potrebbe essere cambiato dall'ultima volta che l'hai letto, devi rileggerlo"
(può cambiare spesso quindi si deve rileggere e scrivere sempre quando nominato)

const

vuol dire che è una variabile in sola lettura.
Non può essere modificato il valore
la qualificazione può essere fatta per tutto un oggetto o solo ad una parte (ovvero a singoli elementi della struttura)

costanti letterali

Sono costanti non simboliche, ad esempio numeri ecc…
esistono diverse sintassi, vediamo la rappresentazione del numero decimale 12 in alcune sintassi

- sintassi binaria: 0b1100
- sintassi ottale: 014
- sintassi esadecimale: 0xc

ODR (one definition rule)

ODR (one definition rule)

La one definition rule stabilisce il modo corretto di far interagire le unità di traduzione dice che:

1. ogni unità di traduzione può contenere non più di una definizione di ogni variabile, funzione, classe, enumerazione o template
2. ogni programma deve contenere esattamente una definizione di ogni funzione non inline usate nel programma
3. ogni funzione inline deve essere definita in ogni unità di traduzione che la utilizza
4. in ogni programma ci possono essere più definizioni di una classe /funzione/enumerazione/template di classe o funzione a condizione che:
    - tali definizioni siano sintatticamente identiche
    - tali definizioni sono semanticamente identiche

funzioni inline

funzioni dichiarate con la dicitura “inline” prima della funzione stessa.
Analogamente a quanto succede con una define, quando si definisce una funzione inline e la si richiama, quando si richiama viene “scambiata” la sua firma con il valore ritornato
(si usano più che altro per una questione di efficienza)
Es:

```
inline int funz() {return 3;}
int main(){
    printf("%d",funz()+1);
}
//è come se ci fosse scritto
int main(){
    printf("%d",3+1);
}
```

esempi di violazione

violazione del punto 1

```
Struct S{int a;};
Struct S{
char c;
double b;
}
------------------------------
int a;
int a;
```

in questo caso si compie una violazione, dato che non si potrebbero distinguere le due struct o le variabili a

ricordarsi però che esistono diversi scope, quindi:

```
Namespace N{int a;}
int a;
```

a e N::a sono diverse entità distinguibili non si fa violazione

E per quanto riguarda le funzioni è presente l'overloading, quindi:

```
Int incrementa (int a){...};
float incrementa (float a){...};
```

Sono distinguibili, quindi non è una violazione

violazione del punto 2

si usa una funzione/variabile dichiarata ma mai definita, la compilazione in senso stretto andrà a buon fine ma il linker darà errore al momento della generazione del codice eseguibile

oppure semplicemente dichiariamo più funzioni con lo stesso nome in diversi file, e poi chiamiamo quella funzione, il compilatore non saprà quale usare

violazione del punto 3

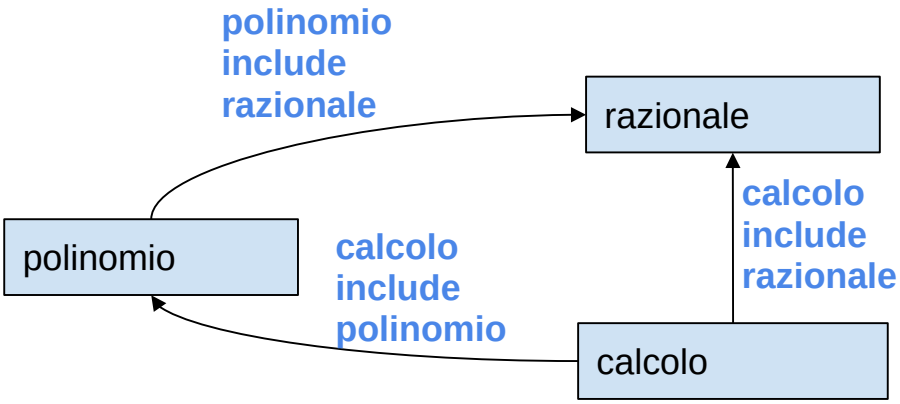
partiamo dal presupposto che il linker vede l'object file, per distinguere le funzioni inline da quelle non inline, esse vengono rappresentate rispettivamente dai simboli "w" e "t".

- w indica che è una funzione “debole” ovvero può presentarsi più di una volta, e basandosi su quanto detto nel punto 4 quando una funzione si presenta più volte questa deve essere rappresentata sempre uguale, quindi basta prendere una a caso tra le definizioni trovate

soluzioni

Una soluzione per non violare la ODR sarebbe usare la tecnica DRY (don’t repeat yourself) ovvero in sostanza scrivere le cose una sola volta.

Però a dir il vero questo non da sicurezza assoluta.
Immaginiamo una struttura del genere:



Problema: Calcolo include 2 volte razionale, dato che lo include direttamente e poi attraverso polinomio

Possibili soluzioni

Alcune soluzioni possibili:

- non includere razionale all’interno di calcolo dato che contiamo sul fatto che lo includa polinomio
  - non va bene perché se ad esempio non siamo noi i possessori della classe polinomio e di punto in bianco in “polinomio” non includiamo più razionale finiremmo con il non poter usare più la classe ottenendo errori

guardie contro l’inclusione ripetuta

Per ovviare a questi problemi usiamo delle direttive chiamate appunto guardie contro l’inclusione ripetuta e servono per evitare di includere più volte lo stesso header file

Sintassi

```
#ifndef nome_file
#define nomefile
//se non è definito nome_file allora definiscilo, altrimenti
passa a dopo la fine dell'if

#endif
```

elenco incompleto di possibili cose trovabili in un header file

- direttive: del pre processore (inclusioni, guardie, macro)
- commenti
- dichiariazioni/definizioni di tipo
- dichiarazione di variabili
- dichiarazione di costanti
- dichiarazione di funzioni
- definizioni di funzioni (solo inline)
- dichiarazioni/definizioni di template
- alias di tipi tramite typedef o using

cosa non dovremmo trovare

- definizione di variabili
- definizione di funzioni non inline
- dichiarazione/definizione di using

Passaggio argomenti a funzione

gli argomenti possono essere passati alla funzione secondo due modalità principali

Passaggio per valore

viene fatta una copia del valore dell'argomento nel parametro della funzione
sintassi:

- creazione: nomefunzione(tipo variabile)
- richiamare: nomefunzione(valore)

passaggio per riferimento

Il passaggio per riferimento ad lvalue consiste nel passare un collegamento al valore effettivo, non si fa quindi una copia.
Se la funzione modifica il valore del parametro si va a modificare l'effettivo valore

- creazione: Nomefunzione(tipo & variabile)
- richiamare: nomefunzione(variabile)

In questo caso non potremo passare un rvalue, dato che ci serve che effettivamente sia un oggetto creato

Se vogliamo impedire alla funzione di modificare il valore del parametro passato inseriremo il qualificatore const prima del tipo durante la dichiarazione della funzione
nomefunz(const tipo & variabile);

A volte le funzioni ritornano riferimento ad oggetti, ciò potrebbe essere un problema nel caso in cui si ritorni il riferimento ad un qualcosa che è stato creato all'interno della funzione (e che quindi dopo il return della funzione uscirebbe fuori scope, cancellandosi dalla memoria)

Passaggio puntatore

Possiamo passare alla funzione dei puntatori, quindi si va a ridurre lo "spazio" occupato passando effettivamente degli indirizzi di memoria a cui far puntare i nostri puntatori

- creazione: Nomefunzione(tipo * variabile)
- richiamare: nomefunzione(&variabile) //si passa l'indirizzo della var

utilità

In genere si attua la seguente scelta:

- se l'oggetto da passare è piccolo e/o non dobbiamo modificarlo passiamo per valore
- se l'oggetto da passare è grande e/o va modificato passiamo per riferimento, dato che si accorciano i tempi e i costi

overloading di funzione

Quando si implementano funzioni analoghe per diversi tipi possiamo definire una stessa funzione che in base al tipo cambia comportamento (o meglio diverse funzioni ma con lo stesso nome che secondo delle regole vengono scelte per essere utilizzate)

Possiamo infatti definire diverse funzioni con stesso nome, ma tipo di ritorno e parametri presi diversi.

Quando chiameremo la funzione quale sarà quella scelta dal compilatore?

Questa scelta avviene a tempo di compilazione e in modo statico, si basa sulla propria unità di traduzione e basta.
Si guarda solo il tipo degli argomenti, non il numero.

risoluzione dell'overloading

la risoluzione dell'overloading (ovvero questa scelta) si divide in 3 fasi

1) Individuiamo funzioni candidate
2) eliminiamo le funzioni non buone scegliendo quelle effettivamente utilizzabili
3) scegliamo la funzione migliore da invocare (se esiste)

1) Individuare funzioni candidate

Si controllano le funzioni solo della stessa unità di traduzione controllando:

- le funzioni con lo stesso nome della funzione chiamata
- e se sono visibili nel punto della chiamata

per gli operatori ricordiamoci che la macchina usa sempre il nome della funzione per esteso
++a= operator++(a)
ai[5]= operator[](ai,5)

regola ADL

Regola da seguire quando si parla di overloading, stabilisce che:

- nel caso di una chiamata di funzione non qualificata
  (quindi non sono presenti cose come std::... , c::... , ecc...)
- e la chiamata contiene al suo interno uno o più argomenti che hanno un tipo definito dall'utente (cioè come struct/class/enum) o puntatore/riferimento ad un tipo definito da utente
- e il tipo suddetto è definito nel namespace (generico N)

Allora la ricerca delle funzioni candidate viene effettuato anche all'interno del namespace N
Es:

```
namespace N{
struct S;
void foo(S s);
void bar(int N);
}
int main() {
N::S s;
foo(s);
//chiamata di funzione non qualificata, ed essendoci N::S come tipo del parametro (essendo S un tipo definito da utente) siamo autorizzati ad andare a cercare nel namespace N
int i=0;
bar(i);
// invece i non è definito da utente, quindi non possiamo definire la bar di N come funzione candidata
}
```

2) scegliere funzioni utilizzabili

- verificare che il numero di argomenti nella chiamata alla funzione sia compatibile con il numero di parametri nella dichiarazione della funzione
- e che ogni argomento nella chiamata di funzione abbia un tipo compatibile con il corrispondente parametro nella dichiarazione della funzione

compatibilità di numero di argomenti

- fare attenzione ad eventuali valori di default per i parametri
  Es:

```
void somma (int a; int b=5)
somma(1,3)
somma(1) //equivale a somma(1,5)
```

- fare attenzione all'argomento implicito this nelle chiamate di metodi non statici, ovvero:

```
class S{
    void foo(int a){...};
    void bar(){
    foo(5);
//che corrisponde a this->foo(5);
}
}

foo(5)//errore
//fuori dalla classe non possiamo usare this, dobbiamo fare
S s;
s.foo(5);
```

compatibilità di tipo degli argomenti

Ricordiamoci che in genere sono applicabili conversioni esplicite ed inplicite, soprattutto quest'ultime danno "problemi"
Es:

```
int a=5.5;
//alla fine a=5
```

classificazione di conversioni implicite

Per le conversioni implicite possiamo seguire una lista di preferenze (più è in alto la preferenza nella lista più si è preferibile rispetto alle altre sottostanti)

1) corrispondenza esatta: identità, trasformazioni lvalue, qualificazioni
2) promozioni
3) conversioni standard
4) conversioni definite dall'utente

corrispondenze esatte

preservano esattamente il valore dell'argomento

I. identità di match perfetti: non è una conversione perché il tipo dell'argomento è

esattamente il tipo del parametro
dove:

- i=var di tipo intero
- r=const int&
- r=i

tipo parametro	argomento
int	5
int	i
int *	&i
const int &	r
double	5.2

II. trasformazioni di Lvalue
Vedono le lvalue in modo leggermente diverso
si assume
b=int a[10];
void foo();

tipo parametro	argomento	
int	5	da lvalue a rvalue
int*	b	decay array-puntatore
void(*)()	foo	decay funzione-puntatore

III. conversioni di qualificazione:
Aggiungo const al tipo

tipo parametro	argomento
const int &	i
const int*	&i

le promozioni

Anche esse preservano il valore dell'argomento, in generale i tipi vengono promossi ad uno simile leggermente più grande

Es: il tipo migliore per fare calcoli è unsigned int, quindi i tipi vengono prima traslati ad essi e poi si esegue l'operazione

I. promozioni intere:
    A. da tipi più piccoli (char/short signed o unsigned al tipo int (signed o unsigned))
    B. Da bool a int, è un caso speciale, viene trasformato false in 0, true in 1
II. promozioni floating point: da float a double
III. promozioni delle costanti di enumerazione del c++ 2003 al più piccolo tipo intero (almeno int) sufficientemente grande per contenerle

conversioni standard

Tutte le conversioni (implicite, non definite dall'utente) che non ricadono negli altri casi
Es: da int a long è conversione standard, non promozione

In quelle da tenere in considerazione ci sono le conversioni tra riferimenti e tra puntatori, in particolare:

- la costante intera 0 e il valore nullptr (di std::nullptr_t) sono le costanti puntatori nullo, esse possono essere convertite implicitamente nel tipo t*, (per qualunque t) la costante 0 può essere convertita in nullptr
- ogni puntatore t* può essere convertito nel tipo void* che corrisponde ad un puntatore ad un tipo ignoto.
  non esiste il contrario, da void* a t*
- se una classe D è derivata (direttamente o indirettamente) dalla classe B allora ogni puntatore D* può essere convertito implicitamente in B*.
  Si parla quindi di up-cast, ovvero una conversione verso l'alto, questo perché semplicemente le classi base sono più astratte, la cosa analoga può essere fatta per riferimento da D& a B&.
  Non esiste il contrario da B&/B* a D*/D&

Conversioni implicite definite dall'utente

l'utente può fare in modo di creare delle conversioni che si applicano implicitamente

I. uso implicito di costruttori che possono essere invocati con un solo argomento (di tipo diverso dal tipo della classe) e che non sono marchiati "explicit"

explicit

la marchiatura explicit indica che non può essere fatta conversione implicita ma solo esplicita (quindi solo se la conversione è dichiarata)
Es:

```
struct razionale{
razionale(int num, int den=1);
}
foo(razionale r);
foo(5)
//essendo che il denominatore ha il valore standard 1 se diamo un solo parametro il valore int viene convertito a razionale direttamente

struct razionale{
explicit razionale(int num, int den=1);
}
//così facendo invece non sarebbe convertito implicitamente si dovrebbe fare con dei cast
```

II. uso di operatori da tipo utente verso l'altro tipo
Es:

```
Struct razionale{
    operator double() const; //conversione da razionale a double
}
```

3) selezione migliore funzione utilizzabile

Non è detto ci siano funzioni utilizzabili, quindi prima bisogna valutare le precedenti fasi.
Sia N il numero di funzioni utilizzabili, se

- N=0 allora ho un errore di compilazione
- N=1 allora utilizziamo l'unica utilizzabile
- N>1 si devono classificare le funzioni utilizzabili in base alla "qualità" delle conversioni richieste
  (Se ne esce solo 1 migliore semplicemente si prende quella, se non si riesce a definirne una migliore si ottiene errore di compilazione)

sequenza stilata nel capitolo "**classificazioni di conversioni implicite**"

corrispondenze esatte > promozioni > conversioni standard > conversioni definite da utente

Se ogni funzione ha M argomenti per ognuno di essi viene fatta una sorta di gara (capendo appunto di quale categoria fa parte il parametro) per capire la migliore funzione

Quindi una funzione X è preferibile rispetto ad Y se:
- non perde nessuno degli M confronti (quindi vince o pareggia)
- e vince almeno uno degli M confronti

In caso di sequenze di conversioni la peggior conversione usata in X_i vince se è migliore rispetto alla peggior conversione usata in Y_i

Analisi esercizi sull'argomento
- **analisi 1 appello**
- **analisi 2 appello**

sequenza di conversione

sequenza di conversioni standard
una serie di operazioni attuate sui dati:
- trasferimento di lvalue +
- promozione/conversione standard +
- qualificazione

(Le varie componenti non sono obbligatorie ma se ci sono devono essere nell'ordine specificato)

Es: supponendo che la classe D è derivata dalla classe B
double d=3,1415 //lvalue di tipo double
void foo(int) //funzione che accetta un intero per valore

foo(d) //chiamata dove forniamo un double come valore
//quindi viene fatta un trasferimento di lvalue e poi una trasformazione da double ad int

sequenza di conversione utente
- sequenza conversioni standard +

- conversione utente +
- sequenza conversione standard

progettazione di un tipo di dato

Inserendo costruttore()= default
(di base i compilatori suggeriscono funzioni standard)
//diciamo di lasciare il costruttore standard fornito dal compilatore e che ci va bene
(sarebbe uguale a non fornirla ma almeno, si mette per scritto lo usiamo)

Possiamo fare lo stesso per i valori costruttori di copia, ecc…

Consigli
Se non si sa quale tipo usare conviene definirne uno con typedef e poi in caso cambiare solo lui Es:

```
typedef miotipo=int
miotipo tot
miotipo add(){};
//se ora volessi fare il tutto cambiando a float basterebbe
inserire nel type def in alto miotipo=float
```

Quale è il tipo di this?

- void foo(): qui this sarebbe tipoclasse* const, quel const vuol dire che non possiamo cambiare il valore di this
- void bar() const: sarebbe tipoclasse const* const

Osservazione
• in generale per gli operatori /=/*=/+=/-= conviene ritornare un riferimento Es:

```
Razionale& operator+=(const razionale y){
...
}
```

tanto il valore viene riscritto nello stesso oggetto

- conviene poi definire esternamente alcuni operatori, dato che questo vuol dire che non potremo accedere ai dati “private” della classe
  Es:

```
class razionale {
pubic: ...
private: ...
}
```

```
razionale::operator++(...){}

O meglio ancora

operator++(razionale a,
razionale b){}
```

Sviluppo dell’implementazione

const correctness

Marchiare con const i metodi quando possibile ci permette di poterli richiamare anche su oggetti costanti, è importante farlo e viene chiamato **const correctness**

lista di inizializzazione delle classi base

Possiamo ottimizzare il processo di costruzione della istanza utilizzando la lista di inizializzazione delle classi base e dei dati membro.
Ovvero dopo il costruttore, prima di aprire la parentesi graffa, possiamo inserire una lista che ci permette di inizializzare con dei valori specifici i dati membro della classe

Si inserisce nella posizione indicata : dato(valore)
Ovviamente la lista può contenere quanti dati membro vogliamo, essi si dividono con una virgola.
Il valore può essere sia un valore numerico, sia una variabile (magari appunto tra quelle passate nei parametri del costruttore)

Es:

Scritture analoghe	
con lista di inizializzazione	senza lista di inizializzazione

`Razionale{`
`  int num;`
`  int den;`
`  razionale (int n, int d) : num(n), den(n){`
`  …`
`  }`
`}`
	`Razionale{`
	`  int num;`
	`  int den;`
	`  razionale (int n, int d){`
	`  num=n;`
	`  den=d;`
	`  }`
	`}`

Si consiglia di inserirli nella lista di inizializzazione nello stesso ordine in cui si inizializzano.

Si utilizza la lista di inizializzazione anche per una questione di efficienza, infatti con essa il tutto risulta più efficiente perché inizializziamo direttamente con il valore, mentre nella seconda modalità andiamo ad inizializzare con un valore casuale (o a 0, dipende dal compilatore) e poi cambiamo il valore alle variabili.

asserzioni

Le asserzioni permettono di fare controlli.
é una “domanda” che viene fatta e se la risposta è

- affermativa: il codice continua
- negativa: il codice scaturisce in un errore

Es:
assert(d!=0);

Per utilizzarla serve includere la libreria base delle asserzioni.
#include <cassert>

In caso l’asserzione fallisce il compilatore ritorna nell’errore il controllo fallito citandolo,

quindi è facile capire quale asserzione non andava bene e cosa dover “sistemare” o comunque individuare il problema

Es:

```
//nel caso dell'asserzione precedente ritornerebbe
assertion 'd!=0' failed
```

Le asserzioni sono molto importanti, soprattutto in caso di debug.

Per non dar retta alle asserzioni, quindi non farci ritornare “errori” dobbiamo inserire in compilazione l’opzione -DNDbug
Esso ci fa uscire dalla modalità di debug

test

Mettendosi nei panni di un utilizzatore scriviamo del codice di test delle singole funzionalità.
L’attività di sviluppo viene quindi “guidata” dai test.
L’attività di test dinamico (fatto a tempo di esecuzione) è essenziale per verificare che la classe si comporti correttamente nei casi “coperti”, ovvero dove assicuriamo un comportamento definito.
Per fare le cose per bene serve ovviamente un numero di test elevato che permetta di testare le varie evenienze fatto in modo sistematico.
Ovviamente non si riesce sempre a fare test esaustivi su tutto, **quindi raramente possiamo dimostrare che in quel codice non ci sono errori**

invariante di classe

Proprietà che deve essere soddisfatta dalla rappresentazione scelta per il tipo di dato quando è possibile è utile codificarne il controllo mediante un metodo della classe o un’asserzione.

Ovviamente queste proprietà di invarianza potrebbero variare ma solo per piccoli momenti, in genere si controlla dopo una modifica e/o prima di rilasciare il dato

Es: In nessun caso dobbiamo avere che il denominatore di un numero razionale sia 0, quindi dobbiamo controllare tale cosa ad ogni operazione che si fa

If(den!=0) throw //eccezione
Else { … resto del programma … }

precondizioni e postcondizioni

condizioni da controllare ogni qual volta si chiamano alcune funzioni, prima o dopo averle chiamate.

programma per contratto

Queste condizioni vengono racchiuse in un “programma per contratto”.

- precondizioni: delle richieste che l’utente utilizzatore deve soddisfare per poter utilizzare la funzione.
    - sono spesso difficili da automatizzare quindi si codificano dei test espliciti dove si controllano risultati attesi per determinati dati forniti in input
- postcondizioni: sono delle condizioni che lo sviluppatore deve garantire siano rispettate al seguito delle funzioni offerte.

Il contratto viene identificato come:
precondizioni → postcondizioni

ovvero, se le pre condizioni sono rispettate anche le post condizioni **devono** esserlo. Se invece le pre condizioni non sono rispettate allora lo sviluppatore non ha nessun obbligo nel fornire un giusto funzionamento.

Contratto migliorato

pre condizioni ed invarianti → post condizioni ed invarianti

contratti narrow e wide

Possiamo dividere i contratti in 2 categorie:

- narrow(stretti): si restringe l’insieme di valori che sono ammessi per invocare il metodo/la funzione
  Es: (x=5 || x=10).
  Definendo quindi un dominio di valori, sono i contratti più usati e consentono di fare meno test, quindi sono molto efficienti.
  Il controllo dei valori forniti ricade sull’utilizzatore

    - sono i più usati nelle librerie standard di c e c++

  Ambito precondizioni
- wide(larghi): controllare la legittimità delle invocazioni ricade sull'implementatore, ovvero l'utente può chiamare un metodo/funzione con un qualunque valore, e spetta all'implementatore controllare che questo sia valido gestendo eventuali eccezioni ecc

  Ambito postcondizioni

Analisi esercizi

gestione delle risorse

Risorsa: entità disponibile in quantità limitata, utile al software per il suo normale funzionamento.

Il software deve evitare di sprecare risorse, per farlo attuiamo delle "tattiche", gestiremo quindi le risorse secondo uno schema predefinito:

- acquisto la risorsa
- uso la risorsa
- rilascio la risorsa

E quindi:

- non possiamo usare/rilasciare la risorsa prima di averla acquisita
- al termine del suo uso la risorsa deve essere rilasciata
- non possiamo usare la risorsa dopo averla rilasciata

Gestione eccezioni

Esempi di risorsa

- **memoria dinamica**: l'allocazione in essa avviene tramite l'operatore new, la deallocazione tramite l'operatore delete, permette di gestire più autonomamente la vita delle nostre risorse, da essa derivano alcuni "problemi"
    - memory leak: mancato rilascio della risorsa esaurendo, a lungo andare, la memoria
    - double free: rilasciamo la risorsa anche se l'avevamo già rilasciata in precedenza
- **descrittori file del file system**: vengono acquisiti con l'apertura del file e vengono utilizzati per leggere/scrivere sul file e infine rilasciati alla chiusura di essi, la mancata chiusura potrebbe compromettere il contenuto del file

- **i lock per l'accesso**: acquisizione di lock per la gestione condivisa delle risorse se ci sono più thread/processi, il mancato rilascio può portare a deadlock o starvation

Ricordiamo però che non sempre le cose vanno bene, possono esserci delle eccezioni dovute a flussi di esecuzione non lineari detti "eccezionali"

Esempio	descrizione

`Void foo(){`
`  int * pi= new int (42);`
`fai_qualcosa(pi);`
`delete pi;`
`}`
	Questo sarebbe il flusso corretto di gestione, infatti:
	• acquisiamo la risorsa
	• usiamo la risorsa
	• rilasciamo la risorsa
	
	Immaginiamo però che la funzione "fai_qualcosa()" ritorni un'eccezione non controllata, finiremmo per non calcolare mai le istruzioni successive, quindi a non rilasciare mai la risorsa **ottenendo un memory leak**

gestione eccezioni

Questo codice non si comporta bene con le eccezioni, viene detto exception unsafe

Come facciamo a sistemare?
Attuiamo diverse tecniche.

Ricordiamo però che scrivere codice exception safe richiede sforzo programmativo, quindi in certe occasioni non ne vale più la pena e possiamo evitare di gestire le eccezioni.
Le situazioni in cui possiamo evitare sono:

- gestiamo risorse poco importanti (abbiamo tanta memoria quindi anche sprecandola non ci sarebbero problemi gravi)

- stiamo programmando un software che se anche si interrompesse non causerebbe problemi (ad esempio videogiochi)

tecniche di gestione

Ci sono diverse cose che potremmo fare, ad esempio:

- controllare l'acquisizione della risorsa prima di utilizzarla
- stare bene attenti al rilascio delle eccezioni, ad esempio se prendo A e B ma B non posso prenderla devo ricordarmi di rilasciare anche A, in caso contrario si finirebbe con il tenerla occupata creando sprechi

Per la tecnica migliore è quella di utilizzare gli strumenti appositi che ci fornisce il linguaggio, ovvero l'uso del try catch

gestione risorse con try catch

è composta da 2 parti:

- blocco try: si inseriscono al suo interno tutte le istruzioni, tra cui quella che può generare le eccezioni
  - al suo interno si possono lanciare eccezioni esplicitamente tramite la dicitura throw
- blocchi catch: tramite esso (o essi possono essere più di 1) si gestiscono le eccezioni generate nel blocco try

osservazioni

- si crea un blocco try/catch per ogni risorsa acquisita
- il blocco **try** si apre subito **dopo** l'acquisizione della risorsa anche perché se l'acquisizione fallisce non ci sarebbe nulla da fare nel catch
- la responsabilità del blocco try/catch è proteggere la relativa risorsa ignorando le altre
- al termine del blocco try(prima del catch) va effettuata la restituzione delle risorse (questo perché in caso non si generino eccezioni, non finendo nel catch dovremmo ugualmente rilasciare la risorsa)
- inserendo catch(...){azioni} possiamo catturare tutte le eccezioni (intendo i 3 punti nei parametri del catch)
- inserendo il "throw" alla fine di un blocco catch facciamo in modo di rilanciare l'eccezione in modo che venga presa dal catch dopo
- nel catch dobbiamo solo rilasciare la risorsa e fare throw garantendo la neutralità

esercizi a riguardo

Approccio alternativi

Come si può vedere, i punti non sono molto semplici da rispettare, quindi cerchiamo un'altra via.

Utilizziamo 2 approcci alternativi

- RAII: l'acquisizione delle risorse è una inizializzazione deve essere fatto all'interno della classe
- RAID: il rilascio delle risorse è una distruzione deve essere fatto nel distruttore

L'idea che ci sta dietro è incapsulare le risorse in un tipo definito da utente

- il costruttore del tipo acquisisce la risorsa (lancia la funzione dove se non la riesce ad acquisire lancia un'eccezione)
- il distruttore del tipo restituisce la risorsa (sempre tramite funzione)
- possiamo usare la risorsa tramite metodi della classe (ad esempio un get)

La cosa buona di questo approccio è che sia che si esca in modalità normale, sia con modalità eccezionale, si arriverà alla distruzione dell'oggetto, di conseguenza la risorsa verrà sempre rilasciata in modo adeguato.

exception safety

una porzione di codice si dice **exception safe** quando si comporta in maniera adeguata anche in presenza di comportamenti anomali segnalati al sistema tramite eccezione, ovvero se non compromettiamo lo stato del programma dopo tali anomalie
Esempi di compromissione: mancato rilascio delle risorse.

livelli

l'exception safety esiste in 3 livelli

livello base

se si verificano delle eccezioni valgono 3 condizioni:

- non si hanno perdite di risorse
- si è neutrali rispetto alle eccezioni
- anche in caso di uscita in modalità eccezionale devo comunque lasciare gli oggetti su cui lavorare in uno stato che consenta agli altri di distruggerli

livello forte

Richiede **tutti i requisiti del livello base** ma anche che:

- il codice si comporti con una sorta di atomicità delle operazioni, ovvero:
  - o acquisiamo tutte le risorse ed eseguiamo
  - o se non riusciamo ad acquisire anche solo 1 delle risorse che ci serve dobbiamo annullare le operazioni fatte (rilasciando tutte le risorse acquisite

e non usate)

livello nothrow

Si da la garanzia che non verrà mai eseguita una eccezione durante la sua esecuzione, quindi che vada tutto a buon fine e che **non si debba mai uscire in modalità eccezionale.**

Ovviamente questo livello è difficile se non impossibile da ottenere, alcune situazioni sono:

- in operazioni molto molto semplice
- se c'è un errore riusciamo a risolverlo senza chiamare eccezioni
- se l'errore è così critico da farci terminare direttamente

Possiamo definirlo con la parola noexcept dei linguaggi

I distruttori non devono mai lanciare eccezioni, fanno parte del livello nothrow e non è neutrale rispetto alle eccezioni

smart pointer

Abbiamo visto che attraverso l'utilizzo dell'idioma RAII e RAID riusciamo ad evitare molti dei problemi derivanti dall allocazione della memoria dinamica

Il problema con essi sarà il fatto di definire oggetti per ogni tipo.
Si utilizza quindi una classe templatica per la gestione dell'approccio RAII e RAID

Troviamo queste classi templatiche nelle librerie standard.
in queste librerie standard troveremo i puntatori "furbi" di 3 diverse categorie

1. **unique_ptr:** puntatori unici
2. **shared_ptr**: puntatori condivisi
3. **weak_ptr**: puntatori deboli

unique_ptr

gli unique ptr sono dei puntatori "intelligenti", attraverso di essi si implementa il concetto di puntatore proprietario, quindi che "possiede" la risorsa.
C'è un solo puntatore proprietario per risorsa che deve occuparsi di gestirla nel migliore dei modi, soprattutto in quanto unici proprietario **deve preoccuparsi di far rilasciare la risorsa al sistema una volta che non la utilizza più.**

```
std::unique_ptr <int> pi(new int (42))
```

Queste strutture dati forniscono gli operatori che li rendono utilizzabili come puntatori.

caratteristica

non sono copiabili ma solo spostabili.
Anche perché la copia li renderebbe “cooproprietari” di una risorsa, invalidando quindi la loro principale caratteristica

metodi

ci sono alcuni metodi che permettono a questi puntatori di interagire con quelli tradizionali:

- reset: pi.reset(raw_pi): prende la proprietà del puntatore passato (**non dobbiamo fare la delete su raw_pi dato che esso (essendo un puntatore normale) punterà allo spazio di memoria, quindi andrebbe ad invalidarlo**)
  - dove: int* raw_pi=new int(42)
- get: int* raw_pk=pi.get(): restituisce il puntatore canonico contenuto in pi, quindi fornisce un nuovo punto di accesso alla risorsa, ma la proprietà rimane sua (**quindi anche in questo caso non dobbiamo fare la delete su raw_pk**)
- int* raw_pj: pi.release(): pi rilascia la risorsa (perdendone la proprietà) fornendo il suo puntatore canonico come punto di accesso alla risorsa (**qui dato che non c’è più la proprietà dobbiamo fare delete su raw_pj**)

shared_ptr

In questo caso si usano sempre dei puntatori “intelligenti” ma questa volta c’è una condivisione della risorsa, e non più una proprietà.

```
std::shared_ptr<tipo> nome;
```

comportamento

Dovendo implementare il concetto di condivisione della risorsa gli shared_ptr basano il loro comportamento su un **contatore**

- Ogni risorsa puntata da uno shared_ptr ha un contatore al suo interno che
  - se aggiungiamo un nuovo shared_ptr che punta ad essa il contatore si incrementa
  - se rimuoviamo uno shared_ptr che punta ad essa il contatore si decrementa
    - **nel caso uno shared_ptr vede il contatore a 1 quando sta per decrementarlo (quindi intuitivamente è l’ultimo puntatore a quella risorsa) allora andrà a far rilasciare la risorsa al sistema**

```
void foo(){
```

```
    std::shared_ptr<int> pi;
    {
        std::shared_ptr<int> pj (new int(42));
    //il ref_count della locazione di memoria dove è 42 va a 1
    pi=pj; //il ref_count va a 2
    ++(*pi) //la variabile cambia valore da 42 a 43
    ++(*pj) //la variabile cambia valore da 43 a 44
}// qui pj esce dallo scope quindi ref_count va a 1
++(*pi) //la variabile cambia valore da 44 a 45
}
// qui pi esce dallo scope quindi ref_count va a 0, quindi la
risorsa viene rilasciata
```

metodi

- reset(): equivale a “distruggere” il puntatore facendolo puntare a nulla, quindi si deve di conseguenza decrementare il ref_count alla risorsa (e rilasciarla se siamo l’ultimo puntatore che la puntava)
- get(): restituisce il puntatore normale del nostro shared_ptr (come prima è un altro punto di accesso alla risorsa, non va fatto il delete su di esso o perdiamo la risorsa)
- move(): permette di spostare la nostra risorsa puntata tra due shared_ptr, dopo che viene spostata il puntatore su cui viene chiamata la move non gestisce più la risorsa

```
std::shared_ptr<int> pj(move(pi))
//pi non gestisce più nessuna risorsa
```

problema

primo

Con gli shared_ptr abbiamo un problema, ovvero che potremmo trovare la risorsa puntata e il contatore in 2 blocchi di memoria fisica separata, questo vuol dire che la lettura sarà inefficiente.

Per risolvere potremmo usare la make_shared.

Essa crea un unico blocco di memoria contenente la risorsa e il contatore ad essa associato

```
auto pj=std::make_shared<int>(42)
```

secondo
con i puntatori condivisi poteva crearsi un problema, immaginiamo le seguenti istruzioni

```
void foo(){
  bar(
    std::shared_ptr<int>(new int(42));
    std::shared_ptr<int>(new int(42));
  )
}
```

Quando chiamiamo una funzione la valutazione delle espressioni contenute nei suoi parametri avviene in un ordine non precisato, potrebbe quindi avvenire:

- eseguiamo il primo new int(42); senza eseguire la creazione dello shared_ptr associato
- eseguiamo il secondo new int(42), se questo cadesse in eccezione otterremo che la memoria occupata dal primo new non viene rilasciata (dato che non aveva un oggetto/puntatore associato) e otterremo un memory leak

andando ad eseguire invece tramite le make_shared, quindi:

```
void foo(){
  bar(
    std::make_shared<int>((42));
    std::make_shared<int>((42));
  )
}
```

questo problema non si porrebbe perché la make shared fa i passaggi insieme, crea lo shared_ptr e gli associa l'indirizzo a cui puntare, quindi eviteremmo il problema

In realtà questo problema con le nuove versioni di c++ è stato risolto senza l'uso delle make_shared
infatti nei nuovi standard di c++ quando valutiamo un'espressione (ad esempio la prima riga dentro bar) iniziamo sì dal new int, ma se facciamo quello il flusso di esecuzione obbligatoriamente prosegue creando il puntatore associato ad esso.

In generale con gli smart pointer si consiglia all'utente di evitare

esplicitamente new e delete, perché potrebbero portare problemi di memoria

weak_ptr

I weak_pointer vengono implementati per risolvere un problema degli shared_ptr, infatti essi potevano andare a creare una struttura ciclica, così facendo tutti i puntatori coinvolti avrebbero avuto almeno una puntatore che li puntava e non sarebbero mai "morti".

I weak_ptr sono sostanzialmente degli shared_ptr che però non partecipano attivamente alla gestione della risorsa, possiamo si usare la risorsa attraverso di essi ma non ci occupiamo di distruggerla quando smettiamo di puntargli.

vanno quindi usati insieme agli shared_ptr, se abbiamo 1 shared_ptr e un weak_ptr collegati alla risorsa, e lo shared_ptr smette di puntargli la risorsa dovrà essere ugualmente (nonostante la presenza del weak_ptr alla risorsa) distrutta.

Così facendo però ci troveremmo nel problema di puntare ad un indirizzo ormai vuoto, non inizializzato ecc..

lock

Il metodo lock è il metodo usato dai weak_ptr per controllare che la risorsa a cui puntavano esista ancora.
Dovrà essere usato ogni qual volta si vuole accedere alla risorsa (in lettura o scrittura) per controllare che un altro puntatore non l'abbia distrutta.

esso ritorna lo shared_ptr che punta alla risorsa, se non esiste nessuna risorsa a cui puntare allora ritornerà un null_ptr

```
void maybe_print(std::weak_ptr<int> wp) {
  if (auto sp2 = wp.lock())
    std::cout << *sp2;
  else
    std::cout << "non più disponibile";
}
```

categorie di espressioni

Le espressioni del c++ possono essere classificate in:

a) lvalue (left value): valore che sta a sinistra dell'assegnamento; locazione di memoria possiamo scrivere dei valori
b) xvalue (expiring value): un valore che sta per terminare il ciclo di vita (può essere sfruttato per altro)
    i) Es:
    ```
    int somma(){
        int x;
        return x;
    }
    ```
    In questo caso quando ritorniamo x quello diventa uno spazio di memoria non più utilizzato e quindi riutilizzabile.
c) prvalue (primitive value): valore che sta a destra dell'assegnamento

Es:

- x=57
    - x: lvalue
    - 57: rvalue
- int i, int Ai[100]; Ai[5]=7; i=5;
    - i e Ai[5] sono lvalue
    - 7 e 5 sono rvalue

d) l'unione di lvalue e xvalue forma i glvalue (generalized value)
e) l'unione di xvalue e prvalue forma gli rvalue (right value)

In alcuni casi i prvalue possono diventare "materializzati" creando un oggetto temporaneo (un lvalue) inizializzato con i prvalue.

Es:
ad una funzione con argomento un "riferimento costante" viene passato un prvalue
```
void foo(const double &a){}
foo(0.5);
```

riferimenti

è molto importante questa classificazione per capire la differenza tra:

- riferimenti lvalue (indicati con una sola & → t&)

- riferimenti rvalue (con 2& → t&&): introdotti nel c++ 2011 per risolvere problemi tecnici di efficienza

C++ 2003

Nel 2003 c++ era fornito di 4 funzioni speciali:

- costruttore di default
- costruttore di copia
- assegnamento per copia
- distruttore

Problema

Una funzione che avesse voluto prendere in input un oggetto di quel tipo e produrre in output una sua variante modificata (senza modificare l'oggetto in input) doveva tipicamente ricevere l'argomento per riferimento a costante e produrre il risultato per valore (per copia), questo ovviamente causava inefficienza.

Poiché non si poteva indicare:

- di non riutilizzare il parametro passato, in caso non servisse più
- di non passare per copia il valore ritornato

C++ 2011

Sono state aggiunte altre 2 funzioni speciali

- costruttore per spostamento
- assegnamento per spostamento

Prendono come parametro un riferimento r-value (t&&) ed essendo un expired rvalue può riutilizzare la sua memoria

- copia
  - costruttore di copia: Matrice(const matrice&)
  - assegnamento per copia: Matrice& operator=(const matrice&)
- spostamento
  - costruttore per spostamento: matrice(Matrice&&)
  - assegnamento per spostamento: Matrice& operator=(Matrice&&);

Esempio	
modo 1	**modo 2**

```
Matrice bar(const Matrice&
arg){
    Matrice res= arg //copia
    ...modifico di res...
    return res; //sposta, non
copia
}
```

```
Matrice bar(Matrice && arg){
//modifichiamo direttamente il
valore
...modifica "in loco" di
arg...
return arg; //sposta, non
copia
}
```

stdmove

Per indicare che una variabile può essere riutilizzata e quindi passarla come r-value si usa la funzione std::move()

Es:

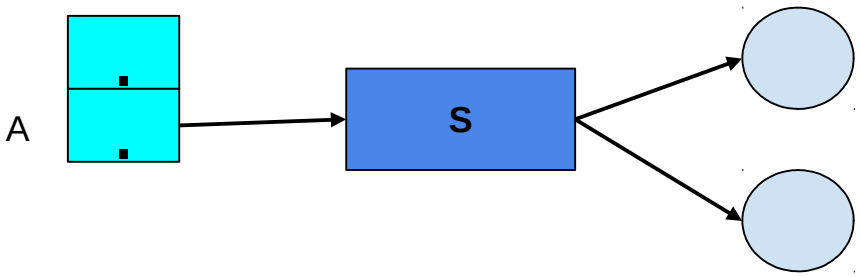
bar(std::move(M))

//fa usare M come r-value e non l-value

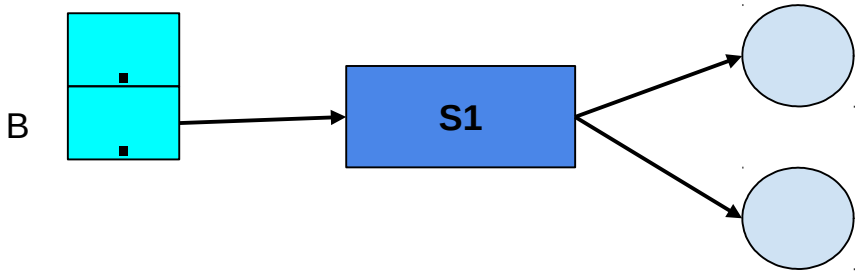
modalità di copia

Quando abbiamo una classe possiamo fare 2 tipi di copie

- copia profonda: copiamo tutti i riferimenti collegati



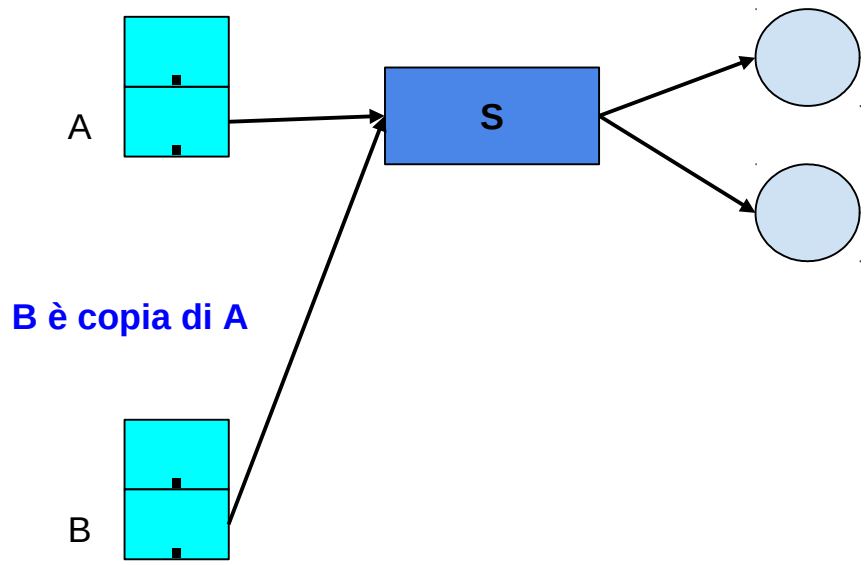
B è copia di A



- è la scelta migliore però effettivamente si spreca molta memoria e risorse

in generale

- shadow copy (riferimento al contenuto): si va a fare semplicemente un riferimento al contenuto dell'altro oggetto senza copiare effettivamente nulla



  - In questo caso però dovremo stare attenti perché se A o B vogliono cancellare il puntatore ad S non devono cancellare anche S stesso poiché l'altro rimarrebbe a puntare al nulla, ottenendo così un dangling pointer

Approccio in RAII e RAID

Seguendo l'approccio RAII e RAID dove trattiamo le risorse tramite oggetti di una classe creata per tale scopo dall'utente

disabilitare copia

Possiamo disabilitare i **costruttori** di copia (e quindi la copia profonda dell'elemento) inserendo:

Costruttore di copia

```
gestore_risorsa(const gestore_risorsa&)=delete;
```

//ipotetica classe gestore risorsa

l'assegnamento per copia

```
gestore_risorsa& operator=(cost gestore_risorsa&)=delete;
```

abilitare spostamento

Possiamo adesso aggiungere il **costruttore** per spostamento (quindi passare risorsa da un oggetto ad un altro)

Costruttore per spostamento

```
gestore_risorsa(gestore_risorsa&& y): res_ptr(y.res_ptr){
y.res_ptr=nullptr;
/*nella lista di inizializzazione andiamo ad assegnare il valore del puntatore dell'oggetto da copiare, e poi in quell'oggetto andiamo ad eliminare il puntatore, dato che è uno spostamento (quindi l'oggetto vecchio non deve più usare quella risorsa)*/
}
```

assegnamento per spostamento

```
gestore_risorsa& operator=(gestore_risorsa&& y){
    restituisci_risorsa(res_ptr);
    res_ptr=y.res_ptr;
    y.res_ptr=nullptr;
    return *this;
}
```

Analisi esercizi

lez9

stack.hh

lez10

teststack.cc
elem.cc

template

template di funzione

I template sono un costrutto del linguaggio c++ che permette di scrivere un modello (schema) parametrico per una funzione

Sintassi

dichiarazione

```
template <typename T>
/*potremmo indicare più di un tipo all'interno dei parametri
nelle <>, i vari tipi devono dividersi con , e inserire per
ognuno typename e nomedeltipo*/
T max (T a, T b){
    return (a>b)? a:b;
/*
questa istruzione valuta la "domanda" a>b e se vera ritorna il
primo elemento (prima dei due punti) se falsa ritorna quello
dopo i due punti

così facendo, otteniamo una funzione che può valere per
qualsiasi tipo, float, int, bool, double ecc...
Basta che gli si passi il tipo in questione tramite parametro
del template
*/
}
```

istanziazione

istanziazione di un template di funzione; come facciamo da questo schema generale a definire una funzione per un tipo definito?
Dobbiamo fornire gli argomenti della tipologia corretta ai vari parametri del template (quelli tra parentesi angolate)

```
Int maggiore= max<int>(5,4);
double maggiore= max<double>(5.2,4.7);
```

Si potrebbe anche evitare di passare il parametro di tipo al template, e lasciare che esso lo deduca dai parametri passati alla funzione, ciò però non è sempre conveniente, a volte il compilatore potrebbe non capire bene il tipo e sbagliare o andare in errore, ad esempio quando abbiamo 2 parametri di tipo differente

parti del template

- nome: abbiamo definito un template di una funzione di nome max
- parametri del template: T è un parametro del template, esso viene dichiarato come typename (nome di tipo), è analogo chiamarlo invece class, quest'ultima dicitura non viene usata perché crea confusione, facendo pensare i template siano utilizzabili solo per nomi di classi definite da utente, ma così non è, si possono usare tranquillamente anche tipi primitivi

Da notare

un template di funzioni non è una funzione, è più un generatore di funzioni, infatti quando chiamiamo una funzione in modo templatico stiamo parlando di una istanza di template.

Ovviamente la funzione potrebbe essere utilizzabile solo con certi tipi, ad esempio MAX pretende che per il tipo sia possibile utilizzare il confronto tramite l'operatore maggiore.

specializzazione

Per eventuali tipi da trattare diversamente possiamo usare una **specializzazione di template**
(ricordarsi che la specializzazione messa sempre dopo il template normale)

sintassi

Una specializzazione di template si specifica tramite la seguente sintassi:

```
template <>
tipo nome_funzione< tipo_da_specializzare >(parametri){
/*comportamento speciale*/
}
```

```
/*possiamo evitare di inserire il tipo da specializzare e
lasciare che il compilatore lo deduca da solo dai parametri
della funzione*/
```

Es:

```
template <>
const char* max <int> (const char* a, const char* b){
/*comportamento*/
}

/*si sta specializzando int per fargli avere un comportamento
speciale, diverso dal classico template*/
```

attenzione

Ricordiamo però che potremmo definire una funzione con lo stesso nome, al di fuori del template, in questo modo il template di funzione e la funzione fuori dal template andranno in overloading, ovviamente ciò complica le cose

```
template <typename T>
T max <T> (T a, T b){
/*comportamento*/
}

int max (int a, int b){
/*comportamento*/
}
```

istanziazione esplicita

Serve per separare il momento di creazione del codice di una particolare istanza del template dal momento in cui verrà usato

si divide in due parti:

- **dichiarazione**: la dicitura **extern** serve per indicare al compilatore di "prepararsi a creare" il codice per quando qualcuno (in un altro file) farà la definizione, senza però ancora crearlo

```
Extern template
    float max(float a, float b);
/*si nota però che non si inserisce <> con i relativi
parametri al template*/
```

- **definizione**: a differenza della dichiarazione diciamo appunto al compilatore di creare (in questo momento generiamo il codice) una particolare istanza con il tipo T corrispondente a float,
  template
  float max (float a, float b)

overloading con template

Quando abbiamo parlato dell'overloading di funzione abbiamo detto che si dovevano usare delle regole speciali per quando entravano in gioco anche i template di funzione.

Vediamo ad esempio una situazione in cui potremmo avere problemi

`template <typename T>`
`void foo(T t1, T t2){ ... }`
`template <typename T, typename U>`
`void foo(T t1, U u1){ ... }`

Se andassimo a chiamare la funzione foo<int>(5,5) oppure foo<int,int>(5,5); avremmo il problema che non sapremmo riconoscere quale delle due chiamiamo dato che si va in overloading

Quello che faremo è di ordinare i template per specificità, e di conseguenza trovare un ordine nella risoluzione dell'overloading.

ordinamento parziale

Definiremo in questo paragrafo quando un template è più specifico
Denotiamo con:

- istanze(x): l'insieme di tutte le possibili istanze del template x
- istanze(y): l'insieme di tutte le possibili istanze del template y

Diciamo quindi che il template di funzione x è più specifico del template di funzione y quando istanze(x) è un sottoinsieme proprio di istanze(y)
(ovvero andando ad espandere tutte le possibilità x può avere meno elementi e comunque sono tutti elementi che può avere y (y ovviamente ne può avere anche

altri))

Ritornando all'esempio di prima, il primo template (con un solo parametro) è più specifico rispetto al secondo, dato che tutte le possibili funzioni create da esso possono essere create anche dal secondo, e il secondo ne ha anche altri

regole speciali per il template di funzione

se ci sono più template che sono sceglibili quando istanziamo la nostra funzione **andremo a scegliere quello il più specifico**, questo perché così facendo è meno probabile che si ottengano situazioni che portano ad errori (magari perché ci imbattiamo in casi speciali)

di base un'istanza di template entra nelle candidate quando si può fare la deduzione dei parametri del template, cioè se riesco a istanziare quel template facendo la deduzione in maniera corretta.
durante il processo di deduzione non possiamo applicare promozioni, conversioni standard ed esplicite, ma solamente le corrispondenze esatte.
Se essa rientra nelle utilizzabili allora il processo di deduzione va a buon fine

a questo punto tra le tante istanze di funzione prendiamo quella più specifica.
Dato che si vogliono rimuovere tutte le ambiguità finché possibile, avendo a disposizione diverse funzioni tra cui scegliere, **in caso di ambiguità si andrà a dare la precedenza a quelle non templatiche**

template di classe

Analogamente a quanto detto prima per quello di funzione fornisce un generatore di istanze di classi secondo alcuni parametri.

Sintassi

dichiarazione

```
template <typename T>
class nome_classe{
    T nome;
    T get_name(){return nome;}
}
```

istanziazione

istanziazione di un template di classe

Ovviamente, nel template di classe non forniamo alcun parametro all'oggetto, quindi non potremo evitare di scrivere il tipo del template

```
- Stack <int> uno; //ok
- Stack due=uno; //no, dobbiamo ugualmente passare il param <int>
- auto due=uno; //ok
  /*
  il c++ 2011 implementa la parola auto che permette di capire automaticamente il tipo del template
  */
```

casi speciali

omettere parametro

C'è un caso in cui il punto due va bene, ovvero quando lo utilizziamo all'interno dello stesso template di classe, in quel caso è come se venisse auto completato

```
template <typename T>
class stack{
    stack& operator=(const stack &);
    // è come se venisse scritto stack<T>& operator=(const stack<T>&);
}
```

definizione funzione fuori dalla classe

Quando vogliamo inserire una funzione della classe esternamente alla classe dobbiamo fare attenzione.

```
template <typename T>
    stack<T>& stack<T>::operator=(const stack& y) ...
/*
nei primi stack dobbiamo inserire <T> dato che siamo fuori dalla classe, quindi dobbiamo definire a quale istanza del template ci riferiamo.
```

```
In quello nel parametro non serve inserirlo perché siamo già interni allo scope (dato che abbiamo appunto passato lo scope operator :: )
*/
```

Da notare

Vengono istanziate in memoria solo le istanze dei tipi richiesti, non le altre.
è un bene perché **velocizza e alleggerisce il codice**, anche un male perché **se ci sono errori in una specifica istanza del template ce ne accorgeremo solo quando la istanzieremo effettivamente**

Specializzazione

Al contrario del template funzione ci sono 2 tipi di specializzazione:

- totale/completa: analoga a quella di funzione
  - esiste una libreria standard chiamata limits che fornisce un template di classe numeric_limits, attraverso il quale si possono avere informazioni sui tipi built-in (primitivi)

  Sintassi: Si usa la stessa sintassi delle funzioni

  ```
  template <>
  class nomeclasse<tipo_da_specializzare>
  {comportamento_speciale}
  ```
- specializzazione parziale: sono specializzazioni esclusive del template di classe (non è possibile farle in quello di funzione), si definiscono solo alcuni parametri del template quindi divento sempre un template di classe ma meno generale, dato che alcuni parametri sono già stati forniti
  Questo tipo di specializzazione è la meno frequente

Analisi esercizio

Analisi esercizio stack_templ.hh

processo di compilazione dei template

Il processo di compilazione dei template richiede lo stesso codice in almeno 2 contesti distinti:

1. al momento della definizione del template
2. al momento dell'istanziazione del template

nella prima fase indicata il compilatore si trova a dover operare con informazioni incomplete.
Es:

```
template <typename T>
void incr(int &i; T& t){
++i; //A
++t; //B
}
```

Nel caso:

- A: il compilatore saprà come fare l'operazione su int con relativi controlli ecc…
- B: nel secondo caso no, quindi dovrà rimandare alla fase di inizializzazione del template quando gli sarà fornito il tipo T, da quel momento potrà fare i controlli del caso

Questo vuol dire che quando definiamo un template dovremo far si che esso sia disponibile in tutti i punti in cui viene richiesta la sua istanziazione.
Quando si parlava di funzioni normali invece veniva richiesta che **solo la firma** fosse disponibile.
I modi tipici per organizzare i template sono:

- includere le definizioni di template (compresa la definizione di eventuali funzioni membro dei template classe) prima di ogni loro uso nelle unità di traduzione.
    - abbiamo già visto ciò per funzioni inline, ad esempio in alcune delle librerie standard
- includere le dichiarazioni del template (comprese le dichiarazioni di eventuali funzioni membro di template classe) prima di farne uso e successivamente (prima o dopo gli usi) includere le definizioni del template nelle unità di traduzione

Un'altra conseguenza è quella di dover fare delle piccole modifiche al codice quando lo templatizziamo, in caso contrario, a causa di quanto detto potrebbero esserci errori

Es:

```
struct S {
  using value_type = /* ... dettaglio implementativo ... */;
  /* ... */
};

void foo(const S& s) {
  S::value_type* ptr;
  /* ... */
}
```

Supponiamo ora di voler templatizzare la classe S, rendendola parametrica rispetto ad un qualche tipo usato al suo interno. Intuitivamente, il processo di "lifting" porterebbe ad adattare il codice in questo modo:

```
template <typename T>
struct S {
  using value_type = /* ... dettaglio implementativo ... */;
  /* ... */
};

template <typename T>
void foo(const S<T>& s) {
  S<T>::value_type* ptr;   // errore: ptr non dichiarato
/* da errore perché il template viene passato a tempo di
esecuzione quindi il compilatore non sa come comportarsi, cioè
potrebbe essere che il tipo passato al template faccia andare
in una “specializzazione” e che in essa valuetype non è un
tipo ma un valore*/
}
```

Per risolvere il problema e comunicare correttamente le nostre intenzioni al compilatore, occorre informarlo che S<T>::value_type indica il nome di un tipo, aggiungendo la parola chiave "typename":

```
template <typename T>
void foo(const S<T>& s) {
  typename S<T>::value_type* ptr;   // ok, dichiaro un
puntatore
  /* ... */
```

```
}
```

consiglio

in generale per i codici templatici il consiglio **è di scrivere tanti test** per testare quanto più possibile le classi in tutti i casi.

e come visto sopra se dobbiamo riferirci ad un tipo è consigliabile notificarlo all'operatore tramite **typename**

polimorfismo statico

Polimorfismo statico

Con il polimorfismo statico si va ad usare la logica secondo la quale si scrive una sola versione di codice che però possa essere utilizzata per generare varianti diverse

La scelta delle istanze da generare viene fatta a tempo di compilazione (**fase statica**)

La programmazione templatica (generica) de c++ si basa sul polimorfismo statico

Per ottenere una buona generalità del codice serve coordinare diversi elementi tra di loro (ovviamente perché in un contesto funziona tutto meglio)

Contenitori

Classi che hanno lo scopo di contenere collezioni di oggetti, quest'ultimi sono spessi chiamati elementi del contenitore.
Il tipo del contenitore è arbitrariamente scelto dal programmatore, quindi tramite template.
I contenitori si organizzano in base al modo in cui organizzano i dati

contenitori sequenziali

Si chiamano sequenziali quei contenitori i quali elementi sono disposti in sequenza.
Ovvero secondo un ordine, 1° elemento 2° elemento 3° …
L'ordine è semplicemente quello di inserimento dell'elemento

Quelli della libreria standard

I contenitori della libreria standard sono:
(fanno tutti parte del namespace std, quindi prima di ognuno di essi va usato std::)

- vector<T>: sequenza di dimensione variabile (a tempo di esecuzione) di elementi di tipo T. (con a tempo di esecuzione intendiamo che durante l'esecuzione del programma, il vector può ridimensionare i suoi spazi ottenendo nuova memoria o rilasciandone pezzi se quella che ha è troppo/poca per quello che deve fare)
  Fornisce **accesso a tempo costante**
  - inserimento e rimozione: sono efficienti solo se fatti in cima/coda alla sequenza di elementi, se fatti in mezzo (o l'inserimento in cima) si richiede un numero lineare di spostamenti (semplicemente per spostare gli elementi restanti un posto più avanti/indietro)
- deque<T>: implementazione della coda a doppia entrata, ovvero una coda dove si può entrare sia dall'inizio che dalla fine.
  é molto simile al vector (però non si possono inserire elementi a "metà" della sequenza, solo in cima o in coda), però al contrario di esso non è necessario inserire gli elementi in memoria in modo contiguo, per tale motivo l'inserimento in cima è più efficiente
- list<T>: liste doppiamente concatenate (ci si può muovere sia avanti che indietro tramite i puntatori)
  - punto negativo: accesso sequenziale dalla testa o dalla coda.
  - punto positivo: l'inserimento/rimozione a "metà" sequenza è più facile e meno costoso dato che basta gestire bene i puntatori
- forwardlist<T>: una lista che permette di scorrere solo in avanti, ha gli stessi punti a favore e contro della precedente (ovviamente l'accesso può essere fatto quindi solo dalla testa, forse può essere rimosso l'elemento in coda)

Pseudo contenitori

- array<T,N>: dove N è costante, cioè un valore, non un tipo.
  Sono semplici array (non ridimensionabili) con N elementi e di tipo T, possiamo trovare facilmente la loro dimensione
- string: sequenza di caratteri char, in realtà std::string è un alias per l'istanza std::basic_string<char>, possiamo quindi istanziare il template basic_string con altri tipi
- bitset<N>: sequenza di N bit (con N sempre valore costante, non tipo), torna utile quando si vogliono fare degli elenchi di marcatori per eventi, e quindi si segna 1 se l'elemento corrispondente a quel bit rientra nell'evento, 0 altrimenti

differenze

Questi contenitori appena visti sono simili ma non identici, andremo a guardare diverse caratteristiche:

- costruttori
- operatori per interrogare
- operatori per consentire

Possiamo controllare le info di questi contenitori su cppreference.com
Lì troviamo tutte le librerie standard, come sono formate ecc…
Tra esse quindi troviamo anche questi con i vari parametri interni, tra questi parametri troviamo:

- iteratori: serve per iterare la sequenza di elementi, ne esistono di vari tipi, vengono differenziati dalla “modificabilità” degli elementi (se hanno const ecc) e dal fatto di poter o meno scorrere la sequenza al contrario
- costruttori: ci sono vari costruttori all’interno di questi contenitori, tra cui quelli che per parametro accettano coppie di iteratori, spesso non viene definito il tipo della sequenza seguita dall’iteratore, quindi potenzialmente possiamo fare cose come riempire il nostro vector con elementi di un vector di un altro tipo
- vari metodi come:
    - size: dice il numero degli elementi nella sequenza
    - capacity: dice il numero massimo di elementi che possono entrare nella sequenza
    - empty: dice se la sequenza è vuota o no

Si noti però che mancano metodi generali come sort, find, ecc..
poiché si vuole far presente che sono metodi che si consiglia di creare 1 sola volta per strutture generiche, non per ogni struttura differente. (appunto basandosi sul principio di polimorfismo statico)

contenitori associativi

i contenitori associativi sono dei contenitori che organizzano gli elementi al loro interno in modo da facilitare la ricerca in base al valore di una “chiave”.
Esistono i seguenti contenitori

1. std::set<Key, Cmp>
2. std::multiset<Key, Cmp>
3. std::map<Key, Mapped, Cmp>

4. std::multimap<Key, Mapped, Cmp>
5. std::unordered_set<Key, Hash, Equal>
6. std::unordered_multiset<Key, Hash, Equal>
7. std::unordered_map<Key, Mapped, Hash, Equal>
8. std::unordered_multimap<Key, Mapped, Hash, Equal>

queste otto categorie di contenitori si ottengono combinando in vario modo tre distinte possibili proprietà:

A) La presenza (o assenza) negli elementi di ulteriori informazioni oltre alla chiave utilizzata per effettuare le associazioni.
   Ovvero se il tipo dell'elemento è formato solo dalla chiave (key) abbiamo contenitori detti insiemi "**set**", altrimenti utilizziamo contenitori detti mappe "**map**", essi uniscono appunto valori o di tipo set o di tipo map
B) La seconda differenza è la possibilità di memorizzare nel mio contenitore più elementi con lo stesso valore di chiave
   a) Ad esempio pensando ad un videogioco potremmo avere massimo 1 giocatore per livello, oppure più giocatori per livello, in questo caso il livello è la nostra key

   Se si possono memorizzare più elementi si parla di "**multi**", quindi multiinsieme multi mappe ecc
C) ultima differenza, è il criterio di ordinamento delle chiavi usato all'interno del contenitore, ne esistono diversi:
   a) cmp: quindi in un ordine delle key alfabetico, numerico ecc (solitamente questi contenitori si basano su alberi bilanciati di ricerca), in questi contenitori si richiede che il costo della ricerca sia O(log(n))
   b) hash: attraverso una opportuna funzione hash (passata come parametro) si ricava una posizione nella quale verificare se è contenuto l'elemento cercato, si basa su strutture unordered (ovvero strutture senza ordinamento, come unico ordinamento hanno quello dato dalla funzione hash che possiamo definire quasi casuale).

Per usare quindi le strutture che ne derivano, cercando sulla documentazione, vediamo che è un contenitore che accetta in input 3 parametri:

I. tipo della chiave
II. tipo del valore mappato (usato appunto solo nelle mapped)
III. tipo di confronto, di base il valore è un callable, ovvero un std::less

se utilizziamo un algoritmo di confronto scelto da noi dobbiamo rispettare una regola, ovvero deve essere un algoritmo di confronto stretto debole, questo perché 2 oggetti (A,B) saranno considerati uguali se andando ad utilizzare l'algoritmo di confronto tra di loro A non è minore di B e B non è minore di A, in tal caso li definiremo uguali

funzioni e operatori

ci sono diverse funzioni e operatori all'interno di questi contenitori speciali

- operator[]: prende in input una chiave
  - se esiste un elemento con quella chiave restituisce un riferimento ad esso
  - se non esiste crea un elemento con quella chiave e restituisce il riferimento
- count: conta gli elementi con un certo valore di chiave all'interno del contenitore

esercizi

- map-numeroparole

analisi esercizi

Analisi librerie standard

- **primo**
- **secondo**
- **terzo**

informazioni

differenza tipo/concetto

Differenza tra tipo e concetto: il concetto è qualcosa di astratto ed è implementato mediante l'uso di tipi che utilizzano quella tecnica (elemento)

in generale per controllare che un tipo implementi un concetto si danno dei requisiti, se li soddisfa allora lo implementa.

for

Possiamo inserire diverse istruzioni in ogni parte dei parametri del for separando le istruzioni con la virgola.
Ovviamente ogni istruzione va nella parte giusta dei parametri, quindi:

- 1° parte: la dichiarazione/valorizzazione di variabili usate nel for
- 2° parte: controllo da fare affinché il for sia eseguito
- 3° parte: modifica valore delle variabili

Es: for (int a=0, int b=5; b<10; a++, b--){ … }

iteratori

iteratori

Molti iteratori lavorano sul concetto di sequenza, in generale il concetto di iteratore prende spunto da quello di puntatore

categorie

possiamo classificare gli iteratori in 5 categorie distinte, **si differenziano per operazioni supportate e per le corrispondenti garanzie fornite all'utente**

1) **iteratori di input**
2) **iteratori forward**
3) **iteratori bidirezionali**
4) **iteratori random access**
5) **iteratori di output**

Come convenzione si usa che per ogni categoria di iteratore si usi per il loro nome un qualcosa che riporti alla categoria di appartenenza.
Es: in_iter per quelli di input

iteratori di input

Gli iteratori di input permettono di effettuare solamente le seguenti operazioni:

- ++iter: avanza di una posizione nella sequenza di elementi
- iter++: idem ma postfisso (si consiglia di usare sempre la forma ++iter, in generale sempre quando si fa l'incremento di qualcosa, anche semplici int)
- *iter: accesso **in sola lettura** all'elemento corrente (vediamo come è analogo alla sintassi dei puntatori)
- iter->m equivale a (*iter).m e come per i puntatori si assume che l'elemento "puntato" dall'iteratore abbia al suo interno (quindi l'elemento in sé è una struttura/classe) un elemento di nome m, accediamo quindi ad esso
- iter==iter2: confronto di uguaglianza, tipicamente si usa per controllare se siamo giunti alla fine di una sequenza (si mette iter2 puntante alla fine della sequenza, e con iter andiamo avanti man mano, tramite questo controllo ci accorgeremo quando avremo finito)
- iter!=iter2: confronto di disuguaglianza

problema

Negli iteratori di input bisogna fare attenzione al fatto che **ogni qual volta si**

incrementa un iteratore che punta ad un elemento si vanno ad invalidare tutti gli altri iteratori che puntavano ad esso

Esempio

```
std::istream_iterator <double> i(std::cin);
auto j=i;
cout<<*i;
cout<<*j;
//stampano la stessa cosa
++i; //ci muoviamo all'elemento successivo
cout<<*j; //errore: comportamento non definito
```

Possiamo dire che ogni qualvolta andiamo avanti venga “consumato” il carattere precedente.

in_iter

iteratori forward

Gli iteratori forward sono analoghi a quelli di input, però a differenza di essi dopo l’incremento non si invalidano tutti gli altri iteratori collegati all’elemento, questo rende possibile scorrere più volta una sequenza.

per questo sono gli iteratori associati alle forward list

fwd_iter

iteratori bidirezionali

Analoghi ai forward iterator, però consentono di scorrere la sequenza anche in senso contrario (andare indietro).
Per farlo implementano le operazioni:

- --iter; come detto per l’incremento si consiglia di usare questo
- iter--;

Usati ad esempio nelle liste doppiamente concatenate

bi_iter

iteratori random access

Può fare tutto ciò che possono fare quelli bidirezionali, ma aggiunge anche le seguenti operazioni:

- iter+=n: fa andare avanti (se n è positivo) o indietro (se n è negativo) di n elementi il nostro iteratore
- iter -=n: fa andare indietro (se n è positivo) o avanti (se n è negativo) di n elementi il nostro iteratore
- iter+n: calcola un nuovo iteratore spostato di n posizioni
    - n+iter: equivalente al precedente
- iter-n: analogo a quelli precedenti ma nella direzione opposta
- iter[n]: equivale a *(iter+n), ovvero leggere l'elemento di n posizioni dopo (o prima, dipende dal segno di n) iter
- iter-iter2: calcola il numero di elementi che separano i 2 iteratori, ovviamente essi devono essere definiti sulla stessa sequenza di elementi
- iter<iter2: restituisce
    - true se iter occorre prima di iter2 nella sequenza di elementi puntata
    - false se al contrario occorre prima iter2
- iter>iter2: analogo ma inverso
- iter<=iter2: analogo ma conta anche l'uguaglianza
- iter>=iter2: analogo ma conta anche l'uguaglianza

utilizzati per vector, coda, stringa, array

iteratori di output

iteratori che permettono solamente scritture sugli elementi di una sequenza
se si scrive un elemento l'iteratore **deve** essere aumentato
permette le seguenti operazioni:

- ++iter
- iter++
- *iter: accesso **in sola scrittura** all'elemento puntato

da notare

quando il tipo degli oggetti "puntati" è accessibile in scrittura gli iteratori forward, bidirezionali e random access soddisfano i requisiti degli iteratori di output, quindi possono essere usati ovunque sia necessario fornire un operatore di output

problema

Gli iteratori di output lavorano in modalità "sovrascrittura", ovvero:

- immaginando la sequenza: abcd
- puntiamo con l'iteratore nella seconda casella (b), se volessimo scrivere

andremmo a sovrapporre all'elemento precedente di quella casella

- Es: se volessimo scrivere e, otterremmo: aecd

inserter

Possiamo fare in modo che ciò che scriviamo non vada a sovrapporsi ma vada a spostare gli elementi di un posto (inserendosi quindi idealmente tra la casella prima e quella selezionata).
Per fare ciò le librerie standard forniscono una classe templatica che può essere istanziata con l'iterazione di output di un certo tipo.
Tale classe trasforma l'iteratore da modalità sovrascrittura a modalità inserimento
Gli iteratori che lavorano per inserimento si chiamano "**inseritori**" (inserter), gli inseritori sono definiti in un header file delle librerie standard.

Esistono diversi tipi di inseritori:

back_insert_iterator

La classe **back_insert_iterator** è una classe templatica istanziata con un tipo di contenitore, memorizza un puntatore al tipo del contenitore nella parte private dalle classe, salvandolo nella variabile "contenitore".

Quando andiamo a dare l'operazione di = non facciamo altro che la pushback alla struttura tramite il puntatore salvato prima (per questo motivo il metodo pushback è in tutti i contenitori) (andiamo quindi ad accodare un altro elemento)

```
back_insert_iterator& operator=(typename
_container::const_reference __value){
    container->pushback(__value);
return this;
}
```

Per tutte le altre operazioni (*(dereferenziazione),++iter,iter++) non facciamo nulla, semplicemente facciamo "return this;"

front_insert_iterator

Il front_insert_iterator funziona in modo analogo, però al posto di lavorare con le push_back lavora con le push_front.

(ricordiamoci che "vector" non ha a disposizione questo metodo, quindi

usandola su di esso otterremmo errore)

insert_iterator

L'insert iterator è una classe che permette di inserire gli elementi in una posizione specifica delle strutture spostando gli elementi (che seguono quella posizione) avanti

A differenza degli altri inserter, questo non prende in input solo un puntatore alla struttura ma **prende anche un iteratore che punta alla posizione in cui inserire l'elemento**

Come negli altri inseritori l'unica operazione a fare effettivamente qualcosa è **l'operatore =**, esso **richiama tramite il puntatore alla struttura la funzione insert passandogli in input la posizione in cui inserire e il valore da inserire.**
(come prima il resto di operazioni fa semplicemente un return this)

ostream_iterator

fornisce un iteratore di output per le stream
Accetta come input:

- un puntatore al nostro stream
- un puntatore ad un carattere (che verrà usato come separatore)

Come per gli altri l'unica operazione a funzionare è l'operatore = che affidandosi al nostro canale di stream out stampa i caratteri a cui punta su quel canale

problema

Quando utilizziamo i nostri inserter dobbiamo specificare il tipo di struttura utilizzata ogni qual volta la usiamo (vector, list ecc).

Le librerie standard però ci vengono incontro aiutandoci ad evitare di scrivere tutte queste informazioni che potrebbero portarci a sbagliare facendole dedurre in automatico al compilatore
tutto questo viene fatto attraverso la classe back_inserter (analogamente per gli altri avremo front_inserter ecc)

```
std::back_insert_iterator<std::vector<double>> out(vd);
//è analogo a
auto out =std::back_inserter(vd)
//è facile notare come così facendo andiamo a togliere molte problematiche logiche durante la creazione degli inseritori.
```

```
//ad esempio sicuramente se si dovesse passare come parametro sarebbe più comodo da passare senza creare una variabile
copy_if(vi.begin(),vi.end(),std::back_inserter(vd),pari)
```

relazioni di ereditarietà

Tra gli iteratori abbiamo delle relazioni di ereditarietà:

- l'input iterator: deriva dal forward iterator
- il forward iterator deriva dall'iteratore bidirezionale
- il bidirezionale deriva dal random iterator

operazioni

Nelle varie categorie di iteratori ci sono diverse implementazioni per le operazioni (intendo iter++ ecc), in realtà non si definiscono le operazioni per tutti gli iteratori ma solo per alcuni, si va a sfruttare infatti l'ereditarietà degli iteratori risolvendo l'overload di funzione per utilizzare quella corretta.

Ad esempio la classe di iteratori forward non ha una definizione per l'operazione +n, va ad utilizzare quella della classe bidirectional da cui deriva.

Ovviamente l'overloading è un buono strumento ma può anche portare problemi, quello che possiamo fare per differenziare bene le funzioni e chiamare sempre quelle giuste sono:

- chiamare le funzioni specializzate (per un certo tipo) con un nome diverso, ad esempio "foo_casospeciale" così da ritrovarla sempre
- oppure creare le funzioni con un altro parametro oltre quelli richiesto, quest'ultimo sarà un tipo e serve per differenziare i vari tipi di iteratori, così da saperli riconoscere

iterator_traits

Esiste la classe iterator_traits che ci permette di ricevere alcune informazioni sui nostri iteratori, la categoria a cui appartiene, il tipo di valori puntati ecc…

abbiamo poi 3 specializzazioni parziali di questa classe:

1) gli iteratori sono delle struct/classi, semplicemente per tutte le richieste ritorna la risposta della classe principale

2) gli iteratori puntano ad un oggetto di un tipo definito (lo chiameremo tp), il problema è che essendo un puntatore non è a conoscenza delle informazioni, quindi fare le operazioni sarà più complicato rispetto a prima
3) uguale al 2° però è un puntatore a costante, quindi sarà impossibile cambiare valori

esempi di iterator stream

Possiamo vedere gli stream di input e output come una sequenza di cose da iterare tramite appunto gli iteratori.

```
#include <iterator>
std::istream_iterator<std::string> first(std::cin);
std::istream_iterator<std::string> last;
//per convenzione il costruttore senza parametri (di default) punta alla fine dell'input

std::ostream_iterator<std::string> out(std::cout,"\n")
//quel "\n" indica l'elemento nel quale separare una stampa e l'altra
```

analisi esercizi

file icecream lez 15

callable

callable

il callable definisce un “concetto”, ovvero tutto ciò che può essere “invocato” come se fosse una funzione.

gli algoritmi infatti in genere sono di due versioni possibili:

- una dove vengono implementati semplicemente
- una versione parametrizzata rispetto ai comportamenti specifici da applicare

Ci chiediamo quindi quali sono i tipi di dato che consentono di essere invocati, come funzioni. ce ne sono 3 categorie:

puntatori a funzione

Immaginiamo di avere una funzione: **bool pari(int i);**
Può essere chiamata attraverso un puntatore a funzione dal tipo:

- bool (*nome_puntatore_creato)(int): puntatore che se viene dereferenziato ottiene una funzione che prende un intero e ritorna un bool
    - non va bene bool * (int): perché così' facendo otterremo una funzione che prende un intero e restituisce **un puntatore ad un bool**

Es:

- bool (*puntatore)(int)
  //puntatore sarà il nome del puntatore che stiamo creando, un puntatore di tipo bool che punta ad una funzione che prende un parametro int.
- puntatore=pari;//avremo che il nostro puntatore adesso punta a pari, potremo quindi usarlo come se fosse pari

oggetti a funzione

Sono dei dati che hanno la possibilità di essere usati come funzioni.
Ne fanno parte tutte quelle classi che contengono l'overloading dell'operatore ().
Es:

```
struct pari{
    bool operator() (int i)const {return i%2==0;}
}
int foo(){
    pari pari;
    if(pari(1234)){ ... }
}
```

é molto comodo perché offre flessibilità:

- ci permette di fare facilmente overloading di funzioni()
- o ad esempio se vogliamo avere informazioni su alcuni elementi della nostra struttura potremmo chiedere semplicemente così

ottimizzazione

Oltretutto ci rende comoda l'ottimizzazione per alcune funzioni che ricercando secondo criteri, ad esempio la find_if, noi sappiamo essere della forma:

```
std::find_if<std::vector<int>::const_iterator,bool(*)(int)>
```

Per tutte le chiamate verrà quindi istanziata una sola volta e richiamata con diversi valori.

Immaginiamo invece di chiamarla con degli oggetti a funzione che implementano richieste diverse, ad esempio:
bool pari(int i)
bool dispari(int i)
bool positivo(int i)
bool negativo(int i)

si dovranno implementare diverse istanze della find_if

```
iter std::find_if<iter,pari>(iter first, iter last, pari pred)
iter std::find_if<iter,dispari>(iter first, iter last, dispari
pred)
iter std::find_if<iter,positivo>(iter first, iter last,
positivo pred)
```

e di conseguenza il compilatore potrà sfruttare questa cosa per ottimizzare i tempi
da un certo punto di vista lavora di più sprecando memoria per diverse istanze, però può poi ottimizzare il tutto

Gli oggetti a funzione spesso non vengono usati perché sono lunghi da implementare, si cercano quindi escamotage, spesso arrivando a problemi.
Da ciò ne deriva il 3° tipo di callable

espressioni lambda

deriva dalla teoria della programmazione funzionale e denota una funzione senza nome.

Sintassi

```
[](const long& i){return i%2==0}
```

- []: capture list, lista delle catture, non è sempre vuota, potremmo usarla quando l'espressione lambda deve **accedere a valori locali visibili nel punto in cui la creiamo** (il punto di creazione è diverso da quello di invocazione)
  Es:

  ```
  void foo(const std::vector<long>& v, long soglia){
      auto iter=std::find_if(v.begin(),v.end(),[soglia]
  (const long& i){return i>soglia;});

  //in questo caso, dato che viene creata all'interno dei
  parametri dell find_if dobbiamo esplicitare di catturare
  la variabile soglia
  }
  ```

  Potremmo vedere tutto questo come se venisse fatta la seguente operazione

  ```
  struct nome_indefinito{
  ```

```
    long soglia;
    nome_indefinito (long s): soglia(s){};
bool operator() (const long&i){return i>soglia;}
}
//e poi richiamato così:
auto
iter=std::find_if(v.begin(),v.end(),nome_indefinito(soglia));
```

- Di base andando a passare i parametri nella **capture list** li passeremo per copia, possiamo per passarli anche in modo diverso con:
  - [&soglia]: passiamo per riferimento

    ```
    long& soglia;
    nome_indefinito (long& s): soglia(s){};
    ```
  - [=soglia]: passiamo esplicitamente per valore
- Possiamo anche evitare di passare tutti i parametri nella capture list, possiamo dire al compilatore di dedurli
  - [=] diciamo al compilatore quali parametri catturare dal corpo della funzione e di catturarli tutti per valore
    - [=,&pippo,&pluto]: chiediamo sempre di capire quali parametri catturare e di catturarli per valore, ad eccezione di pippo e pluto che deve catturare per riferimento
  - [&] diciamo al compilatore quali parametri catturare dal corpo della funzione e di catturarli tutti per riferimento
    - [&,=pippo,=pluto]: chiediamo sempre di capire quali parametri catturare e di catturarli per riferimento, ad eccezione di pippo e pluto che deve catturare per valore
- Però si consiglia comunque di specificare parametro per parametro, per evitare errori o “incomprensioni del compilatore”

mutable

Anche se chiamiamo per riferimento non possiamo lavorare in scrittura modificando il valore, dato che il corpo della funzione è “const”, quindi si lavora in sola lettura.

Se volessimo ugualmente lavorare in scrittura dovremmo usare la dicitura “**mutable**” prima del corpo della funzione.
serve per dire che possiamo modificare nonostante il const (è una specie di controsenso), quindi va usato poco

```
void foo(const std::vector<long>& v){
    long num_chiamate=0;
    auto iter=std::find_if(
        [&num_chiamate](const long&i) mutable{
            ++num_chiamate;
            return i%2==0;
         }
    );
}
```

Mettere mutable lì corrisponde a metterlo prima della variabile nella struct nome indefinito.

```
struct nome_univoco{
mutable long& num_chiamate;
}
```

tipo di ritorno

Nella funzione scritta sopra:

```
[](const long& i){return i%2==0}
```

lasciavamo fosse il compilatore a intuire il tipo di ritorno, ma se vogliamo possiamo esplicitarlo con

```
[](const long& i) bool{return i%2==0}
```

equivalenza

possiamo dire che l'espressione lambda corrisponde circa alle seguenti 3 operazioni:

I. definizione di una classe "anonima" per oggetti funzione, cioè una classe dotata di un nome univoco scelto dal sistema
II. definizione all'interno della classe di un metodo operator() che i parametri, il corpo e il tipo di ritorno specificati o dedotti dal compilatore
III. creazione di un oggetto funzione "anonimo" avente il tipo della classe anonima suddetta, da passare dove definita l'espressione lambda

riutilizzo

se volessimo riutilizzare più di 1 volta la stessa lambda expression dovremo creare un oggetto che contiene il risultato di essa, lo faremo scrivendo:

```
auto nome= tutta_espressione
```

deduzione automatica dei tipi

deduzione automatica dei tipi

Cosa succede quando facciamo la deduzione automatica dei tipi per i template di funzione, o anche quando usiamo la parola auto?
Immaginiamo il seguento codice:

```
template <typename TT>
void f(PT param);
```

il compilatore di fronte alla chiamata di funzione f(expr) usa il tipo (**te**) dell'espressione per dedurre:

- un tipo specifico **tt** per TT (per l'istanziazione del template)
- un tipo specifico **pt** per PT (definire la funzione)

Così facendo otterremmo : void f<tt>(pt param)

(in genere tt e pt saranno correlati ma **non uguali**)

il processo di deduzione dei parametri distingue 3 possibili casi:

1) PT è sintatticamente uguale a "TT&&" (riferimento ad un Rvalue di tipo TT), ovvero una "**universal reference**". in realtà potrebbe diventare un riferimento ad un Lvalue, quindi potremmo avere riferimento ad Rvalue o ad Lvalue
2) PT è un tipo puntatore o un riferimento (ma non una **universal reference**)
3) PT non è ne un puntatore ne un riferimento

analizziamo i casi:

universal reference (1)

Deve essere scritto per forza TT&& altrimenti non è una universal reference.
ad esempio:

- TT&& lo è
- const TT&& non lo è

- std::vector<TT>&& non lo è

Entrambe le precedenti ricadono invece nel secondo caso

//si assume int i=0; const int ci=1;
//Rvalue:

- f(s)
  - te: int
  - deduciamo pt: int&&
  - deduciamo tt: int
- f(std::move(i))
  - te: int&&
  - deduciamo pt: int&&
  - deduciamo tt: int

//Lvalue:

- f(i)
  - te: int&
  - deduciamo pt: int&
  - deduciamo tt: int&
- f(ci)
  - lte: const int&
  - deduciamo pt: const int&
  - deduciamo tt: const int&

Le stesse regole vengono applicate quando si utilizza “auto”

polimorfismo dinamico

Si va a contrapporre al polimorfismo statico

classi derivate e relazione “is-A”

Si consideri una classe base e una classe derivata pubblicamente dalla classe base

```
classe base{ ... }
classe derived:public base{ ... }
```

per derivare da una classe si utilizza l’operatore “:” ogni classe da cui si deriva dovrà avere prima una parola che ne indichi la visibilità per la classe derivata

di base:
- se stiamo derivando da una classe sarà private
- se stiamo derivando da una struttura sarà protected

possiamo quindi cambiare il comportamento base con la parola prima del nome della classe/struct
- se utilizziamo public: tutti i metodi e variabili della classe da cui deriviamo saranno public anche dentro di noi
- se utilizziamo protected saranno protected
- se utilizziamo private saranno private

derivazione pubblica

derivare pubblicamente da una classe vuol dire che l'utente potrà fare le conversioni di tipo up-cast

ovvero utilizzare un oggetto derivato come se fosse uno base
(e quindi eventuali modifiche di comportamento potremmo farle nella classe derivata)
quindi fare:

```
base* base_ptr=new derived;
```

In caso la derivazione fosse stata private/protected l'utente non avrebbe potuto fare gli upcast. li avremmo potuti fare solo internamente alle funzioni della classe derivata.

relazione has-A

La relazione Has-A a differenza della precedente non intende dire che la nostra classe è (deriva da) un'altra classe

ma che al suo interno contiene un elemento di un'altra classe.
Ad esempio logicamente, la classe macchina che al suo interno ha oggetto motore, oggetto sportello (non deriva da essi ma è formata dall'unione di essi).
Quindi per utilizzare le loro funzioni dovremo far capo all'oggetto che conteniamo.

Possiamo ottenere questo tipo di relazione in 2 metodi:
1) tramite derivazione private: come detto prima se derivassimo la classe in modalità private potremmo fare un upcast solo internamente alle funzioni della classe, quindi sarebbe possibile ottenere questo tipo di relazione
2) tramite contenimento: ovvero all'interno della nostra classe che contiene l'altra

classe andremo appunto a creare un oggetto dell'altra classe
Es:

```
class macchina{
    Motore motore;
}
```

override

Con la derivazione abbiamo un problema, immaginiamo le seguenti classi:

```
class Printer {
public:
  void print(const Doc& doc);
};

class FilePrinter : public Printer {
public:
  void print(const Doc& doc);
};

class NetworkPrinter : public Printer {
public:
  void print(const Doc& doc);
};
void stampa_tutti(const std::vector<Doc>& docs, Printer*
printer) {
  for (const auto& doc : docs)
    printer->print(doc);
}
```

In questo caso useremo tutte e tre le classi come se fossero delle normali printer, ignorando le possibili specializzazioni fatte nelle classi derivate
dobbiamo quindi trovare un metodo per andare a far capire al nostro compilatore che abbiamo modificato la funzione nella nostra classe derivata.

per fare ciò parliamo di override e di risoluzione dell'override.

virtual

Per far si che il compilatore capisca che potremmo aver modificato la funzione nelle classi derivate andiamo a marchiare la funzione della classe base con “**virtual**” esso dirà che i metodi possono essere virtuali

```
class Printer {
public:
  virtual void print(const Doc& doc);
};
```

quindi la dicitura virtual ci dice che la classe **potrebbe** essere stata riscritta e modificata nelle classi derivate da questa.

classi dinamiche

Se in una classe definiamo almeno un metodo con virtual allora quella classe sarà chiamata classe dinamica poiché supporta il polimorfismo dinamico

RTTI (Run time type identification)

se una classe è definita come dinamica il compilatore a tempo di esecuzione saprà rispondere alla domanda “a quale tipo punta il puntatore? punta a questa classe o ad una delle sue derivate?”

Es: Printer* printer attualmente punta a Printer, NetworkPrinter o FilePrinter?

override

Quindi andremo a fare il cosiddetto “**override**” nel momento in cui in una classe derivata andremo a scrivere un metodo che nella classe base era marchiato con **virtual.**
In questo caso vuol dire che stiamo “sovrascrivendo” la definizione del vecchio metodo, che se chiamato nel tipo derivato farà il comportamento differente

importante

Per effettivamente fare l’override di un metodo non basta che abbia lo stesso nome di quello usato nella classe base, ma deve essere analogo, ovvero

- stesso tipo di ritorno
- stesso numero di parametri (almeno che non siano definiti con valori di default)
- stesso tipo dei parametri (non vanno i cast impliciti, devono essere perfettamente uguali) altrimenti si fa hiding

parola override

Per evitare errori possiamo usare la dicitura “**override**” dopo la firma dei metodi nella classe derivata, essa ci permette di capire se stiamo sbagliando qualcosa.
Infatti nel caso in cui c’è un errore ce lo notificherà
Esempi di errore

- il metodo descritto non era virtuale nella classe base, quindi in realtà stiamo facendo hiding (ovvero semplicemente creiamo un metodo con lo stesso nome che va a rendere inutilizzabile il precedente) non override
- abbiamo modificato un po’ il metodo descritto, non si sta quindi facendo override, ma stiamo aggiungendo un metodo totalmente diverso oppure andando a nascondere il vecchio metodo

metodi puri

Si definisce un metodo come puro quando esso offre solo l’interfaccia del concetto ma non la sua implementazione (che verrà fornita nelle classi derivate)

sintassi

Per definire un metodo come puro andremo ad aggiungere un =0 dopo la sua firma

```
class printer{
public:
virtual std::string name() const=0;
virtual void print(const Doc& doc)=0;
}
```

Ovviamente i metodi per essere puri dovranno essere virtuali (dato che va fatto l’override nelle classi derivate)

classi astratte

Una classe viene definita come astratta quando al suo interno c’è almeno un metodo virtuale puro.

Se una classe derivata da una classe astratta non fa l’override del metodo puro anche essa sarà una classe astratta, altrimenti, se lo fa, potremo definirla concreta.

Se abbiamo una classe astratta non possiamo definire oggetti di tale classe, in quel caso il compilatore darebbe errore

Printer p; //errore
NetworkPrinter p; //ok, se la classe fa l'override di metodi puri

distruttore

All'interno delle classi astratte dobbiamo **obbligatoriamente** definire **un distruttore non puro (anche se senza istruzioni)**, questo perché quando sarà distrutto l'oggetto dovremo avviare il distruttore della classe, e di conseguenza anche di quella madre così da fare una distruzione totale.

Dovrà però essere marcato virtual perché in caso contrario potremmo avere dei memory leaks.
Se la nostra **classe derivata creasse dei dati** durante l'esecuzione di un suo metodo eseguito durante un override, quindi in generale stiamo **trattando la classe come una classe base**, avremmo che **senza l'override** del distruttore **non distruggeremmo quei dati creati nella classe derivata**, giungendo quindi ad un **memory leaks**

osservazione

il distruttore sarà l'unico metodo che nonostante abbia un nome diverso nella classe derivata rispetto a quella base (dato che ovviamente deve essere ~nome_classe())

risoluzione dell'override

La risoluzione dell'override consiste nel capire se dobbiamo eseguire la funzione della classe base o della derivata.
Precisiamo che prima di capire questo dobbiamo comunque capire quale funzione delle classe base eseguire facendo la risoluzione dell'overloading dei suoi metodi.

condizioni di attivazione

per attivare l'override dobbiamo avere diverse condizioni:

- Il metodo chiamato sia virtuale (esplicitamente o implicitamente, ovvero se ereditiamo la virtualità da una classe base)
- il metodo deve essere invocato tramite puntatore o riferimento, se lo si fa con copia non si riesce a differenziare, quindi si invocherebbe il tipo della classe base
    - **in aggiunta direi che deve esserci un mismatch, ovvero che il tipo del puntatore è differente dal tipo dell'oggetto puntato.**
      Quindi abbiamo un puntatore della classe base che punta ad un

oggetto della classe derivata

- nella catena di derivazione delle classi ereditate almeno una classe ha effettuato l'override, in mancanza di ciò si invoca il metodo della classe base
- il metodo non deve essere invocato mediante qualificazione esplicita (in quel caso semplicemente si invocherebbe il metodo della classe indicata senza fare l'override), ovvero non inserendo l'operatore di scope.
  Es:
  printer→ Printer::print() (così facendo non andremo ad eseguire l'override perchè sapremmo a quale metodo ci riferiamo)

esercizi

gli esercizi di questa parte sono nella lezione 18
esercizio 1 lez 18
esercizio 2 lez 18

consigli

- é preferibile creare delle interfacce base, così da evitare di modificare il codice iniziale creando possibili problemi. e di proseguire nelle modifiche del codice base attraverso delle classi derivate dove si fa l'override dei metodi ecc

conversioni esplicite di tipo

Esistono 6 categorie possibili conversioni che l'utente può richiedere esplicitamente per una certa variabile:

1) static_cast
2) dynamic_cast
3) const_cast
4) reinterpret_cast
5) cast funzionale
6) cast stile c

motivi per fare cast esplicito

Di base abbiamo visto che il linguaggio implicitamente fa già dei cast, quindi perché l'utente dovrebbe voler fare esplicitamente i cast?

Ci sono diversi motivi:

A. Il cast implementa una conversione di tipo non consentita dalle regole del linguaggio come conversione implicita poiché potrebbe scaturire in errori. Il programmatore quindi potrebbe richiederla esplicitamente assumendosi eventuali errori

Esempio

```
Struct B{}
struct D:public B{}
D d;

B* b_ptr=&d
/*
… altre istruzioni…
*/
D* d_ptr=static_cast<D*>(b_ptr)
```

Nell'ultima istruzione il programmatore di assume la responsabilità di ciò che fa, immaginando infatti che b_ptr non puntasse più ad un oggetto di tipo d, nell'ultima istruzione staremmo provando a leggere un tipo totalmente differente come D*, ottenendo ovviamente un errore

B. Abbiamo una situazione analoga a prima, però questa volta andiamo ad usare un dynamic_cast per essere sicuri che quello che stiamo facendo non crei errori

Esempio

```
Struct B{}
struct D:public B{}
D d;

B* b_ptr=&d
/*
… altre istruzioni…
*/
if(D* d_ptr=dynamic_cast<D*>(b_ptr)){
/*qui dentro abbiamo la sicurezza che il cast è potuto
avvenire e quindi effettivamente D° d_ptr puntava ad un
tipo compatibile con D */
}else{
```

```
/*qui invece siamo nel caso in cui il cast non è avvenuto
e quindi dobbiamo procedere in modo diverso*/
}
```

C. Il cast esplicito non è necessario in quanto il cast implicito è consentito dal linguaggio, il programmatore però scrive ugualmente il cast esplicito per 2 motivi
   a. non c'è possibilità che venga trasformato in un altro tipo o che il cast non si capisca
   b. quando si rilegge il codice si vede esplicitamente quel cast ed è quindi più semplice capire cosa succede

Esempio

```
double b=...;
int i=static_cast<int>(b);
//il cast sarebbe stato fatto ugualmente implicitamente ma
così il programmatore lo esplicita rendendolo più visibile
```

D. Potrebbe accadere in certe occasioni che abbiamo dei valori utilizzati solo per motivi di debug, quindi in asserzioni ecc..
   In questo caso se compilassimo con le opzioni apposite per ignorare le asserzioni il compilatore darebbe dei warning per evidenziare che quel parametro (che prima usavamo nelle asserzioni) non viene usato da nessuna parte.
   Per evitare questi warning possiamo fare dei cast di quel valore a void, questo sarebbe un po' come dire al compilatore di ignorare il dato.

Esempio

```
void foo(int pos, int size){
  assert(0<=pos && pos<size);
  static_cast<void>(size);
/*così anche se usassimo ndebug(non calcolando le
asserzioni) e quindi se size non venisse utilizzato da
nessuna parte, eviteremmo di ricevere errori facendo
questo cast a void*/
}
```

   a. é anche l'unico caso in cui è lecito usare una conversione stile c, ovvero (void) size, negli altri casi è considerato cattivo stile

diversi tipi di cast

Ad inizio capitolo abbiamo indicato i diversi tipi di cast espliciti presenti nel linguaggio. andiamo a vedere come funzionano e le loro caratteristiche

static_cast (1)

tutto ciò che c'è da sapere per il cast si conosce a tempo di compilazione (detto appunto statico) (tipo di partenza e tipo di arrivo, quindi cosa fare per la conversione)

sintassi

```
static_cast<tipo_arrivo>(expr da convertire)
static_cast<int>(bool) //si passa da boolean ad int
```

Si va a calcolare un nuovo valore, convertendo il valore ottenuto dall'espressione al tipo segnato tra parentesi angolate.

legittimità

Questo cast è legittimo nei casi:

- è legittima la corrispondente conversione implicita

```
double d=3.14;
int i=static_cast<int>(d);
```

- è legittima la costruzione diretta di un oggetto di tipo T (intendo il tipo tra parentesi angolate) passando l'espressione come argomento

```
razionale r=static_cast<razionale>(5)
//era legittimo fare anche
razionale r=razionale(5)
```

- si effettua la conversione inversa rispetto ad una sequenza di conversioni implicite ammissibili (con alcune limitazioni, ad esempio non possiamo invertire le trasformazioni di Lvalue)

```
int i=42;
void* v_ptr=&i;
//con void* si considera un puntatore ad un dato di cui non si conosce tipo
int* i_ptr=static_cast<int*>(v_ptr)
//come nel caso della casistica A) per fare i cast, se v_ptr non puntava effettivamente ad un int avremmo avuto un undefined behavior
```

- il cast implementa un downcast di una gerarchia di classi

- il cast implementa un cast da un tipo numerico ad un tipo enumerazione (poiché per conversione implicita non sarebbe permesso questo cast)
- il tipo di destinazione è void

dynamic_cast(2)

è uno degli operatori che forniscono il supporto al poliformismo dinamico, quindi per la RTTI(Run Time Type Identification)

Possono essere usati per conversioni all'interno di una gerarchia di classi, legate da ereditarietà.

tipologie

- up-cast: conversione da classe derivata a classe base, raramente fatte con un dynamic_cast perché già possibili come conversioni implicite, quindi non necessitano della RTTI
- down-cast: conversione da classe base a classe derivata.
  è l'uso più comune dei dynamic_cast, si usa la RTTI per capire se la conversione è legittima, ovvero se il puntatore al tipo di partenza puntava appunto ad un tipo della classe derivata che indichiamo noi (vedere esempio B)
- mixed-cast: ovvero quando abbiamo che una classe eredita da più classi, quindi possiamo spostarci tra classi differenti combinando up-cast e down-cast

possiamo applicare il dynamic_cast a:

- puntatori: il caso più frequente
- riferimento: caso poco frequente
- non ad oggetti normali: infatti per esserci un dynamic_cast deve esserci un mismatch, ovvero che il tipo del puntatore e il tipo indicato da esso siano differenti (ciò si ottiene quando si passano oggetti di una classe derivata a puntatori di una classe base)

esempi

```
struct B{}
struct D1:public B{}
struct D2:public B{}
void foo(B* b_ptr){}

D1* d1_ptr=dynamic_cast<D1*>(b_ptr)
```

//se la classe B è dinamica (ovvero contiene almeno un metodo virtuale) allora è fornita delle informazioni per la RTTI, quindi possiamo applicare cast dinamici ai puntatori per

sapere se essi sono di un determinato tipo

dopo questa esecuzione avremo 2 possibili casi:

- b_ptr punta ad un oggetto di tipo D(anche ad un suo derivato) allora il cast va a buon fine e d1_ptr ottiene un indirizzo di memoria a cui puntare
- b_ptr non punta ad un oggetto D1 (o suoi derivati) ad esempio punta a D2, quindi d1_ptr viene inizializzato con NULL

Possiamo controllare quindi che il cast sia andando a buon fine con un if

```
if(d1_ptr!=nullptr){}
else{}
//oppure in modo equivalente (sfruttando conversione a bool)
if(d1_ptr){}
else{}
```

Oppure come visto nell'esempio di cast B) possiamo inizializzarlo in un if

```
if(D1* d1_ptr=dynamic_cast<D1*>(b_ptr)){
    //qui dentro d1_ptr non è null
}else{
//qui è null
}
```

info

- Nel caso in cui utilizzassimo piuttosto che puntatori dei riferimenti avremmo dei problemi, infatti per essi non esiste un valore analogo a null_ptr, quindi quando il riferimento ritorna un valore "nullo" al suo posto viene lanciata una eccezione, che ovviamente dovrà essere gestita complicando il tutto.

- **Possiamo usare il dynamic cast solo su classi dinamiche, ovvero che posseggono almeno un metodo virtuale, sennò non potremmo risolvere la RTTI**

Const_cast(3)

Si usa per togliere la qualificazione const a qualsiasi cosa fosse const, solitamente applicata a puntatori/riferimenti a oggetti const (quindi non modificabili) ottenendo oggetti modificabili.

problema

Così facendo però andiamo contro una sorta di promessa fatta, ovvero dichiarando un qualcosa const stiamo “promettendo” di non modificarne mai il valore, ma usando questo stratagemma potremmo.

```
void promessa(const int& ci){
int &i=const_cast<int&>(ci);
++i;
}
```

allora perchè è permesso dal linguaggio?
Perché viene usato quando si vuole modificare la rappresentazione interna di qualcosa senza modificarne il significato.

Esempio
Vogliamo ordinare una certa lista passata con const, se l’ordinamento viene richiesto tante volte potremmo decidere piuttosto di prendere la lista e ordinarla direttamente su quella variabile stessa **senza però modificarne i valori**, così da evitare di riordinare ogni volta sprecando prestazioni

Una tecnica analoga a questa è usare il modificatore **mutable**

reinterpret_cast(4)

guarda una sequenza di bit codificata in un modo e la interpreta in un altro modo.

- da puntatore ad intero (quindi memorizziamo l’indirizzo in un intero abbastanza grande da poterlo rappresentare)
- da un tipo intero/enumerable ad un tipo puntatore
- da un tipo puntatore/riferimento ad un altro puntatore/riferimento
  (qui si intende quando i due tipi non sono legati, ad esempio da puntatore ad int a puntatore a double)

Non possiamo usarlo per rimuovere il const, per quello dobbiamo usare per forza il const cast

cast funzionale(5)

per il cast funzionale si usa la sintassi “tipo(espressione)” ed equivale a creare un oggetto del tipo “tipo” utilizzando il suo costruttore che accetta “espressione” come parametro.

Anche un tipo() è un cast funzionale che equivale ad usare il costruttore di default della classe.

cast stile c(6)

sintassi

(tipo) expr
Es: (int) 5.4

Sono considerati cattivo stile, ovvero:

- non si capisce che tipo di cast stanno applicando (static, interpret, const), e possono applicarne diversi
    - non possono applicare il dynamic_cast perché esso prevede delle informazioni per la RTTI
- non sono facili da individuare nel testo graficamente

non sono considerati cattivo stile solo nel caso mostrato in D), ovvero quando andiamo a fare un cast a void dei parametri che usiamo nelle asserzioni per evitare che appaiano dei warning

principi di progettazione object oriented

Questi principi servono per fornire una sorta di guida per sviluppare progetti più flessibili, ovvero per i quali sarà più facile effettuare modifiche in un secondo momento, implementando:

- modifiche a funzionalità esistenti
- nuove funzionalità

(Per tale motivo è inutile fare questi sforzi per codice “usa e getta”, risulterebbe costoso inutilmente, si utilizzano quindi per tutti quei codici longevi per i quali ha senso “sprecare tempo” per ottimizzare il tutto per le varie esecuzioni)

solid

I principi che studieremo sono chiamati **solid:**

- **S (srp) (Single Responsibility Principle)**
- **O (ocp) (Open-Closed Principle)**
- **L (lsp) (Liskov Substitution Principle)**
- **I (isp) (Interface Segregation Principle)**
- **D (dip) (Dependency Inversion Principle)**

(non sono stati proposti tutti insieme e in quest'ordine, sono solo stati raggruppati in questa sequenza in modo da poter formare l'acrostico SOLID che fa pensare a "sicuro" "affidabile")

SRP(Single Responsibility Principle)

Questo principio non si usa solo per la programmazione object oriented, dice sostanzialmente che quando progetti una qualsiasi cosa (classe, struttura, funzione ecc...) essa si deve occupare di un solo compito.

Questo sotto un'altra luce può anche voler dire che abbiamo un solo motivo per modificare la classe.
Se si occupa di molte cose è meglio dividere il codice in "parti" da poi comporre in una unica
(Ad esempio la funzione che si occupa solo di far la somma) (se abbiamo una funzione che deve fare la somma e dividere è meglio che implementi implementi 3 funzioni, una che faccia la somma, l'altra che faccia la divisione e la 3° che unisca le 2 funzioni)

OCP(Open-Closed Principle)

é forse il più conosciuto, nel tempo ne sono state proposte 2 varianti
la seconda è quella che interessa i principi solid, essa dice:
Il software **dovrebbe** essere:

- chiuso alle modifiche
- aperto alle estensioni

Per fare questo idealmente andiamo a dire che quando progettiamo un software dobbiamo **progettare le funzionalità base**, e quindi quando dovremmo aggiungere funzionalità ad esso dobbiamo strutturare una classe derivata in modo tale che **attraverso l'override potremo aggiungere funzionalità (o modificare le vecchie) senza però modificare mai effettivamente il software base**

visione

Possiamo andare a guardare i due prossimi principi, DIP e LSP come 2 modi specifici di guardare al principio OCP

- DIP: concentrandoci sulla sintassi del nostro codice
- LSP: concentrandoci sugli aspetti semantici del nostro codice

DIP(Dependency Inversion Principle)

Le entità software possono dividersi in 2 categorie, moduli di software di:

- alto livello: si occupano dei problemi in modo generale (astratto)
- basso livello: si occupano dei problemi in modo più concreto e quindi vicino al problema

Il DIP dice che i moduli ad alto livello non devono dipendere da quelli di basso livello, ed entrambi devono dipendere dalle astrazioni (che sono i moduli di più alto livello) quindi non deve essere che le astrazioni dipendono dai dettagli ma il contrario

(ad esempio prima strutturiamo una classe chiamata blocco_note, poi magari strutturiamo una classe chiamata note_righe dipendente da blocco note, che non fa altro che aggiungere un dettaglio in più alla classe iniziale) (quindi in questo caso la classe blocco_note è quella astratta, la classe note_righe è quella più dettagliata)

Classificazione

Scrivendo del codice ci troviamo quasi sempre a dover scrivere delle effettive dipendenze tra due entità, il principio va a classificare queste dipendenze dicendo quali di queste sono ammesse (anche perché indispensabili) e quali invece andrebbero evitate (perché dannose)

- le dipendenze buone sono quelle in cui una classe più concreta dipende da una più astratta
- le dipendenze da evitare sono quelle in cui un qualcosa di più astratto dipende da qualcosa di più concreto

Soluzione

Nel nome del principio parliamo di inversione perché **nella classica programmazione** andiamo ad **utilizzare** un **approccio top down**, ovvero partiamo da un qualcosa a cui servono delle funzioni, e man mano scendiamo a strutturare quelle funzioni da utilizzare, in breve stiamo **applicando delle dipendenze cattive (il progetto iniziale è quello più astratto che dipende da implementazioni di cose più concrete e vicine al problema)**

Quello che vogliamo ottenere è di invertire queste dipendenze, appunto attraverso l'ausilio dell'astrazione delle classi

Esercizio

Manual_generator lez 19

LSP(Liskov Substitution Principle)

In questo principio indichiamo quando un dato può essere "sostituito" ad un altro,

immaginiamo quindi la seguente situazione
Abbiamo 2 tipi di dato:

- T
- S: sottotipo di T

Possiamo dire che effettivamente S sarà sottotipo di T solamente nel caso in cui ogni qual volta un modulo usa un oggetto di tipo T possiamo fornire al suo posto un oggetto di tipo S ottenendo un risultato equivalente (ovvero che soddisfi le aspettative dell'utente)

Rivisitazione: possiamo anche vederlo come
Ogni funzione che usa puntatori o riferimenti a classi base deve essere in grado di usare oggetti delle classi derivate senza saperlo

behavior subtyping

Definiamo il **behavior subtyping** come:
Le sottoclassi S devono rispettare le aspettative degli utenti che accedono ad esse tramite puntatori a tipo T, ovvero le sottoclassi S devono non solo fornire metodi sintatticamente forniti da T ma che abbiano comportamento analogo ad essi (aspetto semantico).

Fare le cose in **modo analogo** vuol dire che rispettano le condizioni date "per contratto", **ovvero che rispettino precondizioni e postcondizioni**

Esempi

generale

Un esempio di rispetto di contratto è dato nella fattoria dove noi richiediamo che quando viene ritornato il "nome" dell'animale si parli di nome comune dell'animale non del proprio, quindi non "rocky" ma "cane".

concreta

Esempio in cui non si rispetta LSP
abbiamo una classe rettangolo che ha come invariante (lunghezza>0 larghezza>0), e ha delle funzioni

- set_larghezza
- set_lunghezza
- get_area(): essa ritorna (lunghezza*larghezza)

Se volessimo ora derivare da questa classe la nostra classe quadrato da quella

rettangolo semplicemente diremmo che il quadrato è un rettangolo che ha lunghezza e larghezza uguali
Cosa succede però, che abbiamo una funzione chiamata
“raddoppia area” che riferendosi alla classe rettangolo non fa altro che calcolare l’area dicendo che devo moltiplicare la lunghezza del rettangolo*2 (così da ottenere area doppia).
Alla fine dovremo avere che raddoppia area ritorna una area=area(vecchia)*2
Se passiamo a questa funzione un oggetto quadrato vediamo che andando a fare lunghezza*2 in realtà alla fine ritorneremo un’area che è **area=area(vecchia)*4**
(dato che andando a modificare la lunghezza per come è fatto il quadrato modificheremo anche la larghezza)

Esercizio vero e proprio sul drive
Quadrato rettangolo lezione 19

concessioni
Nelle classi derivate abbiamo alcune concessioni, ovvero cose che possiamo fare al loro interno senza andare contro questo principio, esse sono:

- indebolire le pre condizioni (se la classe astratta volesse che fossero rispettate le condizioni A B e C, ma a noi per esempio serve che sia rispettata solo A per arrivare alle stesse postcondizioni possiamo tranquillamente non richiedere le condizioni B e C)
- rafforzare le post condizioni (se la classe astratta promette di ritornare un dato che abbia le condizioni A e B noi come classe più concreta possiamo promettere di ritornare un dato che rispetti le condizioni A B e C)

Il polimorfismo dinamico è in poche classi delle librerie standard, in genere esse usano lo statico, una delle poche che usa il dinamico è la classe delle eccezioni standard (exception) con conseguentemente tutte le sue derivate

ISP (Integrate segregation principle)
l’utente non dovrebbe essere forzato a dipendenze da parti di interfacce che non usa, il progettista dell’interfaccia dovrebbe separare quelle porzioni che possono essere usate separatamente l’una dalle altre.

Si preferiscono quindi tante interfacce ma piccole rispetto a poche ma grandi.
Aderendo a tale principio abbiamo i seguenti benefici:

- dato che abbiamo tante piccole interfacce l’implementatore può decidere di

implementare solo alcune (quelle che servono effettivamente)
- se facciamo un errore in una interfaccia separata, tale errore non si propaga sulle altre
- dato questo obiettivo di far usare le interfacce solo a chi serve in teoria così facendo le varie interfacce saranno usate da meno persone rispetto a prima, e quindi quando aggiorniamo l'interfaccia sono meno le persone coinvolte a dover aggiornare i programmi di conseguenza

Per questo dobbiamo poter usare l'ereditarietà multipla
Possiamo vedere questo principio come una forma particolare del SRP(Single Responsability Principle) il quale diceva che ogni interfaccia deve svolgere un solo compito

ereditarietà multipla

quando andiamo a derivare le classi ricordiamoci di dover inserire l'include del file contenente la classe all'inizio

Possiamo usare l'ereditarietà multipla semplicemente inserendo diverse classi da cui derivare dopo i : tutte divise da , e ognuna di essa con un proprio indicatore di visibilità

class C: public A, public B{ … }

adattatori

Possiamo definire gli adattatori come delle funzioni che conoscono la classe che vogliamo utilizzare e ci permettono di chiamare le sue funzioni in modo più semplice. A volte ci permettono anche di migliorare le funzioni offerte o offrono funzioni che sono "l'unione" di alcune funzioni della classe che vogliamo usare

Possiamo quindi vederlo come una sorta di driver

Questioni tecniche polimorfismo dinamico

Possibilità di metodi virtuali

Ci sono alcuni metodi che non possono essere virtuali?
Analizziamo le seguenti funzioni

- **costruttori:** NO
  - non possono esserlo perché stiamo creando l'oggetto base, esso va obbligatoriamente creato per creare anche l'oggetto derivato
- **distruttori:** SI(obbligatoriamente)
  - Come avevamo visto tempo fa deve essere virtuale perché se utilizziamo la classe concreta attraverso puntatori alla classe astratta se il distruttore non fosse virtuale non andremmo a distruggere eventuali pezzi di memoria allocati dalla classe derivata
- **funzioni membro (di istanza, non statiche):** SI (non obbligatorio)
  - si, si intendono le classiche funzioni che abbiamo analizzato fino ad ora
- **funzioni membro statiche:** NO
  - si differenziano dalle normali per il fatto che non fanno capo alle singole istanze ma alla classe direttamente, quindi non si usano attraverso l'operatore this dell'oggetto.
    Questo vuol dire che non possono essere virtuali per il fatto che quando dovremmo capire il RTTI non possiamo interrogare l'oggetto per capire di quale classe effettivamente sia
- **Template di funzione membro (non statiche):** NO
  - Si intendono classi in cui è presente al loro interno un template di funzione. ES:

```
Class prova{
    template <typename T>
    virtual T funzione(T a, T b){ ... }
}
```

Teoricamente potremmo farle virtuali però in realtà viene disabilitato dal linguaggio poiché sarebbero difficili da gestire, dato che per ogni istanza avremmo che possiamo definire la funzione templatica in vari modi (cioè passando ogni volta tipo diversi al template) e questo vorrebbe dire modificare di volta in volta la tabella che racchiude i metodi virtuali della classe.
Ciò da come si può presumere è difficile soprattutto se si vuole essere efficienti

- **Funzioni membro di un template di classe (non statiche):** SI (non obbligatorio)
    - Si intende un template di classe con all'interno delle funzioni
      ES:

```
template <typename T>
class animale{
public:
virtual void foo(T t){ ... }
}
```

In questo caso **non abbiamo il problema precedente** perché una volta parametrizzato il template di classe (e quindi creando una specifica istanza di quel template di classe) **i metodi virtuali rimangono comunque sempre gli stessi, non c'è necessita di modificare la tabella dei metodi virtuali**, e quindi si fa in modo analogo a quanto fatto per le classiche funzioni delle classiche classi

Copia di un oggetto concreto tramite puntatore/riferimento a classe base

```
//Immaginiamo di avere una classe
Class base{ ... }
Class derivata: public base{ ... }
/*e di voler fare una copia in una funzione che accetta un
puntatore al tipo base passando però un oggetto di tipo
derivata*/

derivata prova;
foo(base * ba){
//crea copia di ba
}
foo(&prova);
```

Come dobbiamo implementare quella funzione di copia?

facendo:

`foo(base * ba){`
`base* ba2=ba;`
`}`
	no perché ba2 non sarebbe altro che un altro puntatore di tipo base che punta all'oggetto di tipo derivata

`foo(base * ba){`
`base ba2=*ba;`
`}`
	No perché faremmo uno slicing, ovvero andremmo a copiare solo la parte base della classe, non la parte potenzialmente modificata dall'implementazione di derivata
	
	Oltretutto se la classe base avesse metodi puri otterremmo errore perché non potremmo creare oggetti

Come si deve fare?

Ogni classe avrà un metodo "clone" che sarà virtuale così da essere implementato anche nella derivata, sostanzialmente andremo a creare un nuovo spazio in memoria dinamica inizializzato (tramite costruttore di copia) dal nostro this e ritorneremo il puntatore di tipo derivata che punta allo spazio ottenuto nella memoria dinamica

costruttore virtuale

viene anche chiamato impropriamente così il metodo clone

```
class base{
virtual base* clone()=0;
}
```

```
class derivata:public base{
  derivata* clone(){
    return new
derivata(*this);
    /*istanziazione di un
blocco in memoria virtuale con
dentro il la copia del nostro
oggetto passato*/
  }
}
```

Nella classe derivata avremo una funzione clone che ritorna un puntatore al tipo derivata questo si fa perché se volessimo fare la copia appunto di un oggetto derivata, quindi:

```
foo(base * ba){
```

```
derivata* ba2= ba->clone;
}
```

Andremmo a chiedere un downcast che non è possibile, quindi in ritorniamo un oggetto del tipo in questione.

Questo è consentito dal principio LSP perché andando a cambiare il tipo di ritorno della funzione clone (dalla classe base alla classe derivata) non facciamo altro che andare a rafforzare le post condizioni, quindi è tutto consentito

Invocazione di un metodo virtual durante distruzione/costruzione

Immaginiamo tre classi:

- A
- B: deriva da A
- C: deriva da B

e immaginiamo che A abbia il metodo m virtual che non fa altro che stampare il dato della classe.

costruttore

immaginiamo quindi di invocare m nel costruttore di ognuno.
inizializzando un oggetto di tipo C avremo che:

- vogliamo creare l'oggetto C per dato che deriva da una classe prima di tutto il suo costruttore andrà a richiamare quello della classe base B
- viene richiamato il costruttore di B che analogamente chiama quello di A
- Siamo all'interno del costruttore di A che non deriva da nessuna classe, quindi esegue effettivamente il suo costruttore.

Seguendo il flusso spiegato fino ad ora diremo che adesso il costruttore di A chiamerebbe m, il quale però essendo virtual andrebbe a fare la RTTI e ritrovarsi che il tipo concreto è A ma il tipo chiamante è C, quindi andrebbe a fare l'override della funzione chiamando la m di C. Questo sarebbe un problema perché C, come detto prima, ancora non sarebbe stato inizializzato (dato che prima crea la parte di A poi quella di B e poi finisce di creare se stesso, ancora siamo alla prima parte)

Nelle versioni più recenti di C++ quindi è stato cambiato questo comportamento e quando si invocano metodi virtuali nel costruttore/distruttore avremo che il *this punterà

al tipo di dato attuale che si sta costruendo.

- In questo caso arrivati al costruttore di A il this punterà appunto ad A, e quindi m stamperà i dati di A terminando di seguito la costruzione della parte di A
- finisce il costruttore di A e passiamo quindi all'esecuzione di quello di B, il quale in modo analogo chiamerà la funzione m con il puntatore che punta al tipo B, verrà eseguito e terminerà il costruttore di B
- verrà quindi eseguito anche quello di C

distruttore

Analogamente per il distruttore succederà l'inverso, ovvero distruggeremo prima lo spazio di memoria di solo C, poi lo spazio di memoria di solo B e poi lo spazio di memoria di solo A (ovvero l'ultima parte), e in ognuno di essi il *this punterebbe al tipo di dato in cui si trova attualmente a livello logico, quindi non ci sarebbero problemi

ereditarietà multipla

Come si gestisce l'ereditarietà multipla quando non ci si limita ad interfacce astratte?

scope e ambiguità

Immaginiamo di avere 3 classi:

- B1 e B2: 2 classi separate
  ognuna di esse ha
  - metodo di nome m
  - variabile di nome a
- D: classe derivante sia da B1 che da B2

Se provassimo quindi a chiamare
D.m() o D.a avremmo errore perché non sarebbe possibile capire a quale "m" o "a" ci si sta riferendo (se a quello della classe B1 o a quello di B2)

(in realtà per quanto riguarda m potrebbe essere risolta l'ambiguità tramite l'overloading di funzione, però ovviamente non va sempre bene, ad esempio se le due m fossero uguali)

soluzione

Per risolvere possiamo agire in 2 modi:

- usare l'operatore di scope definendo quale "m" o "a" si sta chiamando
  d.B1::m();
  d.B2::a;
- all'interno della classe d si implementa un metodo con ugual nome che decide quale metodo chiamare effettivamente

derivazione ripetuta

Di base i compilatori non permettono di derivare 2 o più volte dalla stessa classe

```
class D: public B, public B{ ... } //non è concesso
```

per farlo dobbiamo utilizzare uno stratagemma, ovvero creare delle classi intermedie che semplicemente derivano dalla Base senza però modificare nulla, e poi far derivare la nostra classe derivata da queste classi intermedie

```
class B1:public B { }
class B2:public B { }
class B3:public B { }
class D: public B1, public B2,public B3{ ... }
```

ereditarietà virtuale

Se volessimo che si derivi più volte dalla stessa classe ma non moltiplicando le componenti (cioè immaginando che B avesse metodo m() non vogliamo che derivando 3 volte da B (attraverso le classi intermedie) abbiamo tre volte m()) ma facendo in modo che ci sia condivisione di componenti (metodi e variabili con conseguenti valori di esse)

Per farlo si utilizza la **derivazione virtuale**
si va quindi ad inserire virtual quando si deriva da una classe

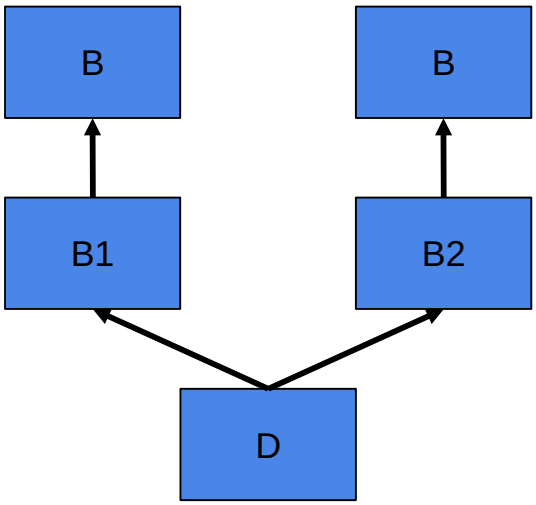
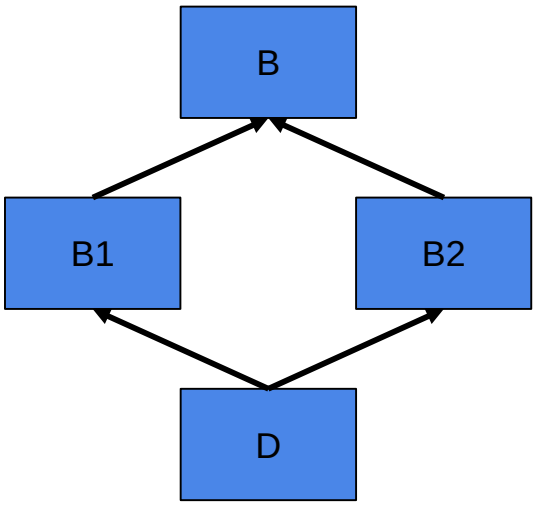
```
class B1 : virtual public B{}
```

Se saranno presenti altre classi che derivano in modo virtuale dalla classe base allora tutte loro condivideranno le componenti

(torna utile a volte quando si eredita dagli stream per poter scrivere e leggere e avere tutto condiviso per alcune operazioni)

problema

Con l'ereditarietà virtuale però ci si trova in una situazione in cui abbiamo una forma a diamante

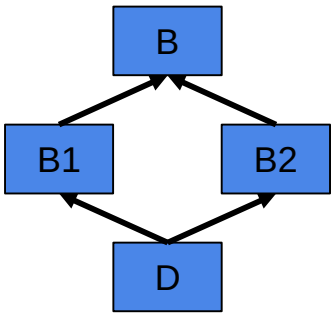
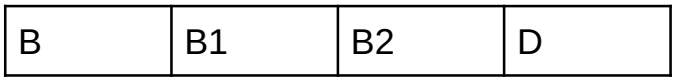
Non così	Così
	

Questa situazione viene chiamata DDD (Deathly Diamond of Death)
ovvero indica che spesso da questa situazione ne derivano vari errori

semantica speciale di inizializzazione

Quando usiamo l'ereditarietà virtuale modificheremo un po' la costruzione e distruzione dei nostri oggetti

immaginiamo

struttura vista prima	Allocazione in memoria
	

Andando ad eseguire normalmente avremo che:

- D per costruirsi costruisce B2
  - B2 per costruirsi costruisce B
- D per costruirsi costruisce B1
  - B1 per costruirsi costruisce B
    Essendo che B è virtuale e dovrebbe essere condivisa **sarà già stata costruita al passo precedente quindi avremo un memory leak**

soluzione

Vedremo le soluzioni per la costruzione e distruzione dato che funzionano in modo analogo ma inverso

costruzione

per la costruzione proseguiremo nel seguente modo

- prima si inizializzano le classi base virtuali
- poi le restanti classi base non virtuali

eseguendo avremo quindi che

1) vediamo se ci sono classi base virtuali
   a) ce n'è una ovvero B0, costruiamo quella
2) poi proseguiamo costruendo le classi non virtuali
   a) costruiamo B1
   b) costruiamo B2
   c) costruiamo D

distruzione

Analogamente per la distruzione dato che anche qui distruggeremo in ordine inverso di costruzione avremo che:

- prima distruggeremo tutte le classi Derivate e Base non virtuali
- poi per ultime distruggeremo le classi base virtuali

1) distruggiamo la parte D
2) distruggiamo quindi B2
3) distruggiamo B1
4) distruggiamo B

esercizi

file Panda.cc lezione 21

file search.cc lezione 22
file aggregatore.cc lezione 22

differenza tra polimorfismo statico e dinamico

- POLIMORFISMO STATICO: basato sui template
- POLIMORFISMO DINAMICO: basato sulle classi dinamiche tramite l'ausilio di metodi virtual

le differenze sono:

- il polimorfismo statico prevede che scegliamo a tempo di compilazione come istanziare i template, in quello dinamico alcune info si sapranno solo a tempo di compilazione
- nel polimorfismo statico si accettano come tipi templatici tutti quelli che richiediamo, in quello dinamico invece dobbiamo avere una relazione di ereditarietà da un determinato tipo per essere accettato
- nel polimorfismo statico se abbiamo una classe templatica le varie funzioni venivano istanziate "on demand", cioè solo se utilizzate (dato che derivano dal tipo usato nel template), in quello dinamico invece dobbiamo implementarle tutte

preprocessing condizionale

Abbiamo visto durante il corso che alcune funzionalità del linguaggio come "auto" o "override" vengono fornite solo da certe versioni del linguaggio in poi

Come possiamo fare quindi ad eseguire queste funzioni se non sappiamo se stiamo utilizzando versioni di linguaggio vecchie oppure nuove?

dobbiamo usare le impostazioni di preprocessing condizionale, ovvero, facciamo l'esempio con override

```
#if defined (_msc_ver) && _msc_ver>=1800
#defined use_override override
#else
#define use_override
#end_if
```

andiamo a controllare se la versione è definita e se è maggiore di 1800 (idealmente si parla di versione del linguaggio e con 1800 possibilmente intendiamo la versione in cui veniva introdotto override), e diciamo se è maggiore di 1800 diciamo che “use_override” verrà cambiato a tempo di compilazione con “override”, altrimenti se non è così verrà cambiato a tempo di compilazione con “ “ (nulla, quindi sostanzialmente sarà come se non l’avremo messo)

una volta fatto questo semplicemente definiamo la nostra classe e dove dovremo mettere override inseriamo invece use_override

```
class derivata::public base{
void print() use_override { ... }
//use override verrà controllata dinamicamente e in base alla
versione seguirà il processo scritto sopra
}
```

divisione operazioni di inizializzazione, allocazione, costruzione e distruzione

Possiamo osservare una situazione nel quale serve questa dinamica ad esempio nello stack, infatti ci troveremmo nella situazione in cui quando creavamo lo stack andavamo ad allocare e costruire un numero di elementi pari alla sua capacity, però quello che in realtà vogliamo fare è si occupare spazio pari alla capacity ma senza costruire nessun oggetto.

(costruendo gli oggetti vuoti vorrebbe dire che quando li dovremo costruire effettivamente dobbiamo prima distruggere quelli vuoti e poi costruire nuovi)

infatti costruivamo tutto con una new:

```
stack::stack (const size_type capacity):vec_(nullptr),
capacity_(capacity),size_(0){
if (capacity>0) vec_=new value_type [capacity_];
assert (check_inv());
}
```

Le new di base fanno allocazione e inizializzazione insieme

allocazione senza inizializzazione

per fare l'allocazione senza inizializzazione dobbiamo utilizzare **l'operator new** (non new, ma proprio operator new)

inizializzazione senza allocazione

dato che nella fase precedente abbiamo già allocato ma non inizializzato ora dobbiamo completare la situazione inizializzando, si fa attraverso la **new di piazzamento**

distruzione senza deallocazione

Nel caso in cui volessimo distruggere un oggetto ma mantenendo in nostro possesso lo spazio (ad esempio nello stack vogliamo che la capacity rimanga nostra) dobbiamo semplicemente **chiamare esplicitamente il distruttore "~nome_tipo()"**

è l'unico caso in cui si chiama il distruttore esplicitamente

ed è diverso dalla delete che invece distrugge e dealloca la memoria

deallocazione senza distruzione

Dobbiamo completare la deallocazione dato che lo facciamo in 2 fasi differenti (prima distruzione poi deallocazione), lo si fa attraverso **l'operator delete**

in realtà queste situazioni in cui si devono fare separatamente le azioni sono viste molto di rado

esercizi

stack_lezione23

template metaprogramming

la tecnica del template metaprogramming serve per svolgere calcoli a tempo di compilazione
immaginiamo di voler svolgere la seguente operazione

```
int fact(int n){
    if (n==0) return 1;
    else return n*fact(n-1);
}

int arr[fact(5)];
//non possiamo perché gli array per inizializzarsi vogliono dei valori costanti a tempo di compilazione, e fact(5) viene svolto a tempo di esecuzione quindi in compilazione non sarà ancora risolto
```

constexpr

nelle nuove versioni di c++ potremo usare **constexpr** prima delle funzioni in modo che vengano calcolate a tempo di compilazione

```
constexpr int fact(int n){
    if (n==0) return 1;
    else return n*fact(n-1);
}
```

nelle versioni precedenti

Per eseguire questi calcoli prima della versione in cui veniva introdotto il constexpr si usava il template meta programming, ovvero si sfruttava il fatto che i template siano risolti a tempo di compilazione e che gli enum siano contati come valori costanti, quindi si andava a creare una struttura templatica che attraverso la combinazione delle due cose eseguiva i calcoli a tempo di compilazione

```
template <long n>
struct fact{
    enum value {value=n*fact<n-1>::value};
}

//quindi potremo istanziare l'array come
int arr[fact<5>::value]
```

problema

così facendo avremo che non termineremo mai continuando ad eseguire all'infinito (eseguendo con n sempre minore fino ai numeri negativi ecc)

soluzione

come faremo?
dobbiamo usare la specializzazione di template che ci dirà che arrivato ad aver passato il valore 0 come n eseguirà una forma speciale della struct

```
template <long n>
struct fact{
    enum value {value=n*fact<n-1>::value};
}
template <>
struct fact<0>{
    enum value {value=1};
}

//quindi potremo istanziare l'array come
int arr[fact<5>::value]
```