Sommario

Esercitazione 1

Occhio che ogni volta che andiamo a compilare dobbiamo fare CTRL+F9 con *rebuild all* perché la compilazione standard di dev è leggera e non vede le modifiche alle classi.

Oggi rivediamo la creazione di una libreria, ovvero la separazione del codice in più file per ottenere pulizia e chiarezza.

È molto importante seguire quest'idea per garantire la modularità del codice, oltre che alla già citata sua pulizia.

Per prima cosa dobbiamo dare le direzioni al linker di includere le librerie, questo si fa ponendo all'inizio del codice le definizioni #ifndef #define #endif; tale costrutto ci permette di essere sicuri che non vengano incluse più volte le stesse librerie creando "clash" tra i vari nomi di metodi e variabili.

Quindi andiamo a creare prima la libreria *libreria.h* e poi creiamo la sua implementazione nel file *libreria.cpp* che appunto contiene l'inclusione di *libreria.h* e ne va a definire le firme dei metodi e delle variabili dichiarate.

Le strutture sono dei contenitori dove mettere funzioni, variabili, metodi come gia visto in prog1.

Ma le struct sono sempre tutte pubbliche, noi magari vogliamo mantenere private alcune informazoini, quindi noi andiamo ad usare le classi invece che le struct.

Nelle classi si possono stabilire degli spazi *private*, *public* e *protected*. Private visibile solo all'interno della classe e public visibile anche all'esterno, protected lo andremo a spiegare in seguito.

Qualsiasi attributo dichiarato all'interno di una classe è default privato.

Quando vado a dichiarare un metodo nella classe lo vado a definire poi nel .cpp e non nel .h, nel .cpp devo utilizzare lo *scope* (Classe::metodo) per definire a quale classe si riferisce la definizione di un certo metodo.

Delle classi posso creare più versioni di costruttore con numero di parametri diverso.

Nota sui parametri facoltativi: il valore di default va segnato solo nella libreria .h non nel .cpp !!!

I parametri facoltativi possono essere messi solo in fondo a tutti gli altri parametri.

Nell'esercitazione abbiamo visto come la memoria venga utilizzata come stack perché vedendo l'ordine di attivazione dei distruttori notiamo che vengono distrutte prima le variabili create dopo.

Nota sulla funzione stampa: chi la implementa può andare a modificare gli attributi! Ma esiste un metodo per trasformare i metodi in metodi di sola lettura, ponendo *const* dopo le () nella dichiarazione e nell'implementazione del metodo (sia nel .h che nel .cpp).

Poi proviamo ad allocare un puntatore e creare un'istanza di Persona con una *new*, rimane sempre il classico problema di cancellazione e deallocazione delle variabili.

Poi discutiamo i vari passaggi per valore, riferimento ed indirizzo, la questione è lasciata da approfondire allo studente.

Passaggio di parametri per indirizzo (riferimento) in C++

Il passaggio di parametri in C avviene solo per valore, il passaggio per riferimento è ottenuto passando l'indirizzo di memoria della variabile.

Il C++, pur mantenendo questa modalità, ne presenta una sintatticamente più semplice: il parametro che deve essere passato per riferimento è preceduto, nell'intestazione della funzione, dall'operatore *&*. Questa è la sola cosa da fare, nel senso che una volta fatto questo all'interno della funzione si può usare il parametro senza complesse manovre di dereferenziazione, evitando però il peso del passaggio per copia.

In pratica abbiamo visto come passare per copia o passare per riferimento e come è più veloce e comodo passare per riferimento, però ci si deve ricordare che se non vogliamo correre rischi dobbiamo scrivere *const*.

Esercitazione 2

Dicevamo la volta scorsa che il passaggio per valore potrebbe essere problematico perché se ho allocazioni dinamiche all'interno della mia classe potrei avere problemi di gestione.

Andiamo ad approfondire come risolvere tali problemi utilizzando proprio il costruttore di copia.

Quando ho un attributo dichiarato dinamicamente in un'istanza di una certa classe, il costruttore di copia si *deve* occupare di copiare anche le variabili una a una ciecamente: deve passare da quella che si chiama copia superficiale ad una copia profonda.

Cosa succede nel caso di copia superficiale? Vado a copiare solo il puntatore all'area di memoria, questo mi mette in pericolo perché è possibile che una volta eliminato il puntatore copia venga ripulita l'area di memoria che però è ancora puntata dall'istanza originale, ciò significa che vado a perdere dati!

Nota che il costruttore di copia di default in C++ è un costruttore che compie copia superficiale.

Nota però che quando hai degli attributi dinamici in una classe devi prestare attenzione a deallocarli una volta che invochi il distruttore.

Se capita all'esame che ci chiedano una classe con attributi dinamici DOBBIAMO per forza creare costruttore di copia profonda e distruttore intelligente.

Il costruttore di copia riceve in input un'istanza passata per referenza e *const*.

Stampare una classe

Andiamo ora invece a implementare una serie di metodi pubblici che ci permettano di vedere i parametri (altrimenti privati) della classe. In questo caso facciamo ben attenzione a mettere il qualificatore *const* perché è buona programmazione.

Andiamo ora a ridefinire l'operatore << per Persona in modo da stampare im maniera più semplice.

cout lavora su uno stream, dunque devo fare in modo di ritornare un flusso per << applicato a Persona, perché devo fare in modo che sia collegabile a ciò che precede e ciò che segue, che sono anch'essi stream;

```
ostream& operator <<(ostream& precedente, const Persona& p){
    return precedente << p.get_nome() << " anni: " << p.get_eta();
}
```

In questo caso vediamo che passiamo lo stream precedente e passiamo un'istanza di persona per riferimento.

Nota che nell'overload di << non dobbiamo mai andare a capo all'interno della stampa, perché l'utente non si aspetta questo comportamento.

Sostanzialmente in C++ non esiste un toString(), il modo per stampare una classe qui è proprio quello di sovrascrivere l'operatore <<.

Costruttori chiamati al volo

Il C++ ci permette di creare un'istanza di una classe direttamente nella chiamata di una funzione, ma non solo!

Se invece di passargli un'istanza gli passiamo gli argomenti del costruttore di una certa classe lui proverà comunque ad istanziare la classe, il che è una porcheria!

```
class Foo{
    private:
        int dato;
    public:
        Foo(int d) { dato=d; cout << "costruttore: " << dato << endl; }
        int get_dato(){ return dato; }
};

void stampa(Foo f){
    cout << f.get_dato() << endl;
}

int main(int argc, char** argv) {
    Foo a(4);
    stampa(a);
    stampa(Foo (9))
    stampa(33);

    return 0;
}
```

Funzionano tutti!

Per evitare questa cosa possiamo dichiarare il costruttore come *explicit* e quindi fare in modo che possa essere chiamato solo esplicitamente.

```
explicit Foo(int d) { dato=d; cout << "costruttore: " << dato << endl; }
```

Funziona meglio.

Però Roberti dice che potrebbe essere anche una feature (e non un bug) dipende da cosa vogliamo noi;

Attributi all'interno di una classe

Gli attributi *const* non si possono settare nel costruttore, si devono inizializzare per forza di cose nella dichiarazione degli attributi.

Esiste però un modo per inizializzare il valore degli attributi di una certa classe anche prima di istanziarla, nel seguente modo:

```
class A{
    private:
        int k;
        const int d=6;
        const float e=2.7;
        static int num_istanze;
    public:
        A():d(3){
            k=0;
//          d=9; non si può fare
        }
        A(float val, float val2):e(val),d(val2){
            k = 7;
    }
```

Come funzionano gli attributi static in C++?

Li si dichiara all'interno della classe ma li si può inizializzare solo fuori dalla classe! Altrimenti ogni volta che vado ad istanziare quella classe vado anche a sovrascrivere la variabile statica.

Però occhio! Tutti i metodi della classe che vanno a maneggiare attributi *static* dovrebbero essere statici essi stessi, perché uno potrebbe voler andare ad usare il metodo prima di istanziare un oggetto di tale classe, e se il metodo lavora con attributi statici dovrebbe poter funzionare lo stesso, l'unico modo per far che questo accada dobbiamo dichiarare il metodo come *static*.

Si possono dichiarare anche attributi sia *static* che *const*, ma con un plot twist!

Si può fare con gli interi ma non si può fare con i float (dipende dal compilatore in realtà).

Come facciamo ad inizializzare quindi anche glil *static const float*? L'unico modo è inizializzarli al di fuori della classe.

```
class A{
    private:
        int k;
        static const int a = 0;
        static const float b = 3.2; // non funziona
        static const float b;
        const int d=6;
        const float e=2.7;
        static int num_istanze;
    public:
        // Costruttore a 0 param
        A():e(9.8),d(3){
            k=0;
//          d=9; non si pu fare
            num_istanze++;
        }
};

int A::num_istanze = 0;
const float A::b = 0.1;
```

Abbiamo due costanti statiche, l'intera può essere inizializzata all'interno della classe, la float no, di fatto andiamo ad assegnarle un valore al di fuori della classe tramite lo scoping.

Salvare su file il contenuto di un oggetto

Si utilizza la libreria *fstram*, ora andiamo a vedere la modalità più generica.

```
~A(){
    num_istanze--;
    cout << "distruction! num istanze:" << num_istanze << endl;
    ofstream my_file;
    my_file.open("num_istanze.txt", ios::app);
    my_file << "num_istanze=" << num_istanze << endl;
    my_file.close();
}
```

In questo caso andiamo ad inserire nel distruttore un routine che va a scrivere sul file "*num_istanze.txt*" (nella stessa cartella, a meno che non specifichi un percorso diverso) in modalità append (*ios::app*) tramite una modalità molto simile al *cout*, nel senso che si gestisce il file come uno stream.

Esercitazione 3

Overload degli operatori

Nota sugli operatori binari:

Un operatore definito all'interno di una classe per essere invocato nel main richiede che sia invocato con il primo parametro (di sinistra) dello stesso tipo della classe. Un operatore dichiarato all'esterno della classe richiede che uno dei due parametri (l o r, indifferente) sia di quella classe e che l'altro sia convertibile ad un'istanza di quella classe.

Andiamo a sovrascrivere l'operatore <<.

Questo operatore genera un ostream& (ostream passato per referenza). Dobbiamo scrivere le seguenti righe di codice:

```
ostream& operator << (ostream& os, const A& a){
    return os << a.k;
}
```

Questo metodo però funziona solo se k è pubblico, altrimenti mi tocca creare nella classe A un metodo *get_k*() che mi permetta di recuperare il calore di

Nota che l'istanza della classe A viene passata by reference e **deve** essere *const*. In C++ quando sovrascrivo << devo sempre scrivere *const*.

Nota che all'interno della stampa devo richiamare solo metodi dell'istanza di A che sono *const*, altrimenti il compilatore si arrabbia.

Ma come posso fare per non creare 17 metodi get_parametro()?

Posso dichiarare l'operatore come *friend* della classe! Quando un metodo viene dichiarato in una classe come *friend* può accedere agli attributi private della classe stessa.

Gli operatori si possono sovrascrivere all'interno della classe stessa.

Nota che si possono sovrascrivere solo gli operatori del linguaggio, non posso crearne di nuovi.

```
// preincremento
A& operator ++ (){
    k++;
    return *this;
    cout << "operator ++ pre" << endl;
}
// postincremento
A operator ++(int){
    cout << "operator ++ post" << endl;
    A a = *this;
    k++;
    return a;
}
```

Postincremento e preincremento

Quando implemento l'overload del postincremento devo ricordarmi di ritornare il valore in ingresso ma modificare la variabile originale, nel codice riportato usiamo il copy constructor per ritornare un temporaneo.

Nota che l'operatore di post-incremento/decremento richiede che sia indicato un int dummy come parametro passato per disambiguare. Un altro fatto interessante del post incremento è che modifica il valore della variabile da incrementare e ne ritorna il vecchio valore ottenuto con il copy constructor prima della midifica.

Cosa succede quando voglio confrontare due istanze di classi diverse? (operatori ==, < o > ecc.)

Se provo a risolvere creando metodi interni ho un sacco di problemi per capire quale delle classi posso dichiarare prima o dopo e come far capire al compilatore che una classe presenta taluni attributi che vengono utilizzati nel confronto.

Quando devo comparare classi diverse è assolutamente sconsigliato creare metodi interni, si tenta sempre di risolvere usando metodi esterni.

Ma c'è un problema con questo approccio! Da metodi esterni non posso accedere agli attributi privati, quindi se io scrivo le classi bene il metodo per raggiungere gli attributi deve usare un sacco di get_attributo()… oppure s può dichiarare il metodo come amico di entrambe le classi.

Però anche qui rimane un problemino, avendo definito l'operatore A == B non so come affrontare operazioni del tipo B == A: l'unica soluzione possibile è di riscrivere anche questa versione (dello stesso metodo).

Si arriva quindi a una soluzione di questo tipo:

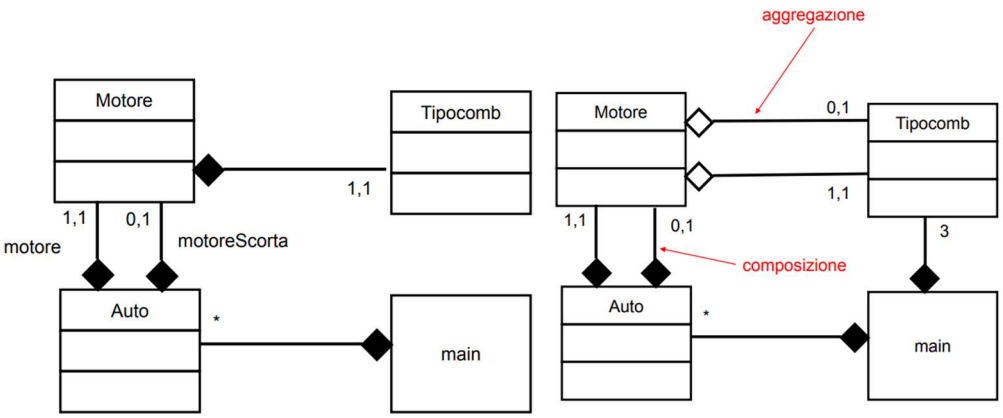
```
class B{
    public:
        int val;
        B(){ val = 5; }
        friend bool operator == (const A& a, const B& b);
};

bool operator == (const A& a, const B& b){
    return a.val == b.val;
}

bool operator == (const B& b, const A& a){
    return a==b;
}
```

Nota che qui abbiamo utilizzato lo sgamo di richiamare a==b nel secondo metodo di confronto, quindi non ci serve dichiarare entrambi i metodi come friend delle classi.

Esercitazione 4

Rappresentazione UML

In questa esercitazione vediamo il concetto della rappresentazione in forma di UML che va a definire le proprietà di una classe in modo visivo.



Partiamo con la creazione della classe combustibile. Qui è interessante ricordare la modalità di utilizzo degli enum, che vengono indicati come etichette e dichiarati tramite la keyword *typedef enum … .*

```
typedef enum Combustibile{
    BENZINA, DIESEL, GPL
    // queste sono in maiuscolo perch sono etichette
}Combustibile;

class Tipocomb{
    private:
        Combustibile comb;
    public:
        Tipocomb(Combustibile _comb);
        ~Tipocomb();
        Combustibile get_combustibile()const;
};
```

Una volta terminato il combustibile, passiamo al motore.

Il motore contiene un'istanza di combustibile; quando facciamo il costruttore del motore dobbiamo anche farci passare il tipo di combustibile del motore (un motore senza combustibile non ha senso). Quindi vado a creare il costruttore con due parametri (cilindrata e combustibile).

Impariamo la good practice di passare invece che un'istanza della classe *Tipocomb*, i parametri che servono per istanziare quell'oggetto di classe *Tipocomb*, così da aumentare la flessibilità del codice al cambio di implementazione della classe *Tipocomb*.

Ora però osserviamo che le istanze di Tipocomb non saranno mai più di tre, perché possono esserci solo tre tipi di combustibile e questi possono essere usati da tutti i motori, quindi non ha senso la classe *Tipocomb* che posso istanziare quante volte voglio, ha più senso creare di default tre oggetti per i tre tipi di combustibile e poi nel codice usare quelli ogniqualvolta ne ho di bisogno.

In pratica la modifica più grossa consta nel cambiare costruttori e metodi di stampa: ora il parametro tipo invece che essere un'istanza di *Tipocomb* è un puntatore a *Tipocomb*, quindi quando un motore viene creato gli viene passato per argomento il puntatore a un combustibile ed il motore da quel punto in poi farà riferimento (letteralmente) a tale istanza del combustibile.

Ora passiamo alla costruzione della classe *Automobile* (non auto perché auto è una parola riservata del C++).

In questa classe avremo necessariamente un motore primario (dato che così è indicato sull'UML) ed avremo anche facoltativamente un motore secondario.

Innanzitutto, cosa ci serve per creare un'istanza di Automobile? Marca, modello e motore, ma noi vogliamo mantenere nascosta l'implementazione di motore nel main, quindi passiamo invece che un'istanza di motore, i parametri per crearne una, ovvero cilindrata e *Tipocomb**.

Come in precedenza l'istanza di motore va inizializzata prima dell'ingresso nel costruttore perché potrebbe non avere (non ce l'ha infatti) il costruttore a zero parametri e quindi generare errori quando viene chiamato il costruttore di *Automobile*.

```
Automobile::Automobile(string ma, string mo, int cilindrata, Tipocomb* comb):
                    motore(cilindrata, comb){
                        //Automobile::Automobile(string ma, string mo, int cilindrata, Tipocomb* comb){
                        marca = ma;
                        modello = mo;
                        motoreScorta = NULL; // IMPORTANTISSIMO
}
```

In questo modo passiamo i parametri per costruire il motore direttamente al costruttore a due parametri, così da evitare che venga istanziato un motore "vuoto" con costruttore a zero parametri, che darebbe errore in quanto il costruttore a 0 parametri per motore non esiste.

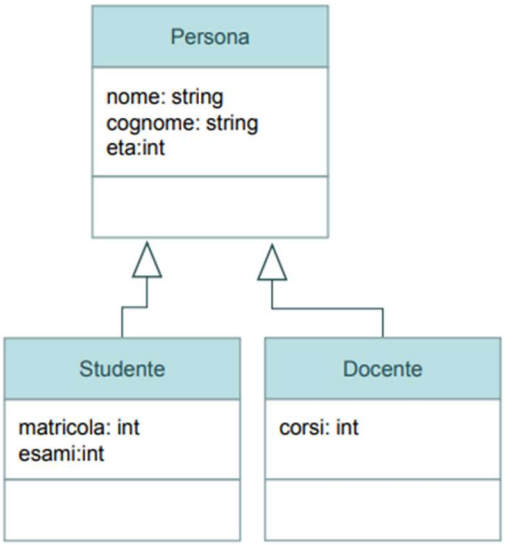
Poi sostanzialmente abbiamo affrontato il problema della composizione 0,1 del motore secondario (ovvero la possibilità che la macchina abbia o meno un motore secondario.

In questo caso dato che il motore secondario deve poter essere aggiunto in seguito è necessario implementarlo come puntatore perché possa essere allocato nell'heap, questo comporta tali conseguenze:

- Nel costruttore di *Automobile*, *motoreSecondario* deve essere inizializzato a NULL.
- Si deve creare un metodo *aggiungiMotoreSecondario* per *Automobile* a cui si passano i parametri per la costruzione di un motore e che crea un'istanza di motore nell'heap
- Si deve modificare il metodo di stampa di *Automobile* per stampare anche il motore secondario quando è presente
- Si deve modificare il distruttore di *Automobile* per eliminare il motore secondario se esiste
- Si dovrebbe implementare un costruttore di copia profonda
- Perché non fare anche il move constructor a questo punto?

Esercitazione 5

Oggi parliamo di ereditarietà.



In primis andiamo a definire la classe Persona.

Questa classe contiene un metodo di stampa, un creatore ed un distruttore.

Nulla di notevole.

Passiamo quindi alla creazione della classe ereditiera *Studente*. Questa classe viene dichiarata nel seguente modo:

class Studente: public Persona{

Il costruttore deve avere questa forma:

```
Studente::Studente(const string& no, const string& co, int ma, int et):
            Persona(no, co, et){
    matricola = ma;
    esami = 0; // lo studente quando inizia ha fatto 0 esami
//  cout << "Studente creato: " << nome << endl; non posso perch nome  privato
    cout << "Studente creato" << endl;
}
```

Nota che per costruire Persona invochiamo il costruttore di persona prima dell'inizializzazione del nostro studente, altrimenti verrà invocato il costruttore vuoto di Persona (che non esiste e genera errori).

Occhio al metodo di invocazione, è diverso da quando abbiamo composizione o aggregazione, in cui dopo i due punti mettiamo il nome dell'attributo per invocare il costruttore => qui usiamo il nome della Classe da cui ereditiamo.

Posso invocare il metodo stampa pur non avendo dichiarato un metodo per stampare *Studente*.

Questo perché posso accedere a tutti gli attributi pubblici della classe da cui eredito.

Ma che succede se mi definisco un metodo con lo stesso nome in Studente?

Sorpresa: il metodo di stampa dello *Studente* va a sovrascrivere la stampa di persona! E se voglio chiamare la stampa di *Persona* devo indicare lo scope.

Quello che facciamo quindi è andare ad implementare un metodo pubblico di stampa per la classe *Studente*, questo non può accedere alle variabili private di *Persona*! Quindi non può stampare nome, cognome ed età.

Per sovvenire a questo problema andiamo a modificare lo scope di queste variabili nella classe *Persona*, riqualificandole come protected. In questo modo le rendiamo disponibili per qualsiasi classe erediti da *Persona*.

Così possiamo stampare tranquillamente tutto nel nostro metodo *Studente.stampa()*.

A questo punto possiamo anche andare a fare l'overload dell'operatore << per la stampa di *Studente*.

Utilizziamo la tecnica di dichiarare il metodo come *friend* della classe. Però se dichiariamo il metodo coì:

return os << "nome=" << s.nome << " cognome=" << s.cognome << " matricola=" << s.matricola;

non stiamo sfruttando l'ereditarietà... io posso stampare così solo perché conosco l'implementazione di Persona ma cosa succede se l'implementazione di Persona cambia?

Quello che abbiamo visto poi in effetti è la dichiarazione nel main di un puntatore *pp* a *Persona* che però viene legato (dinamicamente con una *new*) ad un'istanza di *Studente*.

In seguito abbiamo provato ad utilizzare *pp->stampa()*, ma questo va a riferirsi al metodo stampa di *Persona*!!

Questo avviene perché *pp* è dichiarato come *Persona* e quindi la ricerca del metodo va a trovare quello di *Persona*.

Ma c'è di peggio, la stessa cosa succede anche quando uso *delete pp* e vado ad eliminare l'istanza nell'heap, ovvero viene chiamato il distruttore di *Persona*! Questo può provocare grossi problemi.

come risolviamo questo riferimento? aggiungendo il qualificatore *virtual* a quei metodi di *Persona* che vengono modificati (sovrascritti) dalle classi ereditiere.

Nota che questi problemi arrivano solo quando dichiariamo un puntatore a *Persona* e poi istanziamo uno Studente; comunque sono cose da non tralasciare.

In ultima abbiamo visto come questo problema si presenti anche con l'operatore << che però non può essere qualificato come virtual perché **non è un metodo della classe** *Person*.

In questo caso la soluzione è quella di creare una funzione farlocca (dummy) nella classe *Person* che viene utilizzata per costruire l'ostream& che verrà poi ritornato dall'operatore <<.

A questo punto l'operatore << va a richiamare la funzione farlocca della classe in modo che tutte le classi ereditiere possano sovrascrivere la funzione farlocca per sovrascrivere automaticamente l'overloading dell'operatore.

Nella classe *Persona*

```
        virtual ostream& stampaoperator(ostream& os)const;
        friend ostream& operator << (ostream& os, const Persona& p);
};

ostream& operator << (ostream& os, const Persona& p);
```

Nell'implementazione di *Persona*

```
ostream& Persona::stampaoperator(ostream& os)const{
    return os << "Nome: " << nome << " Cognome: " << cognome << " eta: " << eta;
}

ostream& operator << (ostream& os, const Persona& p){
    return p.stampaoperator(os);
}
```

Quindi è chiaro come sovrascrivere la stampa: sovrascrivendo la funzione *stmapaoperator*.

Nota sui modi in cui si eredita dalle classi antenate.

Se B eredita in modo public da A, tutti gli attributi public di A saranno public in B e tutti gli attributi protected in A saranno protected in B.

Se B eredita in modo protected da A, tutti gli attributi di B saranno protected.

Se B eredita in modo private da A, tutti gli attributi di B saranno private.

Esercitazione 6

Oggi ereditarietà multipla.

Andremo a creare la classe studente-lavoratore che eredita da entrambe le classi Studente e Lavoratore.

L'ordine di ereditarietà è rispettato quando viene invocato il costruttore della classe ereditaria!

Se io dichiaro la classe studente lavoratore in questo modo

```
class StudenteLavoratore: public Studente, public Lavoratore{
```

, l'ordine di chiamata dei costruttori e distruttori sarà il seguente:

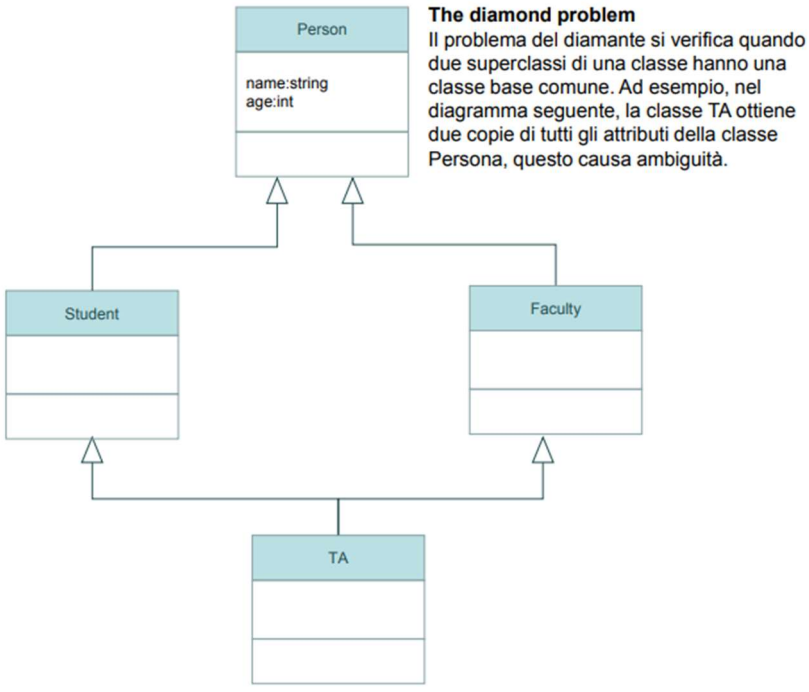
```
creato Studente
lavoraore creato
StudenteLavoratore creato

StudenteLavoratore distrutto
lavoraore distrutto
Distrutto studente
```

Ma cosa succede quando vado a chiamare un attributo che appartiene ad entrambe le classi da cui eredito?

Il compilatore si arrabbia perché sono ambiguo!

Devo specificare lo scope della funzione (indicare a quale classe da cui eredito mi riferisco) ad esempio in questo modo s1.Lavoratore::stampa().

Ora andiamo a vedere il problema del diamante (quando eredito da due classi che ereditano da una classe comune a loro volta). Questo problema, come vedremo, comporta gravi ambiguità.



The diamond problem
Il problema del diamante si verifica quando due superclassi di una classe hanno una classe base comune. Ad esempio, nel diagramma seguente, la classe TA ottiene due copie di tutti gli attributi della classe Persona, questo causa ambiguità.

Problemi:

1. Chiamo due volte il costruttore di Persona quando creo una TA!!
    a. Per risolvere dobbiamo ereditare virtualmente
    b. Ma questo implica la necessità del costruttore a 0 parametri per Persona
2. Una volta che inserisco il virtual risolvo il problema e la classe Persona viene istanziata un'unica volta.
3. Una cosa interessante è che comunque vengono passati i parametri ai costruttori delle classi da cui eredito (da Studente e da Faculty posso stampare la x che ricevo come input)

```
class Persona{
    protected:
        int eta;
    public:
        Persona(){
            eta = -1;  cout << "Creata persona di default" << endl; };

        Persona(int a){
            eta = a;  cout << "Creata persona: " << eta << endl;    };
};

class Studente: virtual public Persona{
    public:
        Studente(int x):Persona(x){
            cout << "Creato studente (" << x << ")" << endl;
        };
};

class Faculty: virtual public Persona{
    public:
        Faculty(int x):Persona(x){
            cout << "Creata Faculty (" << x << ")"<< endl;  };
};

class TA: public Faculty, public Studente{
    public:
        TA(int x):Studente(x), Faculty(x){
            cout << "costruttore TA" << endl;
        }
};
```

Eccezioni

Le eccezioni sono la possibilità di intercettare una serie di errori che non ho a compile time.

gli errori di runtime sono quegli errori che NON possono essere rilevati in fase di compilazione, perché si manifestano solo durante la fase di esecuzione del programma e solo in alcune particolari circostanze, ossia ai veri casi di "eventi eccezionali".

Alcuni esempi di errori di runtime sono:

- divisione per 0;
- l'utilizzo di un pathname non valido o che fa riferimento ad una periferica che in un certo momento risulta scollegata;
- overflow su un tipo di dato da parte di un'operazione aritmetica;
- un'operazione di casting non valido (ad esempio, quando si ha l'inserimento in input da parte dell'utente di una stringa di testo non interpretabile come un numero e che viene assegnata ad una variabile di tipo numerico).

L'eccezione ci permette di intercettare questo tipo di errori e di bloccare il programma o addirittura di ignorare o "skippare" il problema.

Si tratta di errori pericolosi perché se non gestiti, possono generare anomalie di funzionamento che possono determinare persino il blocco inaspettato del programma (crash dell'applicazione). La gestione di questo tipo di errori non è banale ed è per questo che per facilitarla i linguaggi di programmazione ad oggetti (OOP) mettono a disposizione il meccanismo delle ECCEZIONI.

Quando si verifica un errore di runtime i linguaggi OOP, si dice, possono "sollevare" o permettono di "lanciare" (in inglese, to throw) un'eccezione. Ciò in generale di traduce nella creazione di un oggetto che appartiene ad una classe particolare, dipendente dallo specifico errore di runtime che si è verificato.

Questo meccanismo, infatti, prevede che quando viene sollevata un'eccezione, il flusso di esecuzione del programma venga sospeso nel punto in cui si è verificato l'errore e salti in un altro punto del codice in cui esso verrà gestito. Nel linguaggio C++ ciò viene realizzato utilizzando il costruttto try-catch.

Questa è la struttura del blocco try-catch

```
bool compare(int a, int b){
    if(a<0 || b<0){
        throw "Hai inserito un numero negativo!";
    }
    return a == b;
}

void test(){
    try{
        cout << compare(2, -5) << endl;
    } catch(const char* ex) {
        cout << ex << endl;
    }
}
```

Un'eccezione può essere o sollevata direttamente dall'istruzione in corrispondenza della quale si verifica l'errore, oppure se per una certa istruzione ciò non è previsto dal linguaggio, può essere lanciata esplicitamente con un'istruzione throw con la seguente sintassi: *throw espressione;*

throw è la parola chiave con cui è possibile lanciare un'eccezione per segnalare il fatto che si è verificato un errore. L'eccezione viene "marcata" opportunamente, per permettere il suo riconoscimento da parte dei blocchi catch.

Le eccezioni si possono pure propagare! Inoltre, esistono diversi tipi di eccezioni.

Nota questo fatto interessante: le eccezioni possono essere gestite in luoghi diversi rispetto a dove sono state sollevate.

Quando viene lanciata un'eccezione si interrompe l'esecuzione del codice nel blocco try, le operazioni all'esterno del blocco try vengono comunque eseguite!

Però se manca il blocco try, tutto il programma si blocca.

Ricorda che il messaggio passato dal throw e ricevuto dal catch deve essere un **const char *** .

Tipi di eccezioni

Esistono vari tipi di eccezioni presenti nella libreria *stdexcept*, queste eccezioni possono essere lanciate (throw) in qualsiasi punto del programma, vanno chiamate subito dopo il throw e.g.: *throw invalid_argument("Valore negativo!");*

Si noti che alle eccezioni si può passare sempre una stringa esplicativa che può essere recuperata nel blocco catch tramite il comando *ex.what()*.

Se nel blocco catch vogliamo prendere qualsiasi tipo di eccezione dobbiamo passare … come argomento del blocco catch (e.g.: *catch(…) { <codice gestione eccezione> }*).

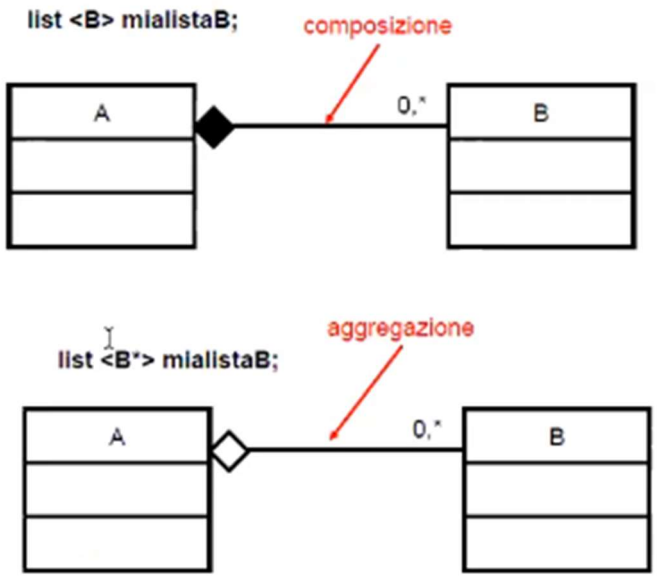
Esercitazione 7

Standard Template Library e aggregazione e composizione.

Vediamo le differenze tra list e vector;

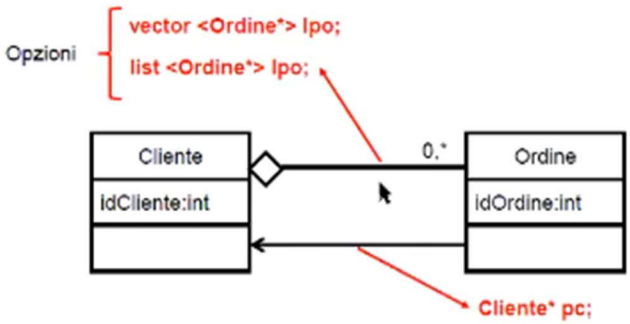
Vector è molto veloce per quanto riguarda la ricerca, mentre è lento per quanto riguarda gli inserimenti e le rimozioni.

List al contrario è molto veloce per inserimenti e rimozioni ma molto lento per la ricerca.



Oggi andremo a vedere le relazioni di composizione ed aggregazione con liste e vettori.

Nel caso della composizione si tratta di istanziare le B direttamente all'interno della classe A, nel caso di aggregazione si tratta di mantenere le B all'esterno della classe A e di tenere in A solo i riferimenti alle B create altrove.



In questo caso dobbiamo creare Cliente con aggregazione per Ordine, ed Ordine dovrà avere il riferimento a Cliente (guai di linking in vista, una delle due classi dovrà essere dichiarata per prima).

Prima di tutto creiamo Ordine ma senza navigabilità verso cliente, facciamo le cose semplici prima, poi quando avremo creato tutto andremo ad aggiungere la navigabilità.

Come salviamo gli ordini in Cliente? La scelta migliore è usare una lista, perché il Cliente potrebbe non avere alcun ordine, in questo caso ci torna comodo l'utilizzo di una lista, perché questa parte già di default vuota, quindi non dobbiamo nemmeno preoccuparci di come differenziare tra un puntatore non inizializzato ed uno inizializzato.

La lista è un contenitore template.

Dato che il Cliente non ha l'obbligo di avere ordini il suo costruttore non deve prendere come parametri ordini.

Quando creo il distruttore devo sapere che non è compito della classe Cliente andare a deallocare ogni istanza di Ordine, ci pensano la List o il Vector che quando vengono deallocate (perché Cliente viene deallocato) vanno ad invocare il distruttore di ogni elemento che contengono (le classi della STL sanno già come fare).

Naturalmente per avere una aggregazione 0…* bisogna avere almeno un metodo per aggiungere un Ordine al Cliente.

È sensato utilizzare (il=ordini.begin(); il<ordini.end(); ++il) ? no! Perché usando List non si è sicuri che gli elementi vengano inseriti in memoria in ordine, quindi il < ha poco senso!

Ma compilando abbiamo problemi, perché? Il fatto è che non possiamo dichiarare questo metodo di stampa come const, perché la lista, mentre la attraverso, tiene traccia di come ci muoviamo e viene quindi modificata mentre la scorriamo, di conseguenza quando usiamo gli iteratori non possiamo usare const.

```
void Cliente::stampa(){  // non ci va il const!!!
    list<Ordine*>::iterator il; // scorro in ordine
    for(il=ordini.begin(); il!=ordini.end(); ++il){
        cout << **il;
    }
    cout << endl;
}
```

A questo punto dobbiamo creare la navigabilità. Siccome l'ordine può nascere senza cliente dobbiamo aver cura di inizializzare il puntatore a Cliente a null.

È qui che il compilatore inizia ad incavolarsi. Non sa chi linkare prima a chi, quindi devo dichiarare in qualche modo che Cliente è una classe.

In cliente.h:

```
class Ordine;

class Cliente{
    private:
        int idCliente;
        list<Ordine*> ordini;
    public:
```

In ordine.h:

```
class Cliente;

class Ordine{
    private:
        int idOrdine;
        Cliente *pc;
    public:
```

Ma non sono finiti i problemi, ci sono problemi con la stampa degli ordini perché in Cliente li stampiamo in un metodo non const mentre in Ordine ne definiamo la stampa come const, dobbiamo quindi andare a togliere il const dalle stampe in classe Ordine.

Alla fine, i problemi sono belli grossi perché nella stampa di Cliente utilizziamo l'overload di << di Ordine, che non viene caricato prima di Cliente.h, quindi devo andare a spostare la stampa di Cliente in un file cliente.cpp, così diamo tempo di linkare l'overload di << per Ordine.

Gli errori sono così brutti che dobbiamo andare a creare addirittura ordine.cpp;

Poi proviamo a costruire una stampa per Ordine, ma non possiamo stamparci dentro anche il Cliente! O almeno non possiamo invocare la stampa di Cliente che a sua volta richiama il metodo di stampa di tutti gli ordini, perché sennò finiamo in un loop.

Nella stampa in ordine reverse dobbiamo utilizzare un reverse_iterator e dobbiamo partire da rbegin ed arrivare a rend.

Adesso ci facciamo una bella carrellata dei modi per scorrere le liste in c++.

Il primo modo che vediamo è l'utilizzo della parola *auto*.

Vediamo che questo metodo è più semplice perché ci leva l'incombenza di dichiarare ogni volta gli iteratori, ma sostanzialmente non cambia nulla.

Ma esiste un modo "ancooooora più furbo", lo riporto qui sotto perché è sexy:

```
for(auto& item : s){
        cout << item << " ";
}
```

Purtroppo però così non possiamo stampare al contrario in questo modo. Per stampare al contrario dobbiamo usare la libreria algorithm e fare il reverse del vector, ma non conviene.

Notiamo anche che la keyword auto può essere usata come lo zucchero (cioè può essere messa dappertutto).

Oltre al costrutto auto andiamo anche a vedere il costrutto range based.

Questo costrutto funziona come auto ma permette di specificare il tipo, questo ci permette di chiarire se vogliamo che gli item su cui iteriamo siano modificabili o meno, in un ciclo range based se dichiaro

for(int item : v)

allora item sarà in sola lettura e non potrò modificarla (lavoro su una copia del dato), nel caso

for(int& item : v)

item sarà modificabile, perché ne avrò copiato il riferimento.

Ma c'è anche da notare che se stiamo iterando su strutture complesse utilizzare il for range based con la copia (non modificabile) potrebbe essere super pesante da fare.

Con gli iteratori invece possiamo istanziare iteratori costanti, quindi avere il passaggio per riferimento con la garanzia di const.

Anche all'interno del metodo auto è stato inserito il metodo per usare copia per riferimento costante, con il costrutto

for(auto& item : as const(v)) // solo dal c++17

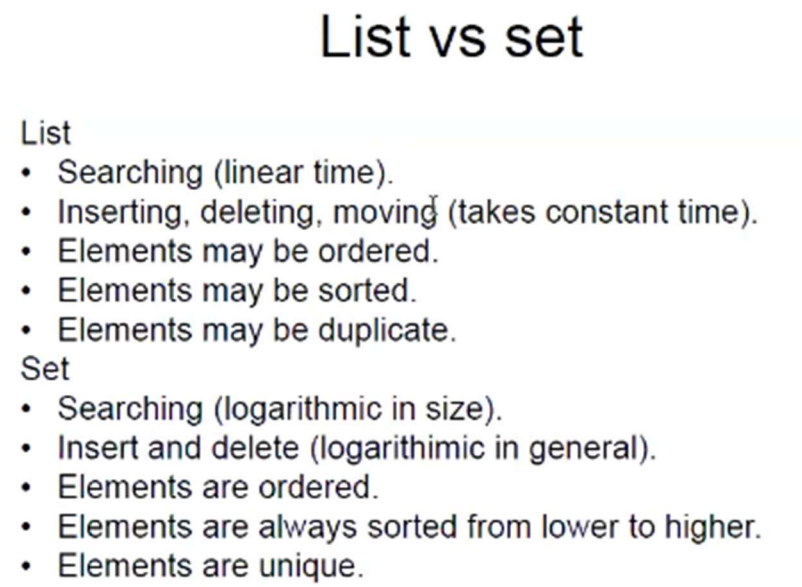
ma si può fare pure

for(const auto& item : v)

Esercitazione 8

Il contenitore Set della STL

Partiamo con il vedere approfonditamente la differenza tra list (lista doppiamente concatenata) e set.



Dato che il set mantiene gli elementi ordinati quello che mettiamo all'interno del set deve avere l'operatore < definito, poiché il set utilizza questo operatore appunto per definire l'ordinamento tra gli elementi.

Non si può modificare un elemento che è presente all'interno del set se non rimuovendolo prima e reinserendolo poi.

Oggi vedremo due tipi di set, ordered e unordered.

Il set offre il metodo per trovare un elemento .find(…) che ritorna un iteratore che punta all'elemento cercato, se non trova l'elemento ritorna l'iteratore end().

Se voglio eliminare un elemento dal set devo 1) cercarlo con find 2) verificare che esista verificando che il'iteratore restituito da find sia diverso da end() e poi 3) utilizzare il metodo .erase(*iteratore*) per eliminare finalmente l'elemento dal set.

Nota che utilizzare gli iteratori costanti con il set in realtà non è una cosa troppo sensata perché appunto gli elementi del set non possono essere modificati di defualt.

L'unordered set è come il set ma non mantiene l'ordine, è più veloce per quanto riguada l'inserimento ma più lento per quanto riguarda la ricerca.

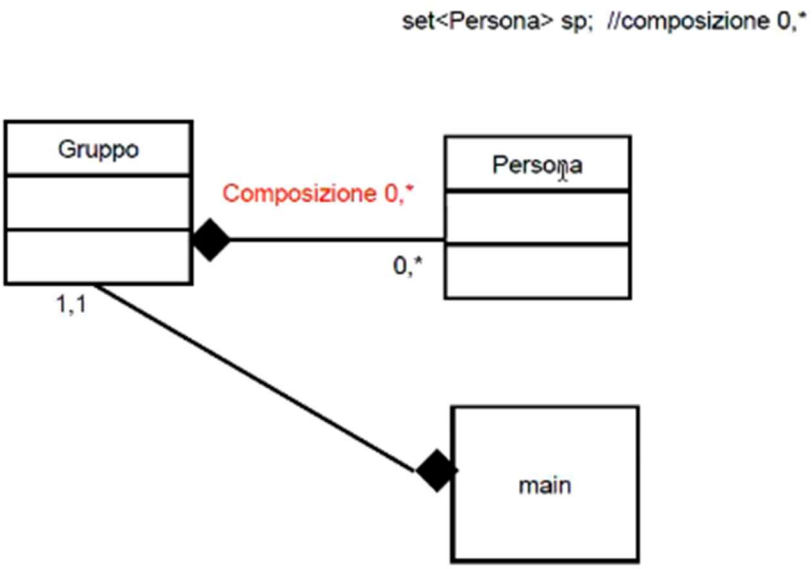
Esistono anche i set in cui si possono inserire valori multipli, questi sono detti appunto multiset.

Nota che il metodo find() ritorna l'iteratore al primo elemento uguale al parametro passato.

Esistono anche gli unordered multiset, guess what they are.

L'unordered multiset utilizza la strategia di inserire elementi simili in un sottocontenitore unico, questo viene effettuato per ottimizzare le performance della struttura dati.

Ora risolviamo il seguente esercizio complesso per vedere a fondo l'utilizzo del contenitore set:



Notiamo queste cose durante l'esercitazione:

- Non possiamo creare set di Persona se non facciamo l'overload dell'operatore < ;
- Le stringhe hanno già l'operatore < definito che riordina in ordine alfabetico (seguendo la tabella ASCII, quindi maiuscole prima delle minuscole ecc.), top;
- Fare sempre l'override di << perché il set (e quindi l'iteratore) non può accedere agli attributi privati, quindi per la stampa non può raggiungerli;
- Normalmente non passo istanze di persona, ha senso passare solo il nome e fare in modo che sia Gruppo a chiamare il costruttore di persona, d'altronde il main non contiene nemmeno il riferimento a Persona;
- I tipi di dati andrebbero inseriti nel .h, nel .cpp vanno inserite solo le utility tipo iostream per stampare;

Esercitazione 9

*Composizione ed aggregazione 0 **

Oggi useremo le mappe, che sono dei contenitori della STL con queste caratteristiche:

- Dimensioni variabili
- Recupera velocemente i dati data la chiave che li identifica
- Iterabili in entrambe le direzioni (reversibili)
- Internamente i valori sono sempre ordinati (per garantire l'efficienza)
- Vengono implementate tramite alberi binari di ricerca, quindi quasi tutte le operazioni hanno complessità log(n)

Le Map naturalmente usano i generics, prendono come tipo una coppia di tipi. Di fatto quando voglio inserire qualcosa in una mappa devo inserirlo in forma di pair (chiave, valore).

Se provo ad inserire un elemento che è già presente non succede niente; per modificare un valore già presente dobbiamo usare la mappa come array.

```
map<string,int> m;

//inserisco delle coppia chiave, valore [MODO1]
m.insert(  pair<string,int> (string("Pinco"),1)  );
m.insert(  pair<string,int> (string("Panco"),1045)  );

//inserisco delle coppia chiave, valore [MODO1-BIS]
pair<string,int> ss(string("Punco"),23);
m.insert(ss);

//non è possibile multipla inclusione se chiave già presente
m.insert(  pair<string,int> (string("Punco"),2000)  );

// modifica chiave già presente
m["Punco"]=2000;

//inserisco una nuova coppia chiave, valore [MODO2]
["Ponco"]=3000;

//verifico se esiste un elemento
if (m.count("NN")>0){ cout <<"elemento esiste"<<endl;}
```

Si possono usare tutti i tipi di iteratori classici anche sulle mappe. Si accede agli elementi tramite gli attributi first e second.

```
void Gruppo::addPersona1(const Persona& p){

        mp.insert( pair<int, Persona>(p.get_id(), p) );

        mp[ p.get_id() ] = p;

}
```

Nota: quando vado ad inserire una coppia nella mappa se lo faccio come mostrato nella seconda riga non devo aver implementato operatori strani per la classe Persona perché la mappa a solo ad inserire il valore: se la chiave è già presente non fa niente, se la chiave non c'è va a creare una nuova associazione chiave-valore salvandosi il valore. Se invece provo ad inserire come mostrato in riga 3 allora devo per forza definire il costruttore a 0 parametri per Persona, perché la mappa andrà prima a creare una coppia chiave-valore con la chiave ed un istanza di Persona creata senza parametri, poi andrà a modificare l'istanza di Persona.

Questa esercitazione ci insegna come creare nel modo migliore le relazioni 0,*.

Nel primo UML è specifata la realzione di composizione 0* tra Gruppo e Persona, fatto sta che però il main non è collegato a Persona, quindi teoricamente il main non dovrebbe conoscere la classe Persona, quindi il modo corretto di implementare l'inserimento di una persona all'interno del gruppo è quello di creare un metodo addPersona a cui non si passa un'istanza di persona ma piuttosto si passano i parametri per crearne una.

Chiudiamo quindi ora questo discorso ed andiamo a vedere un po' di algoritmi della libreria Algorithm della STL.

Gli algoritmi di <algorithm> per i contenitori STL

Andiamo ad osserviare costrutti interessanti quali il for_each, questo è particolarmente utile, ma l'*auto* non è ben più furbo e veloce?

Sì lo è, ma il for_each è nato prima ed inoltre con il for_each si può delimitare il punto di partenza ed il punto di arrivo, mentre per l'auto non si può.

Un altro interessante caratteristica del for_each è che può essere usato per scorrere la lista in parallelo (su diversi thread).

Poi c'è la funzione iota, strana la iota.

Esercitazione 10

Continuiamo il discorso sulla libreria algorithms

La funzione copy

Questa può essere utilizzata per creare una copia di una porzione di contenitore, vengono passati come parametri i puntatori all'inizio e al termine dell'intervallo che voglio copiare + l'iteratore che punta all'inizio della destinazione dove voglio copiare la lista.

La funzione find

Permette di trovare un elemento (lo restituisce) in un intervallo di contenitore, il tipo dell'elemento deve avere l'operator == definito, altrimenti find non funzia.

Accumulate

Serve per sommare gli elementi di un intervallo di contenitore. La cosa interessante è che può essere usata anche per cose diverse dalla somma, basta che si passi una funzione esterna come argomento, ad esempio possiamo passare una funzione per fare il prodotto ed ottenere tutto il prodotto, tuttavia dobbiamo inserire come parametro anche il "valore iniziale" ovvero il valore iniziale dell'accumulo.

Funzioni di ordinamento

Sort

Questa funzione cambia il contenuto del vettore, non restituisce una copia ordinata. Tuttavia, per usare l'ordinamento il tipo di dato del contenitore deve aver l'operatore < implementato.

Shuffle

Riordina gli elementi in un intervallo di contenitore, si deve passare anche il generatore di numeri casuali per il disordinamento. Shuffle accetta solo dei generatori particolari e raffinati, la versione random_shuffle invece può accettare come generatore una nostra funzioncina del cavolo.

```
int random(int max){
    return rand()%max;
}

vector<int> v = {1,5,3,2,6,9,7,3};
sort(v.rbegin(), v.rend());
shuffle(v.begin(), v.end(), default_random_engine());
random_shuffle(v.begin(), v.end(), random);
```

random qui è la nostra funzioncina del cavolo, possiamo usarla solo con random_shuffle, in caso contrario si utilizza il default_random_engine().

Transform

Ci permette di compiere un'operazione su ogni elemento del nostro vettore oppure può compiere un'operazione binaria su un determinato intervallo, vediamo cosa intendiamo:

```
list<bool> lb(v.size());
transform(v.begin(), v.end(), lb.begin(), isEven);

for(auto& item : lb){
    cout << item << " ";
}
cout << endl;
```

In questo caso siamo andati ad usare transform per iterare lungo il vettore v, nell'intervallo v.begin()->v.end(), e lanciare per ogni elemento la funzione isEven (che prende un int in ingresso e ritorna true se è pari). Transform quindi va a popolare la lista con lb (a partire da lb.begin()) con i risultati di isEven per ogni elemento.

count_if

Count_if va a contare quanti elementi rispettano una certa condizione passata come parametro sotto forma di funzione che ritorna un bool.

find_if

Va a trovare il primo elemento che soddisfa una certa condizione passata come parametro sotto forma di funzione che ritorna un bool.

```
auto it = find_if(v.begin(), v.end(), isEven);
if(it != v.end() ){
    cout << *it << endl;
}
```

Occhio però che ritorna un iteratore, quindi:

1- Dobbiamo dereferenziarlo
2- Dobbiamo assicurarci che sia diverso da end!! (find_if ci ritorna end se non esiste un elemento che soddisfa la condizione richiesta)

Programmazione in multithreading (ahia!)

La libreria thread ci permette di separare dal processo main degli altri processi, una cosa da tenere bene a mente è che il tempo di esecuzione non è deterministico, quindi non si deve mai dare per scontato l'ordine di esecuzione.

Ai thread si possono passare 4 tipologie di parametri:

• Funzioni libere

• Funzioni membro

• Oggetti Functor

• Espressioni Lambda

Tuffiamoci in questo fantastico mondo.

Thread e funzioni

Proviamo a creare due thread molto semplici, che sostanzialmente contengono solo delle stampe di debug, tuttavia notiamo che le stampe di debug si sovrappongono a stecca e quindi è tutto un casino fra.

Posso passare come funzione, invece che una funzione *effettiva*, un metodo di una classe, ma in tal caso devo passare il metodo con lo Scoping della classe e poi devo passare un'istanza della classe tramite riferimento ed infine devo passare gli argomenti per il metodo.

```
Useless u;
thread t(Useless::metodo, &u, 5);
t.join();
cout << "fine" << endl;
```

Thread e oggetto Functor

L'oggetto functor serve per eseguire il metodo *operator()* di una certa classe, sempre all'interno del thread.

Sostanzialmente si scrive un metodo per l'operatore () nella classe:

```
void operator()(int v){
    cout << "operator chiamato
}
```

E poi si passa un'istanza della classe al thread, come se fosse una funzione:

```
thread tf(u, 5);
tf.join();
cout << "fine" << endl;
```

Thread e espressioni lambda

Le funzioni lambda sono abbastanza complicate, vedremo in seguito.

Le lambda sono costituite da tre parti:

- Parentesi [] per recuperare il valore di variabili che potrebbero servire alla funzione
- Parentesi () per recuperare i parametri della funzione
- Parentesi {} per scrivere il corpo della funzione

Thread e la memoria condivisa

Thread & Mutex

- Un **mutex** è un oggetto bloccabile progettato per segnalare quando sezioni critiche di codice necessitano di accesso esclusivo, impedendo ad altri thread con la stessa protezione di essere eseguiti contemporaneamente e di accedere alle stesse posizioni di memoria.
- Gli oggetti **mutex** forniscono proprietà esclusiva e non supportano la ricorsività (cioè, un thread non deve bloccare un mutex che già possiede) - vedi recursive_mutex per una classe alternativa che lo fa.

Okay, sostanzialmente il mutex viene usato per gestire l'accesso ad una zona di memoria che rischia di essere modificata allo stesso momento da due funzioni diverse che girano su thread diversi.

Il codice che si trova tra due comandi lock() ed unlock() può essere eseguito *solo* da un thread alla volta.

```
void stampa_bloccante(int n, char c){
    mx.lock();
    for(int i=0; i<n; i++){
        cout << c;
    }
    mx.unlock();
}
```

```
thread c(stampa_bloccante, 10, '+');
thread d(stampa_bloccante, 10, '*');
c.join();
d.join();
cout << "fine" << endl;
```

In questo esempio visto a lezione due thread che chiamano stampa_bloccante non vedranno mai le loro esecuzioni interrotte reciprocamente, mentre la versione di questa funzione senza mutex genera sempre stampe dall'ordine "colorito".

Esercitazione 11

Riprendiamo con l'esempio sull'esclusione in multithreading.

Abbiamo visto come il mutex implementi una sorta di gestione a semafori. Il problema è che questo rende possibile la situazione di stallo che si viene a creare se un processo che blocca un'area critica si impianta e non libera più tale area.

Andiamo a vedere ora come risolvere i problemi che si possono verificare quando due thread diversi agiscono su una stessa variabile. Per sistemare questo problema e rendere la variabile "atomica" useremo la libreria atomic.

Anche atomic è creata tramite template e quindi richiede la specificazione del tipo. Vediamo il nostro esempio:

```
atomic<long> somma_tot(0);

void somma_vettore(const vector<int>& v, int iniz, int fin){
    for(int i=iniz; i<fin; i++){
        somma_tot += v[i];
    }
}

int main(int argc, char** argv) {
    vector<int> v = {1,2,3,4,5,6,7,8,9};
    thread r(somma_vettore, v, 0, 3);
    thread s(somma_vettore, v, 3, 6);
    thread t(somma_vettore, v, 6, 9);
    r.join();
    s.join();
    t.join();
    cout << "somma_totale=" << somma_tot.load();

    return 0;
}
```

Lambda expressions

Le lambda expressions sono *funzioni create al volo* o anche *funzioni anonime*, vediamo la sintassi, gli utilizzi e le possibilità che ci offrono.

Derivano dal lambda calcolo, un formalismo matematico che hai studiato in passato; di solito vengono usate quando si deve passare una funzione come argomento per un'altra funzione.

Le funzioni lambda possono sia accettare parametri in ingresso che utilizzare variabili che sono dichiarate all'esterno della lambda stessa (si dice cattura di variabile).

La lambda è composta da 3 parti principali:

- Parentesi [] per passare il valore di variabili esterne alla lambda (che vengono passate in sola lettura)
- Parentesi () per passare i parametri formali della funzione
- Parentesi {} che contiene il corpo della funzione

Nel momento in cui mettiamo la clausola mutable le variabili passate nella clausola di cattura possono essere cambiate. Nota che la variabile catturata nella clausola di cattura è passata per copia (quindi in reltà nella lambda io comunque andrei a modificare solo la copia). Abbiamo quindi 3 possibilità di cattura:

- Per copia immutabile
- Per copia mutabile
- Per riferimento

Le varie possibilità per invocare una lambda sono queste:

```
1) [capture] (params) mutable -> ret {body}
2) [capture] (params) -> ret {body}
3) [capture] (params) {body}
4) [capture] {body}
```

Come vedi si possono addirittura omettere i parametri (ma non si fa perché è un casino capirsi poi).

Si può addirittura omettere il tipo di ritorno, in tal caso sarà nostra premura utilizzare un *auto* quando prendiamo il return dell'espressione nel codice.

Andiamo a vedere subito qualche esempio.

```
auto l1 = [a] () {
    return a*5;
};
cout << l1() << endl;
```

Qui abbiamo il primo esempio con passaggio per copia non modificabile.

```
auto l2 = [a] () mutable {
    ++a; return a*5;
};
cout << l2() << endl;
// Il valore della variabile a non cambia
// perchè la lambda lavora su una copia di a
// e non su a stessa
cout << a << endl;
```

Il secondo esempio è un passaggio per copia modificabile, dentro la lambda posso modificare la variabile ma fuori dalla lambda non si vedono cambiamenti.

```
auto l3 = [&a] () {
    ++a;
    return a*2;
};
cout << l3() << endl;
// Qui a cambia perchè abbiamo lavorato sulla
// reference ad a
cout << a << endl;
```

Il terzo esempio introduce il passaggio per riferimento, quindi posso modificare anche il valore esterno della variabile.

Presta attenzione! In fondo alle dichiarazioni di lambda scritte così come variabili nel main va messo il ";".

Esempio di tipizzazione:

```
auto somma = [] (int x, int y) -> int {
    return x+y;
};
cout << somma(-4, 8) << endl;
```

Un esempio molto più interessante è il seguente:

```
int mul = 4;
transform(v.begin(), v.end(), v.begin(),
    [mul] (int item) {
        return item*mul;
    }
);
```

Qui viene utilizzata la funzione transform per cambiare tutti gli elementi contenuti tra v.begin() e v.end() che vengono poi scritti nelle posizioni di memoria a partire da v.begin(); la lambda in questo caso viene usata come funzione per la modifica dei valori mentre transform vi itera sopra.

```
int soglia_del = 10;
v.erase( remove_if(v.begin(), v.end(),
        [soglia_del] (int item) -> bool {
            return item<soglia_del;
        }
        )
    , v.end()
);
```

Proviamo ora a fare una cosa simile con la funzione remove_if; questa funzione ci ritorna un iteratore che poi noi useremo per eliminare l'elemento dal vettore.

v.erease va ad eliminare dal vettore gli iteratori passati.

Class template e function template

Diciamo che vogliamo creare una funzione minimo, la creiamo per gli interi e quindi la dichiariamo come (int, int) -> int; mettiamo caso che poi ci salti in mente di creare una versione per i float: dobbiamo riscrivere la funzione?

Grazie ai template non dobbiamo!

```
template <typename T> T minimo(T a, T b){
    if(a<b){
        return a;
    }
    return b;
}
```

Con la sintassi template <…> creiamo un tipo per il template, poi possiamo creare la nostra funzione usando il placeholder che abbiamo scelto come tipizzatore.

La figata è che questa funzione minimo funziona in entrambi i casi riportati sotto:

```
int a = 5, b = 8;
cout << minimo(a,b) << endl;
float c = 3.24, d = 5.675;
cout << minimo(c, d) << endl;
```

In questo caso però c'è una particolarità che non notiamo perché è nascosta: entrambi i tipi *int* e *float* hanno l'operatore < definito nelle loro dichiarazioni. Se voglio usare una funzione template devo sempre assicurarmi che gli attributi e gli operatori del tipo "T" che usiamo all'interno della funzione template devono essere implementati per gli oggetti che mandiamo in input alla funzione template stessa.

Esercitazione 12

Classi template

La volta scorsa abbiamo implementato una funzione template, quindi sappiamo che la difficoltà stava nell'implementare gli operatori utilizzati per tutte le istanze possibili come input.

Oggi andiamo a creare una classe che contiene un parametro che viene inserito a runtime e tipizzato anche a runtime.

Creiamo una classe che ci permetta di utilizzare array di tipo generico in modo un po' furbo. Con più furbo intendiamo che l'accesso controlla i limiti di grandezza dell'array.

Andiamo a creare un metodo init per non inizializzare ogni volta l'array: se decidiamo di utilizzarlo direttamente all'interno dei costruttori allora la cosa corretta da fare è anche spostarlo nella parte private della classe, perché nessuno avrà bisogno di usarlo fuori dal costruttore.

A questo punto dobbiamo andare a fare l'override dell'operatore [] per l'accesso agli elementi dell'array.

Dentro a questo override andiamo ad insanziare anche dei controlli sui limiti di indicizzazione dell'array!

A questo punto dobbiamo anche risolvere il fatto che non possiamo usare la sintassi *dati[0] = 5*, questo perché dati[0] ci ritorna un intero che è un r-value, e non possiamo assegnare un r-value ad un altro r-value, quindi dobbiamo fare n modo di ritornare un riferimento ad intero invece che un intero.

A questo punto passiamo dal tipo intero ad un tipo generico.

La classe di tipo generico va dichiarata così:

```
template <class T> class Myarrayt{
    private:
        int dim;
        T* m;
    public:
        Myarrayt(int d){
            dim = d;
            m = new T[dim];
        }
};
```

T potrebbe essere sostituito da un qualsiasi altro nome.

La classe va dichiarata con la sintassi <class tipoGen> class … {…}

La classe, quando viene istanziata ha bisogno che venga passato il tipo che equivale a T, quindi viene istanziata in questo modo:

```
Myarrayt<double> arr(8);
```

Nota: è molto meglio implementare tutti i metodi all'interno delle classi template, perché la sintassi al difuori diventa molto pesante.

Nota2: scrivi tutti gli attributi di una classe template nello stesso file: non dividere mai in .h e .cpp.

Per implementare l'operatore [] non abbiamo troppi problemi, facciamo la stessa cosa fatta per l'array di interi.

Abbiamo qualche problemino con la stampa…

Per definire l'operatore << di una class template dobbiamo implementare il metodo esternamente (e questo lo sapevamo già), definendo di nuovo il tipo template perché altrimenti non sappiamo che tipo stampare, inserire il metodo come friend della classe (classico), inserire un <> prima degli argomenti del metodo per specificare che usa un template, inserire l'header del metodo prima della classe per far capire alla classe che metodo è ed

inserire sopra a questo header un ulteriore header della classe, perché altrimenti l'header del metodo non sa di che tipo è il secondo argomento. Purtroppo, non è possibile fare altrimenti, se vogliamo implementare il metodo di stampa tramite << dobbiamo fare per forza così.

```
template <class T> class Myarrayt;
template <typename T> ostream& operator <<(ostream& os, const Myarrayt<T>& a);
template <class T> class Myarrayt{
    private:
        int dim;
        T* m;
    public:
        Myarrayt(){…
        Myarrayt(int d){…
        ~Myarrayt(){…
        T& operator [] (const int index){…
        friend ostream& operator << <> (ostream& os, const Myarrayt<T>& a);
};
template <typename T> ostream& operator <<(ostream& os, const Myarrayt<T>& a){
    for(int i=0; i<a.dim; ++i){
        os << "[" << a.m[i] << "]";
    }
    return os;
}
```

Okay, ora che abbiamo completato la classe template ci chiediamo: la possiamo usare con tutte le classi?

No, la possiamo usare con tutte le classi che soddisfano il requisito di avere i seguenti metodi:

- Costruttore a 0 parametri (per l'inizializzazione dell'array)
- Operatore di stampa << per la stampa

Fine. Quindi insomma quello che dobbiamo implementare sono tutti i metodi del template che richiamiamo, esplicitamente o meno, all'interno dei metodi della classe template.

Una differenza semantica tra classi template e funzioni template è il modo in cui si inizializzano i tipi template:

- template <typename T> per usare il template T nelle funzioni
- template <class T> per usare il template T nelle classi

rvalues, lvalues e costruttore di spostamento

Cos'è un lvalue? Un valore che sta a sinistra dell'uguale, mentre il rvalue sta a destra, ez no?

La differenza sostanziale è che tutti i lvalue hanno un indirizzo di memoria specifico, gli rvalue no. Il C++ ci dice appunto che tutto quello che sta a sinistra dell'uguale dev'essere un lvalue, deve poter essere riferito.

```
int x = 10;
int* y = &x;  // Ok, si può.
int* z = &10; // non si può!
```

Non si può passare un riferimento ad un rvalue.

Non permesso:

```
int setValue() { return 6; }
//…
setValue() = 3; // ERRORE
```

Permesso:

```
int global = 100;
int& setGlobal(){
  return global;
}
//…
setGlobal() = 400; // OK
```

Parliamo ora di conversioni: è possibile convertire un lvalue in un rvalue? Certo! È implicita e l'abbiamo sempre fatto.

In C++ invece non posso convertire un rvalue in lvalue.

Come possiamo bypassate questa cosa?

Non possiamo fare in questo modo: int& yref = 222;

non possiamo perché 222 è un valore temporaneo e viene distrutto subito dopo l'esecuzione dell'istruzione, tuttavia il modo per bypassare il problema di trasduzione è simile a questo meccanismo.

Supponiamo di avere una funzione che prende come input un riferimento ad intero:

```
void miaFunz(int& x) { /* … */ }

int main() {
  miaFunz(99); // ERRORE
  // codice corretto
  // int x = 10;
  // miaFunz(x);
}
```

Ad esempio, qui si vede come passare un intero genera un errore, perché appunto gli rvalues non sono referenziabili.

Per questo problema è stata introdotta la possibilità di associare un lvalue ad un rvalue costante.

```
void miaFunz2(const int& x) { /* … */ }
int main() {
  miaFunz2(99); // OK
}
```

Quindi è ammesso il passaggio di rvalues per riferimento, ma appunto sono costanti e non modificabili. Questa è l'unica modalità disponibile.

Nell'esempio precedente, se ricordi, abbiamo creato una persona nel main e l'abbiamo passata per copia all'array template che se l'è copiata nel suo array interno: questa copia è pesante, come possiamo eliminarla? Proprio grazie a giochetti di lvalue e rvalue!

Il mezzo che usiamo in questo caso è il *move constructor* che si occupa di costruire un oggetto per "spostamento", ovvero non copiare l'oggetto ma prendere direttamente i riferimenti alla variabile creata come valore temporaneo (nel nostro esempio la persona creata nel main era un valore temporaneo che abbiamo istanziato solo per passarlo all'array).

Una referenza a rvalue si identifica per il costrutto <tipo>&& (doppia reference). Un rvalue reference tecnicamente è dereferenziabile, quindi può essere modificata, quindi può comparire a sinistra dell'operatore di assegnamento => è un lvalue! Fatto sta però che è temporaneo.

Questa semantica viene utilizzata nel costruttore di spostamento.

Se vogliamo usare in modo esplicito il move nel nostro codice dobbiamo includere la libreria <utility> (meglio includerla, alcuni compilatori sono stronzi).

Così si implementa il move constructor per una classe con array di interi *data* e dimensione *dim*:

```
Myclass(Myclass&& other) {
	data = other.data;
	dim = other.dim;
	//other.data = nullptr;
	//other.dim = 0;
	cout << "costruttore spostamento" << endl;
}
```

Si noti che si va a rendere nulli i riferimenti del temporaneo perché la distruzione di questo non li invalidi.

Allo stesso modo possiamo creare l'operatore di assegnazione =

Che però dobbiamo chiamare esplicitamente, non si sa mai:

```
Myclass h1(10000); // costruttore specifico
Myclass h6( move(h1) );    // costruttore spostamento
Myclass& operator=(Myclass&& other) {
	delete[] data;
	data = other.data;
	dim = other.dim;
	other.data = nullptr;
	other.dim = 0;
	cout << "oper = per spostamento dim("<<dim<<")" << endl;
	return *this;
}
```