LEZIONE 2

2.1 Variabili

Oggetto: Regione di memoria che contiene un VALORE di un determinato TIPO
Variabile: Named object, oggetto che ha un nome
Valore: Sequenza di bit interpretati in funzione del tipo
Tipo: Regole dell'oggetto, definisce: RANGE valore, TIPOLOGIA contenuto, OPERAZIONI.

Type Safety: Riguarda le operazioni di conversione di tipo

Type safe: Passare ad un tipo con RANGE MAGGIORE,
Il compilatore effettua la conversione

Non Type safe: Passare ad un tipo con RANGE MINORE (narrowing conversion),
Il compilatore NON SEGNALA ERRORI

Notazione {}: Fa in modo che il compilatore SEGNALI le narrowing conversion
Si chiama universal and uniform initialization

```
double d = 2.5;
int i = 2;

double d2 = d/i; // OK d2 = 1.25
int i2 = d/i // OK i2 = 1
int i3 = {d/i} // ERRORE, la divisione ritorna un double che si cerca di mettere in un int
```

2.2 Astrazione, Espressioni, Istruzioni, Dichiarazioni, Definizioni

Astrazione: Nascondere i dettagli implementativi di una classe, funzione, ecc

Interfaccia: Definisce come usare una classe, funzione ecc

Espressione: Unità più elementare di computazione
Calcola un risultato di OPERANDI
Le più semplici sono LITERAL

LValue: Ciò che sta a sinistra dell'operatore =, (), {} **RValue** sta a destra di =,(),{}

Inizializzazione: =, (), {} si usa su una variabile NUOVA
Assegnamento: =,(), {} si usa su una variabile GIA' ESISTENTE

Espressioni Costanti: Costanti SIMBOLICHE, evitano i MAGIC NUMBERS (danno un nome a dei numeri)

Constexpr: Note a tempo di compilazione
Const: Note a tempo di esecuzione, si usa il valore dell'inizializzazione

Statement: Specificano AZIONI

Expression statement: Risultati di espressioni

Selezioni: if

Iterazioni: while, for

Dichiarazioni e Definizioni

Dichiarazioni: Introducono un NOME in uno SCOPE, quindi si riferiscono a variabili (che sono oggetti), funzioni e classi

Specificano NOME e TIPO, solo a volte anche il valore di inizializzazione (se è una var)

Quindi NON SEMPRE HANNO EFFETTO SULLA MEMORIA

Interfaccia, importante per il <u>compilatore</u>

Definizioni: Dichiarazioni che SPECIFICANO INTERAMENTE L'ENTITA' DICHIARATA

Quindi specificano NOME, TIPO e RISERVANO MEMORIA PER L'ENTITA'

Variabile: nome, tipo, valore

Funzione: signature (dichiarazione), corpo della funzione

Classe: nome (dichiarazione), membri della classe

Implementazione, importante per il <u>linker</u>

LEZIONE 3

3.1 Scope

Scope: Regione, blocco di codice

Globale: Scope al di fuori di ogni altro scope

Namespace: Scope a cui diamo un nome

Classe: Codice di una classe

Locale: Blocco {}, oppure argomenti di una funzione

Statement: ad esempio for(int i = 0; i<5; i++)

Possono esserci scope ANNIDATI

Posso creare CLASSI DENTRO FUNZIONI ma NON FUNZIONI DENTRO FUNZIONI

```
// x locale ad f, z locale ad f, f appartiene allo scope GLOBALE
void f(int x) {
    int z = x+ 7;
}
```

Strumenti di scoping: Permettono di raggruppare classi, funzioni e variabili negli scope

Namespace: Costrutto specifico per scoping

Si dà un nome ad uno scope che raggruppa classi, funz, var

Permettono di avere classi, funz, var con lo STESSO NOME ma appartenenti a NAMESPACE DIVERSI

Infatti si scrive Namespace::membro

Using namespace: Utile per non appesantire il codice
Possiamo accedere ai nomi di un namespace come se tali nomi fossero dichiarati al di fuori di esso
Può causare NAME CLASH

```
namespace Graph_lib {
    class Text {...};
}

namespace Text_lib {
    class Text {...};
}

using namespace Graph_lib;
using namespace Text_lib;

Text obj; // Errore! Il compilatore non sa quale quale text prendere
```

3.2 Funzioni e reference

Funzioni: dividono il codice in blocchi di computazione

Parametri e argomenti:

```
int d = func(x,y);
int func(int a, int b) { return a+b; }
```

x ed y sono ARGOMENTI. Se il passaggio è per valore, essi sono RVALUE
a e b sono PARAMETRI o ARG FORMALI, inizializz. al valore degli argomenti

Passaggio per valore o per riferimento: Senza reference = per valore, con reference = riferimento

Reference: Label che identificano VARIABILI GIA' ESISTENTI
NON INDIRIZZI, sono solo dei NOMI
Permettono accesso in LETTURA E SCRITTURA della variabile a cui fanno riferimento
Una reference non può riferirsi ad altre varabili se non a quella dell'inizializzazione
Protezione: const reference impedisce la SCRITTURA, permette SOLO LETTURA
La verifica viene fatta dal compilatore
Utilizzi: Per oggetti GRANDI, const reference se non bisogna modificarli
Per oggetti piccoli si fa il passaggio per valore

```
int i = 7;
int &j = i; // i e j equivalgono sempre nel loro scope

voig g(int a, int& ref, const int &const_ref) {...}

g(1,2,x); // ERRORE, una reference ha bisogno di un oggetto
g(1,x,3); // OK, una const reference può avere un LITERAL
```

Conversione degli argomenti: Fare un passaggio per valore significa inizializzare i parametri al valore degli argomenti. Quindi il tipo degli argomenti deve essere convertito

Conv. implicita: Fatta dal compilatore se è TYPE SAFE

Conv. esplicita: Per le NARROWING CONVERSIONS

Si usa static_cast<tipo_range_minore>(var);

Oppure tipo_range_minore(var_range_maggiore);

3.3 Librerie e Linking

File Header: Insieme di DICHIARAZIONI (quindi anche definizioni) di entità

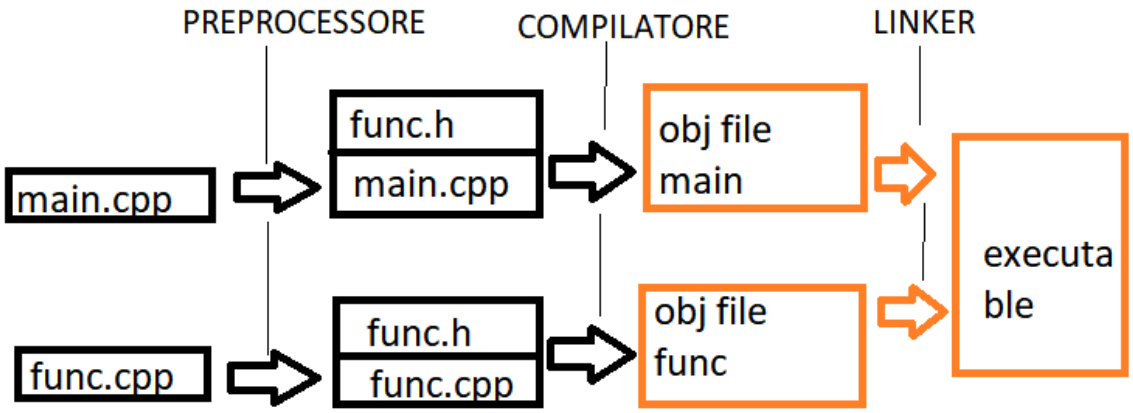
#include <nomefile> se l'header è già compilato, libreria DI SISTEMA

#include "nomefile.h" se deve ancora essere compilato

Divisione del codice: HEADER .h: dichiarazioni di funzioni e classi

SOURCE .cpp: definizione di tutto ciò che è dichiarato nell'header

Compilazione con linking statico:



Preprocessore: Copia il codice presente nei .h e lo mette all'inizio dei .cpp

Per evitare duplicazioni di codice dobbiamo usare le INCLUDE GUARDS

Compilatore: Genera un file oggetto in codice MACCHINA a partire dal file che gli viene dato in input dal preprocessore

Linker: Rende disponibili al file oggetto main le DEFINIZIONI di ciò che è DICHIARATO in func.h

La compilazione è statica dato che la libreria func viene ASSOCIATA a main.cpp dal preprocessore.
Se si mette #include <iostream>, la compilazione è sempre statica, il preprocessore accoda l'HEADER iostream che non ha estensione dato che è una libreria di sistema. Il compilatore crea un obj file ed il linker crea l'eseguibile = obj file main + libreria completa pre-compilata.

Problema: Se ho N file sorgenti, la libreria viene REPLICATA N volte negli N eseguibili

Compilazione con linking dinamico: Meccanismo con cui se ho N sorgenti, l'header di una libreria viene accodato N volte dal preproc., ma il LINKER associa gli N file oggetto ad una UNICA LIBRERIA precompilata

Statico vs Dinamico: STATICO: Libreria replicata per ogni eseguibile in cui viene incluso il suo header
Se modifico libreria devo ricompilare anche gli exe in cui è replicata
Utile per progetti self-sustained
DINAMICO: Libreria esiste una sola volta nel sistema, associata a qualsiasi obj file
Posso ricompilarla senza toccare gli altri eseguibili
Occupazione totale di memoria ridotta rispetto al linking dinamico

LEZIONE 4

4.1 Side Effect, Valutazione delle Espressioni

Side Effect: Quando un operatore fornisce un risultato ma modifica anche uno dei suoi operandi
Operatori ++, -=, –, -= ecc

Ordine di valutazione delle espressioni:

1. Lettura degli operandi, NON SI SA IN CHE ORDINE
2. Operazioni in ordine "matematico", ovvero si rispettano le parentesi
3. Assegnamento del risultato

```
int p = (width + length)*2; // Non si sa se viene letta prima width oppure length
p = width; // Non si sa quale delle due viene letta per prima
```

Libertà del <u>compilatore</u>: Quando l'ordine di lettura non è definito e l'operatore presenta side effect, il <u>compilatore</u> ha liberta sull'ordine di valutazione.
Eventuali azioni illegali <u>NON SONO SEGNALATE necess. dal compilatore</u>
NON USARE DUE VOLTE UNA VARIABILE COINVOLTA IN UN OPERATORE CHE HA SIDE EFFECT

```
v[i] = i++; // Non si sa cosa farà il compilatore, può leggere i ed incrementarla o leggere prima v[i]
v[++i] = i; // Non si sa cosa farà il compilatore
```

4.2 Layout Memoria, Chiamate a funzione, Lifecycle

Chiamata a funzione: Creato un Record di Attivazione nello stackframe quando una funz viene chiamata
Ogni RA contiene copia dei parametri, var locali e ciò che serve per il return
Quando una funzione termina, il suo RA viene poppato dallo stack

Layout di memoria:

Text Segment: Istruzioni effettive eseguibili in codice macchina
Read Only, copiato dal file oggetto

Initialized Data Segment: Variabili globali e statiche e costexpr
Quindi è read e write oppure read only

Uninitialized Data Segment: Inizializzato a 0 dal sistema operativo
Variabili globali e statiche non inizializzate nella dichiarazione o ==0

Stack: Contiene gli RA. Quindi chiamate a funzione e variabili locali (il main è una funz)

Heap: Costituito da blocchi di memoria non contigui
Gestito tramite NEW e DELETE, modificabile da funzioni di sistema

Lifecycle di variabili:

Globali: Scope globale, inizializzate prima della prima ist. del main, a 0 di default.
Sono statiche.

Statiche: Accessibili solo nello scope in cui sono dichiarate, quindi possono essere globali.
Restano in memoria anche se sono al di fuori del loro scope, mantenendo il loro valore
Rilasciate al termine del processo

Locali automatiche: Scope locale in cui sono dichiarate (ricorda che anche un ciclo è uno scope)
Rilasciate in AUTOMATICO, ordine inverso rispetto a quello di dichiarazione

Translation unit: Singolo file oggetto che è in codice macchina

Variabili globali in TU diverse:

```
// CODICE DA EVITARE: x1, y1, y2 sono GLOBALI, ma non sappiamo se vengono prese prima quelle di f1.cpp o f2.cpp. Quest'ultimo vedrebbe y1 non inizializzata se preso per primo

// f1.cpp
int x1 = 1;
int y1 = x1 + 2;

// f2.cpp
extern int y1;
int y2 = y1 + 2;
```

4.3 Intro agli UDT

UDT: Tipi non built-in, definiti dai programmatori. (STL, Classi, Struct)

Rappresentazione: Sapere come rappresentare i dati di un oggetto, per saperne lo STATO
Operazioni: Sapere cosa si può fare con i dati (lo stato) di un oggetto

Strumenti per UDT: **Classi:** Rappresentazione diretta di UDT
Come creare, usare, distruggere un oggetto

Struct: Composte da tipi build-in, altri udt, funzioni = MEMBRI della classe
Classi i cui membri sono public di DEFAULT
Solitamente usate per rappresentare un insieme di dati, no funz

Interfaccia di una classe: Come usare la classe
Tutto ciò che è public, ovvero utilizzabile dall'utente che non ne conosce i dettagli implementativi

Implementazione di una classe: Tutto ciò che è private, gestito dall'implementatore
Accessibilità solo tramite l'interfaccia

LEZIONE 5

5.1 e 5.2 Progettazione di una classe

Helper function: Non sono dichiarate nello scope della classe
Lavorano a stretto contatto con gli oggetti della classe di cui sono helper
Ad esempio overloading di alcuni operatori

Funzione inline: Il compilatore può (ma non è detto) riscrivere il codice di una funzione inline ogni volta che viene chiamata, senza implementare una vera e propria chiamata usando lo stackframe. Controlla che non ci siano cicli o ricorsioni prima di farlo
Molto utile per le funzioni brevi, riduce overhead ovvero il costo

Member function: Dichiarate nello scope della classe, definite dentro o fuori dall'interfaccia
Definizione in-class: Implementazione si mescola con interfaccia
Tali funzioni diventano inline
Definizione fuori: return_value Classe::nome_funct(params) {...}
Const funct: F membro che non modifica lo stato dell'oggetto
Su un oggetto const posso essere usate solo funzioni membro const, altrimenti errore in compilazione

Invarianti: Regole che determinano la VALIDITA' dello STATO di un oggetto

Eccezioni: Separare rilevazione di errori dalla loro gestione
Funzione chiamata: Rileva errore, lancia eccezione
Funzione chiamante: Riceve eccezione, gestisce l'errore

Enumeratori: UDT che associa un nome a ciascun numero di una lista
Enum class: Crea uno scope, quindi l'accesso ai suoi membri avviene con Nome_enum::nome.
Enum senza class: Non viene creato uno scope, acesso libero ai suoi membri

```
// accesso del tipo Month m = Month::jan;
enum class Month { jan = 1,..., dec };

// accesso del tipo Month m = jan;
enum Month { jan = 1,..., dec };
```

Protezione con enum class: Se il membro di una classe è di tipo UDT Enum Class, allora per inizializzarlo è necessario usare un oggetto dello stesso tipo di enum class, NON vale il rispettivo int, errore in compilazione

Init list: Inizializza i membri usando i rispettivi costruttori che come argomenti usano i parametri passati
Evita valori di default e fa un solo passaggio

```
// i membri player1 e player2 vengono direttamente definiti usando il costr. di copia
Chessboard::Chessboard(Player p1, Player p2)
    : player1{p1}, player2{p2} {}

// i membri player1 e player2 prima vengono inizializzati con il costruttore
// di default (ma noi non lo vediamo), poi si usa assegnamento di copia
Chessboard::Chessboard(Player p1, Player p2) {
    player1 = p1;
    player2 = p2;
}
```

Costruttore di default: Ne esiste uno per ogni tipo, sia UDT sia built-in
Per i tipo build-in, inizializza tutto a 0 (equivale a false per bool)

```
// Chiamate EQUIVALENTI, x = 0
int x; // è sempre una chiamata a costruttore di default, anche per UDT
int x{};
int x = int{};
std::string s = std::string{}; // costr di default, NON E' UN ASSEGNAMENTO DI COPIA
```

In class initializers: Valori di default per membri di una classe

```
class A {
    public: A(int y); // sovrascrive costr. di default
    private: int x = 10;
};

A a{}; // ERRORE in compilazione, comunque non esiste più un costr.default
```

5.3 Overloading operatori generale

Regole generali: Deve esserci almeno un parametro UDT
Alcuni solo helper, altri solo member, altri possono essere entrambi, cambia firma

`++d ⟷ operator++(d) // chiamato T operator++(T& d, int); dove int è dummy`

I Più comuni come helper: +, -, *, / Ognuno ha due argomenti di cui uno è UDT, ritorna tale UDT

`<<` Un arg è ostream&, ritorna ostream&

`>>` Un arg è istream&, ritorna istream&

<< e >> devono ritornare reference, altrimenti ritornano un nuovo stream che creerebbe confusione con la risorsa condivisa

LEZIONE 6

6.1 Puntatori

Puntatore: Tipo di variabile
Permette alla variabile di contenere un indirizzo di memoria
Una variabile puntatore punta al Tipo ed anche al valore contenuto in memoria
Quindi va inizializzato con un indirizzo di memoria, ottenibile con & (diverso da ref)

Dereference: Operatore * fornisce un LVALUE

```
*p = 27 // mette 27 nella cella il cui indirizzo è il valore di p
*p = &x // mette l'INDIRIZZO di x nella cella il cui indirizzo è il valore di p
```

Tipi di ptr: Il tipo del puntatore definisce il tipo di dato a cui può puntare

```
char ch = 'c';
char *pc = &c; // puntatore a char
int *pi = pc; // errore, int pointer può puntare solo a int
```

6.2 Casistiche Puntatori

Void ptr: Puntatore a memoria RAW, è un vero e proprio indirizzo di memoria
Gli si può assegnare un qualsiasi puntatore, il viceversa solo con cast esplicito
Non vengono fatti controlli sui tipi
Non permette la scrittura in memoria

```
void *pv = new int;
double *pd = pv; // ERRORE
double d = *pv; // ERRORE, non so a cosa punta pv
int *pi = static_cast<int*>(pv) // Ok

void *pv = new double;
int x= static_cast<int>(*static_cast<int*>(pv)); // Ok
```

	Ptr	Ref
Assegnamento:	Cambia indirizzo a cui punta	Cambia valore di ciò a cui si riferisce
	Indirizzo acquisito con new e &, posso spostarlo	Si riferisce sempre alla stessa variabile
	Accesso con * e []	Accesso come una var normale
	Shallow copy, la copia sposta solo il puntatore	Deep copy, la copia modifica variabile a cui si rif

Casistiche nel passaggio di parametri:

```
int incr(int x) { return x+1 };
void incr_with_ptr(int *p) { ++(*p) };
void inc_with_ref(int &ref) { ++ref };

int n; // costr. default
n = incr(n); // molto chiaro
incr_with_ptr(&n); // esplicito
incr_with_ref(n); // da qui non si sa se modifica n oppure no
```

1. Più chiara dato che c'è il tipo di return, OK per oggetti piccoli o grandi con move constr
2. Va bene per tipi build-in, controllo errori grazie a nullptr
3. Va bene per oggetti grandi, se si vuole aggiungere protezione usare const&

6.3 Array in C++

Stile C: La dimensione è un intero const oppure literal, nota a tempo di compilazione
Sono particolari variabili locali o globali (quindi stanno nello stack o in data segment, no heap)
Quando passati come argomenti di funzione, diventano puntatori ai loro primi elementi

```
double ad[10]; // locale, è nello stack
double *p = &ad[5] // punta al sesto elemento
p[2] = 9 // l'ottavo elemento ha ora valore = 9
```

Stringa stile C: Particolare array stile C che termina con il char '\0'

Const-ness nei ptr:
- **Puntatore costante:** Non non posso farlo puntare ad altro
  Sintassi: `int *const p`
- **Puntatore a costante:** Non posso modificare ciò a cui punta
  Sintassi: `const int *p` oppure `int const *p`

Regola della spirale

Decadimento array: Un array è un puntatore const al suo primo elemento
E' quindi un RVALUE, non posso modificarlo
Non posso usarlo per la copia
Se passato come argomento di funzione, in realtà si passa il ptr al suo primo element, perdendo quindi informazione sulla sua dimensione

Inizializzazione:

```
char ac[] = "Beorn"; // il compilatore aggiunge '\0' quando rvalue è literal
char *pc = "Howdy"; //  il compilatore aggiunge '\0'
char chs[] = {'a', 'b', 'c'}; // non è una stringa stile C
int ai[100] = {1,2,3}; // altri elem = 0
```

Aritmetica puntatori: E' supportata la somma e la sottrazione SOLO TRA PTR ED INTERO
Problemi: Accesso a puntatore non inizializzato o a nullptr. COMPILA

```
int *p; // non si sa cosa sia il default
*p = 9;
```

Accesso fuori dai limiti di un array. COMPILA
Accesso a oggetto uscito dallo scope/ distrutto. WARNING ma COMPILA

LEZIONE 7

7.1 Memoria Dinamica

Heap: è il FREE STORE, allocato dal sistema operativo quando manda in exe il programma

New: Permette di accedere al Free Store
Ritorna sempre un puntatore ad un'area di memoria che non ha nome

Casistiche:

```
double *p = new double[n]; // dimensione nota a runtime
double x = p[100]; // nessun controllo anche se n <= 100
double *p; // non si sa cosa sia
double *p = new double; // punta ad un double, ma il double NON è iniz. a 0
double *p = new double{5}; // Ok
double *p = new double[100]; // punta ad un array i cui double NON sono iniz. a 0
double *p = new double[5] {5,4,3,2,1}; // Ok, chiama il costruttore di ciascun p[i]
double *m = new double[5];
p = m; // RESTA GARBAGE = MEMORY LEAK
Piece p = new Piece[10] // ERRORE SE NON ESISTE COSTR. DI DEFAULT
```

7.2 Delete e Destructor

Delete: Permette di liberare la memoria del free store a cui punta un puntatore
Dopo un delete il free store può far occupare di nuovo quella memoria da altri oggetti
Se chiamato su un nullptr non è un errore, non viene liberato nulla

Manualmente: Perchè il garbage collector richiede risorse
Possiamo liberare memoria appena non ci serve più

Come fare e non: Ad ogni new deve corrispondere un opportuno delete

Memory leak:

```
double* calc(int size, int max) {
    double* p = new double[max]; // MEM LEAK, fare il delete di p
    double* res = new double[size];
    return res;
}
double* r = calc(100, 100); // MEM LEAK se non si fa il delete di r
```

Dangling pointer:

```
double[]* f(int size) {
        return new double[size]{1,2,3} // il resto sono 0
}
double *d = f(100);
delete[] d;
delete[] d; // DANGLING PTR. d mantiene il suo addr dopo il delete ed il free
            // store potrebbe aver occupato l'area in tale indirizzo
            // soluzione: PORRE d= nullptr; DOPO IL PRIMO DELETE
```

Distruttore: Se un UDT necessita di risorse nello Heap, esse sono:

Acquisite nel costruttore con un new

Liberate nel distruttore con un delete

Il distruttore VIENE CHIAMATO IN AUTOMATICO al termine dello scope di un oggetto (il delete NO)

Il distruttore di DEFAULT chiama il DISTRUTTORE (non DELETE) di ogni membro e quello della classe BASE

LEZIONE 8

8.1 Inizializzazioni di oggetti

Oggetto std::initializer_list<T>: Permette di inizializzare un vector come v = {1,2,3,4,9};
Lista iterabile che viene passata per VALORE al costruttore

Casistiche:

```
vector v1 {3}; // vector di 1 elemento il cui valore è 3
vector v2 (3); // vector di dimensione 3
vector v1 = {3} // più esplicito
vector v2 = {1,2,3} // più esplicito
```

8.2 Copia di oggetti

Problema: Da questo nasce la necessità del costruttore di copia

```
void f(int n) {
        vector v = {1,2,3};
        vector v2 = v;
        // distruttori chiamati automaticamente
}
```

Di default, la copia avviene membro a membro, quindi il puntatore di v2 punta allo stesso indirizzo di quello che punta a v. Dunque se v viene distrutto prima di v2 (il distruttore fa delete[], ma non di default), v2 ha un puntatore ad un'area rilasciata

Costruttore di copia: vector::vector(const vector& v) : sz{v.sz}, elem{new double[arg.sz]} {
copy(arg.elem, arg.elem+sz, elem);
}
Definiamo noi come avviene la copia
Posso accedere ai membri di v anche se sono private, lo scope è lo stesso!
vector v2 = v;
vector v2 {v}; // equivalente

Assegnamento di copia: Anche per operator= di default avviene la copia membro a membro
Overloading operator=: Obbligatorio funz membro
vector& vector::operator=(const vector& v) {
double *p = new double[v.sz];
copy(v.elem, v.elem+v.sz, p);
delete[] elem;
elem = p;
sz = v.sz;
return *this; // per concatenare v1 = v2 = v3 (prima v2 = v3)
}
Posso accedere ai membri di v anche se sono private, lo scope è lo stesso!

Shallow copy: Copia superficiale, riguarda solo il copiare indirizzi e non interi oggetti
Spesso è pericolosa (vedi origin story del copy constructor)
Un tipo così definito ha REFERENCE SEMANTICS

Deep copy: Copia "profonda", si ottiene implementando OPPORTUNAMENTE copy constructor e copy assignment
Un tipo così definito ha VALUE SEMANTICS

8.3 Spostamento di oggetti

Problema: T obj = funct(args);
Funct crea un oggetto di tipo T in O(n) e lo ritorna per poi assegnarlo a obj con costruttore di copia che se ridefinito, nel caso di vector, fa la copia di ogni elemento
Quindi vengono creati due oggetti di tipo T in O(2n) per usarne solo 1

Move semantics: Tipo in cui vengono implementati opportunamente move constr e move assignment

RValue reference: Parametro passato per riferimento che DEVE essere modificato dalla funzione
Si indica con T&& arg

Move constructor:

```
vector::vector(vector&& a)
    : sz{a.sz}, elem{a.elem} {
    a.sz = 0;
    a.elem = nullptr;
}
```

ATTENZIONE: SE NON USASSI LA INITLIST COSI' MA FACESSI UN copy(a.elem, a.elem+sz, elem) NON SAREBBE PIU' UNA OTTIMIZZ. IL SUCCO E' CHE CON elem{a.elem} RISPARMIO LA COPIA DI QUALCOSA CHE POI VIENE NULLIFICATO

Move assignment:

```
vector& vector::operator=(vector&& a) {
    delete[] elem;
    elem = a.elem;
    a = nullptr;
    sz = a.sz;
    a.sz = 0;
    return *this;
}
```

Copy elision: Ottimizzazione secondo cui il compilatore si rifiuta di chiamare le move semantics
Il compilatore fa in modo di costruire l'oggetto direttamente nella funz chiamante ovvero quello che si crea in funz(args) diventa direttamente obj, senza copy construct
Possiamo disabilitare questa funzionalità da CLI con no_elide_constructors

LEZIONE 9

9.1 Overloading operator[]

Vector non const:

```
double& vector::operator[](int n) {
    return elem[n];
}
```

Ritorna una reference così si può accedere sia in read che in write
Un vector const non può usarlo dato che la funzione non è const

Vector const:

```
double vector::operator[] const (int n) {
    return elem[n];
}
```

Problema nel primo caso:

```
vector v {1,2,3,4,5};
double& d = v[0];
d = 5; // modifica anche v[0]
```

9.2 Regole di progettazione

Distruttore: Solo se l'UDT necessita di risorse nell'HEAP

Move semantics: Solo per UDT che usano tipi molto grandi

Converting Constructor: Costruttore che riceve un solo parametro

Conversione implicita: vector v = 10 // vettore di 10 elementi
v = 20 // crea vec di 20 elementi e lo assegna a v
f(10) // void f(vector v), crea un vec di 10, lo passa

Costruttore explicit: Converting constructor con la keyword explicit
explicit vector(int n)
vector v = vector(10) // ok
vector v = 10 // errore in compilazione

Gestire tanti file: Includere solo i .h dentro ai file.cpp dato che contengono dichiarazioni necessarie per la compilazione. Dopodichè il linker associa le definizioni
Include guards dentro ai file .h
MAI include i file .cpp, bisogna includere solo ciò che è necessario per la compilaz ed i cpp contengono anche definizioni di funzioni e classi

LEZIONE 10

10.1 Template

Polimorfismo: Abilità di associare comportamenti specifici diversi ad un'unica notazione
Dinamico se avviene a runtime = ereditarietà
Statico se avviene in compilazione = template

Prog generica: Scrivere codice che funziona con tipi diversi che hanno stessi requisiti sintattici e semant

Template: Meccanismo per implementare il concetto di prog. generica in tempo di compilazione
Permette di usare usare un TIPO come un PARAMETRO
Function template o Class template

Specializzazione: Il compilatore genera il codice per ogni specializzazione del template

```
template<typename T>
void funct(T arg) { std::cout << arg };

// vengono create due funzioni diverse a tempo di compilazione
int i; char c;
funct<char>(c);
funct<int>(i);
funct(c); // deduzione automatica fatta dal compilatore
```

I costruttori di una classe template devono essere explicit, infatti se dovessimo fare una conversione implicita non avremmo modo di specificare il tipo finale

Parametri template: Possono essere anche valori numerici, ma devono essere noti a tempo di comp quindi literal oppure const o constexpr
I vantaggi sono una maggiore performance dato che il comp riesce ad ottimizzare e non serve usare il free store a volte (come?)

Effetti collaterali: A volte i compilatori vogliono che i template siano completamente DEFINITI
Ovvero devono vedere la DEFINIZIONE di classi template e di funzioni template PRIMA di usarle
Di solito servono solo le dichiarazioni, ma con i template è così
Divisione dell'header: .h contengono DEF di classi, DICHIARAZ di funzioni
.hpp DEFINIZIONI DI FUNZIONI TEMPLATE
Inclusi nei file .cpp

LEZIONE 12
12.1 STL

STL: Parte di libreria std che fornisce contenitori ed algoritmi, tutti generici
Accesso standardizzato ai contenitori, ovvero possiamo accedere ai dati senza sapere come sono memorizzati e con prestazioni indipendenti dal tipo di dato

Sequenza: Elenco di elementi generici con INIZIO e FINE, gestito tramite ITERATORI
S = [Begin, End[

Divisione tra container e algo: Algo deve poter operare senza conoscere l'implementazione di container
La struttura non deve mostrare la sua implementazione, basta rendere public gli iteratori, che sono di TIPO StrutturaDato<T>::iterator

Algoritmi generici:

```
template<typename Iterator>
Iterator high (Iterator first, Iterator last) {
        Iterator max = first;
        for(Iterator p = first, p!=last, p++)
                if(*max < *p) max = p;          // se è definito il confronto < per T
        return max;
}
```

Funziona con qualsiasi struttura, dato che ogni struttura stl ha il suo begin() e end()
Viene quindi creato un high per ogni tipo di iteratore che lo chiama

12.2 Implementazione di iteratore

Iteratori: WRAPPANO UN PUNTATORE CON FUNZIONI MEMBRO
Consentono di scansionare un contenitore ed operare con i suoi dati, NON sono l'unico modo
BEGIN = primo elemento
END = primo elemento vuoto dopo l'ultimo elemento
Classe definita DENTRO LA CLASSE DEL LORO CONTAINER

Operatori standard: Per essere standard, un iteratore deve implementare l'overloading dei seguenti op:
++, - -, *, ==, !=

Construttore:

```
template<typename Elem>
class list<Elem>::iterator {
        iterator(Link<Elem>* p) : curr{p} {}
}
```

Keyword auto: Spesso il compilatore DEDUCE DA SOLO il tipo di un dato
Utile per gli iteratori, alleggerisce il codice
Al posto di list<T>::iterator iter = list.begin() basta usare auto iter = list.begin()
ATTENZIONE: spesso ambiguo, se l'RVALUE è LITERAL, auto rende CONST la var
auto x = 3; // const
auto s1 = "Ciao"; // const char *
std::string s2 = "Ciao"; // std::string normale

12.3 Contenitori STL

Contenitore STL generico: Sequenza di elementi [begin(), end()[
Operazioni di copia
Il tipo degli elementi è generico, value_type
Ha iteratori con operatori standard *, ++, - -, ==, !=
Operazioni standard: insert, erase, front, back, push_back, push_front, size
Operatori di CONFRONTO TRA CONTENITORI

std::vector: Ha tutto ciò che deve avere un contenitore STL
Anche operator[], altro modo di accedere ai suoi dati
Range check con std::vector::at(i)
Espandibile, elementi contigui in memoria (sono in un array)
Insert(value, pos) ed Erase(pos) inefficienti dato che devono fare shift non sono O(1)
e shiftando rendono invalidi i gli iteratori che stanno puntando alle celle di memoria

std::list Come vector, ma non ha [] e gli elementi sono nodi concatenati, non contigui in memoria
Insert ed Erase sono quindi O(1) e non invalidano gli iteratori

LEZIONE 13

13.1 Ereditarietà

Cosa fa: Implementa il polimorfismo a tempo di esecuzione grazie all'overriding, upcasting e downcasting
Incapsulamento con la protezione dei propri membri

Has Object: Classe base, astratta
Is Object: Derivazione di una classe astratta

Classe Base: Implementa un'interfaccia comune a tutte le classi derivate
Classe Derivata: Eredita l'interfaccia della classe base, può sovrascrivere ALCUNI comportamenti
E' quindi una specializzazione della classe base.
Con una ereditarietà public eredita tutto ciò che è public e protected della classe base

Slicing: Quando faccio UPCASTING di una classe derivata, perdo le informazioni sui suoi membri che non sono previsti dalla classe base
virtual void funct();
SOLUZIONE: Disabilitare copy constructor e copy assignment mettendo =delete nelle loro (dichiarazioni nell'interfaccia della classe base)

Puntatori e derivazione: Un puntatore di TIPO Base può puntare ad oggetti di tipo Derivato
Vector<*Base> v può contenere puntatori a Base e a qualsiasi sua Derivata
(*v[i]).func() chiama la funzione func() dichiarata in Base, ma definita nella classe corrispondente al tipo di dato a cui punta v[i]
Ovviamente non posso chiamare funzioni che sono dichiarate e definite in una sola classe derivata dato che in quella base non ci sono per lo slicing

13.2 Funzioni virtuali pure

Rendere astratta una classe base: Costruttore PROTECTED
Almeno una funzione virtuale pura

Funzioni Virtuali: Funzioni che se NON PURE, possono essere implementate nella classe BASE
Le classi derivate possono farne l'OVERRIDE: (stessa const-ness, stesso nome, stessi parametri (stessi tipi), return value ANCHE DIVERSO) (solo di f virtual)
Pure: Non posso essere definite nella classe base, solo dichiarate
Sintassi: virtual void funct() =0;
DEVONO essere definite in ogni classe derivata, errore in compilaz else
Override esplicito

Layout di oggetti: Ogni classe derivata ha una VTBL ed un VPTR che punta a tale tabella
Le righe della VTBL sono PUNTATORI alle funzioni di cui si è fatto l'override nelle classi derivate di una funz virtuale della classe base

13.3 Recap Ghidoni

Polimorfismo: Associare comportamenti diversi alla stessa definizione

Dinamico: A runtime con ereditarietà ed overriding
Classi base raggruppano funzionalità comuni in funzioni virtuali
Classi derivate specializzano le classi base
Binding runtime: void myDraw(const GeoObj& obj) { obj.draw(); }
Accetta qualsiasi derivata di GeoObj, grazie al VPTR viene chiamata la draw giusta = binding runtime

Statico: In compilazione con template
Sintassi condivisa da ogni istanza del template, ovvero ogni funzione generata dal compilatore a partire dal template per ogni tipo diverso che la utilizza
La sintassi DEVE ESSERE SUPPORTATA DA OGNI UTILIZZATORE, sennè errore in comp
Binding compilazione: template<typename G>
void myDraw(const G& obj) { obj.draw(); }
Funziona come prima per qualsiasi oggetto che abbia la funzione draw sennò errore in compilazione
Vengono create 3 myDraw diverse se myDraw è chiamata passando 3 oggetti di tipo diverso

Altre forme: Overloading di funzioni e di operatori
Casting dinamico o statico
Casting dinamico: Navigare la gerarchia di ereditarietà
Per il downcasting (upcasting è sempre valido)
Effettua controlli a runtime: devono essere chiamate solo le funz virtuali della classe padre e non quelle del figlio, dato che la classe padre non le vede.
Se ciò accade usando un puntatore, ritorna nullptr
Se ciò accade usando una reference, lancia eccezione

LEZIONE 14

14.1 Risorse ed Eccezioni

Gestione di risorse: Tutto ciò che riguarda la loro acquisizione, utilizzo e rilascio con NEW e DELETE
Può essere corrotta dalle ECCEZIONI

Esempio:
```
void sus(int x) {
    int *p = new int[20] {1,2,3}; // resto sono 0
    if(x < 1000) p[0] = p.at(x); // può lanciare eccezioni, in tal caso non si fa il delete
    delete p
}
```

RAII: Resource Acquisition In Initialization of an object
Tecnica che lega lifecycle di una risorsa da acquisire al lifecycle di un oggetto
Risorsa acquisita = oggetto definito, Risorsa rilasciata = oggetto distrutto

Esempio:
```
vector<int>* make_vec() {
    vector<int> *p = new vector<int>; // non si sa a cosa inizializza
    (*p).push_back(10);
    try {
        std::cout << (*p).at(5);
    } catch (...) {
        delete p;
        throw Exception(); // se ci fosse return nullptr; sarebbe no throw guarantee
    }
    return p;
}
```
Se non ci sono eccezioni: Il vector "esce dallo scope" dato che con il return si ritorna un puntatore al vector

Se ci sono eccezioni: Basic Guarantee funzione che con un unico blocco try-catch fa sì che se ci sono eccezioni NON CI SIANO MEMORY LEAK
OGNI FUNZIONE STL HA LA BASIC GUARENTEE

Strong guarantee: Basic guarantee + gli argomenti passati come parametri sono ripristinati al valore che avevano prima della chiamata (backup)

No throw guarantee: Funzione che non lancia eccezioni

14.2 Smart Pointer

Smart Ptr: Presenti in std, in <memory>
Wrappano un puntatore
Gestiscono automaticamente la deallocazione della memoria quando escono dallo scope (ovvero il loro distruttore, chiamato in automatico, fa il delete del ptr wrappato)
Accesso al dato del puntatore che wrappano con * e ->, ovvero come se fossero ptr norm

Unique Ptr: Unique perchè due unique_ptr non possono wrappare lo stesso puntatore
Quindi costruttore di copia ed assegnamento di copia sono disabilitati
E' comunque possibile ritornare uno unique_ptr con le move semantics

Esempio:
```
unique_ptr<int> allocate_int(int x) {
    unique_ptr<int> p {new int(x) };
    return p;
}
```

```
unique_ptr<int> v = allocate_int(10); // move constructor
int *p = v.release();
v = allocate_int(42); // move assignment
delete p; // importante
```

Reset: Deallocata la memoria a cui punta il puntatore wrappato
Eventualmente wrappa un nuovo puntatore se passato come parametro

Rilascio: ESTRAE il puntatore dallo unique_ptr, LO RITORNA
Il puntatore wrappato da unique_ptr è ora nullptr
La DEALLOCAZIONE NON AVVIENE e NON E' PIU' GESTITA DALLO UNIQUE_PTR

Esempio:

```
vector<int>* make_vec() {
    unique_ptr<vector<int>> p {new vector<int>};
    // FILL CON EVENTUALI ECCEZIONI
    return p.release();
}
```

Cosa accade: Se ci sono eccezioni, lo scope di dello unique_ptr termina quindi viene chiamato il suo distruttore in automatico e la memoria viene deallocata
Poi IL CHIAMANTE DOVRA' FARE UN DELETE se non ci sono eccezioni dato che viene rilasciato il puntatore wrappato

Debolezza unique_ptr:

```
int *p = new int{42};
unique_ptr<int> u1{p};
unique_ptr<int> u2{p};
// Compila ma errore LOGICO
// SOLUZIONE: make_unique che prende come parametro 42, non un int ptr
```

Shared ptr: Rimuove i vincoli dello unique_ptr
Ovvero lo stesso puntatore può essere wrappato da più shared_ptr che lo condividono
Esistono quindi copy ctor e copy assignment

Reference outing: Gli shared_ptr che detengono lo stesso ptr si conoscono tra loro
Memoria rilasciata solo quando l'ultimo ptr che la detiene viene distrutto
Maggior consumo di risorse

Esempio:

```
shared_ptr<int> allocate_int(int x) {
    shared_ptr<int> p {new int(x) };
    return p;
}
shared_ptr<int> v = allocate_int(10); // move constructor (se il compilatore non lo elide)
shared_ptr<int> p {v}; // ok, copy constructor
```

LEZIONE 15

15.1 Algoritmi STL intro

Generalità: La libreria STL ne fornisce un'ottantina
Sono tutti template, quindi compatibili con qualsiasi contenitore STL anche perchè usano gli iteratori
Permettono di risparmiare codice

Implementazione find:

```
template<typename In, typename T>
In find(In first, In last, const T& val) {
    while(first!=last && *first !=val)
        first++;
    return first;
}
```

Standard: RITORNARE LAST SE NON SI TROVA VAL
Comparable: Per il tipo T deve essere definito == e != sennò errore in comp
Genericità: Rispetto al container (iteratori) e rispetto al tipo contenuto

15.2 Predicati

Motivazione: Il find_if cerca il primo valore che soddisfa un criterio

```
template <typename In, typename Pred>
In find_if(In first, In last, Pred pred) {
    while(first != last && !pred(*first))
        first++;
    return first;
}
```

Criterio: pred(*first) chiama la funz. pred con argomento *first, ritorna true o false

Predicato: Funzione che ritorna true o false
Viene passata come parametro a find_if come un function object o function pointer

Funct Ptr: Il nome di una funzione, se passato come parametro, è un puntatore a funzione
Però non si sa come passargli più di 1 parametro senza usare variabili globali
Notazione del C

Function Obj: Oggetto in cui si implementa l'overloading di operator(), che sarà il predicato
Rendere operator() inline (quindi IN CLASS DEFINITION o scrivere inline esplicitam)

Esempi:
```
bool odd(int x) { return x%2 == 0; }
void f(vector<int>& v) {
    auto p = find_if(v.begin(), v.end(), odd);
    if(p!= v.end()) { // *p è dispari };
}
```
Cosa accade: Deduzione automatica dei tipi del template a partire dal tipo degli iteratori passati come parametri e da odd
Viene passato il func ptr odd, find_if chiamerà odd(*first)

```
class Larger_than {
    int v;
    public:
        Larger_than(int w) : v{w} {}
        bool operator(int x) { return x > v; }
};
auto p = find_if(v.begin(), v.end(), Larger_than(31));
```
Cosa accade: Deduzione automatica anche qui
Viene creato un oggetto di tipo Larger_than e passato subito a find_if
Il find_if userà Larger_than::operator(*first) come predicato

Debolezze: I function obj possono essere creati come oggetti normali in qualsiasi scope e possiamo usare il loro operator() anche non come un predicato
A noi serve crearli solo quando chiamiamo find_if ed usarli come predicati

15.3 Lambda Function

Lambda function: Strumento che permette di definire un function object in modo anonimo, permettendone l'uso una sola volta

STL sort: Ne esistono più versioni
Una ordina usando l'operator <
Una ordina in base al predicato che viene fornito, se ad esempio un oggetto può essere ordinato in base a più dati membro

Esempio:
```
struct Record {
    std::string name;
    char addr[42];
}
```
Posso ordinare vector<Record> in base a name con < oppure in base ad addr con strcmp

```
std::sort(v.begin(), v.end(),
        [](const Record& a, const Record& b) {
                return a.name < b.name;
        });
```

Cosa accade: Con [] si dice di creare una lambda function
Viene creato a runtime un function object il cui operator() è la λ fnc
Tale oggetto viene distrutto al termine di std::sort
Funziona perchè nel codice di sort il predicato riceve 2 argomenti

```
void f(vector<double>& v, const double& epsilon) {
        std::transform(v.begin(), v.end(), v.begin(),
                [epsilon](double d) -> double {
                        if(d < epsilon) { return 0; }
                        return d;
                });
```

Cosa accade: Viene scritto esplicitamente il valore di return, non necessario (->type)
Con [epsilon] si dice di catturare la variabile locale epsilon e passarla per valore alla λ function, quindi usata nel costruttore del func obj
d è il parametro che std::transform passerà al predicato
F OBJ CREATO A RUNTIME

Catturare le variabili locali:

[&epsilon, zeta]	epsilon per riferimento, zeta per valore
[&] =	tutte le variabili locali catturate per riferimento
[=]	tutte le variabili locali catturate per valore
[&, epsilon]	tutte per rif, tranne epsilon per valore
[=, &epsilon]	tutte per valore, tranne epsilon per riferimento

15.4 Container associativi

Map: E' ORDINATA, realizzata con AVL
Rispetto ad un vector offre una ricerca più veloce ed esplicita indicizzazione con chiavi

Indicizzazione con chiavi:

```
map<string, int> words; // string = T1,  int = T2
for(string s; cin >> s)
        words[s]++;
```

E' definito T2& operator[](T1 key)
Se non esiste una coppia con chiave s la crea inizializzando il valore usando il costruttore di default di T2

Coppie chiave valore: Le coppie (key, val) in una std::map sono oggetti di tipo std::pair
I loro membri sono T1 first, T2 second
Si usa std::make_pair() per creare coppie

```
for(const auto& p: words)
        std::cout << p.first << p.second
// STAMPA IN BASE ALL'ORDINE CRESCENTE DI CHIAVI
```

Set: Contenitore ORDINATO di SOLE CHIAVI

No operator[], basta sapere se la chiave c'è o no con find() o find_if()

No push_back(), ci sono INSERT ed ERASE che tengono ordinata la struttura

Deve essere definita una relazione d'ordine totale per le chiavi, attraverso un Comparator

Esempio:

```
struct Fruit {
        string name;
        int count;
        double price;
        Date last_sale;
};
struct Fruit_Order {
        bool operator()(const Fruit& a, const Fruit& b) { return a.name < b.name; }
};
std::set<Fruit, Fruit_Order> inventory;
inventory.insert(Fruit{"Apple", …});
```

Comparator: E' un oggetto di tipo Fruit_Order
Di fatto è FUNCTION OBJECT dato che si fa ovrld operator()
Il tipo Fruit_Order è il parametro template usato per definire
il comparator nella struttura interna del set

```
for(auto p = inventory.begin(); p != inventory.end(); p++) {
        std::cout << *p; // valid se esiste overload di operator<< come helper di fruit
// STAMPA IN BASE ALL'ORDINE CRESCENTE DI CHIAVI
```