Lab di Algoritmi e Strutture Dati

Florindo Zecconi
Università di Napoli Federico II
2024

Indice

1 Compilazione

1.1 Compilazione Singolo file

Un file sorgente in C++ ha come estensione `.cpp` il comando per compilare un file sorgente in `.cpp` è:

```
g++ sorgente.cpp -o nomefile
```

- **g++** : Compilatore **GNU** (ne esistono anche altri)
- flag **-o** : per rinominare il file eseguibile in uscita
- **.cpp** : Estensione file sorgente C++

Possiamo esprimere anche un livello ti ottimizzazione del codice:

```
g++ -03 sorgente.cpp -o nomefile
```

- **-O3** : L'opzione -ON con N = 0,1,2,3 sono i livelli di ottimizzazione del codice

1.2 Compilazione Multifile

Posso **spezzare** il mio codice in più file sorgente: tutti i file verranno compilati in file oggetto con estensione "`.o`" e poi uniti fra di loro termite il **linker** nella fase di **linking**.

il comando per fare ciò è:

```
g++ -c eseguibile_sorgente.cpp

g++ -c eseguibile_sorgente.cpp -o nomefile
```

- **g++** : Compilatore **GNU** (ne esistono anche altri)
- flag **-c** : non produce l'eseguibile ma solo il file oggetto(file sorgente tradotto in linguaggio macchina)
- flag **-o** : per rinominare il file oggetto (solo se vogliamo un nome diverso da sorgente)
- **.cpp** : Estensione file sorgente C++

Per fare il **linking** abbiamo:

```
g++ file1.o fil2.o -o nomefile
```

quando il programma cresce di dimensione conviene ricompilare solo i files modificati per velocizzare il processo di compilazione del programma. e successivamente fare il **linking** fra i vari file oggetti.

1.3 ScriptBash o ScriptBatch

Possiamo utilizzare dei script di shell (batch `.bat` o bash `.sh`): sono dei file che permettono di eseguire delle istruzioni automaticamente da schell. possiamo scrivere un programma che esegue correttamente tutti i passi per la compilazione de file che sono stati modificati e creare un eseguitivo.

```
//esempio di utilizzo di file bash.sh
#\!/bin/bash
g++ -03 file1.cpp file2.cpp -o eseguibile
```

1.4 Divisione in più file di un programma

Un porgramma si deve dividere in più file per i **seguenti vantaggi:**

- Più facilita nella lettura e comprensione del codice sia per noi che per i futuri programmatori che leggeranno il nostro codice
- Facilita nel lavoro di gruppo
- minor tempo di compilazione

1.5 Modularità

- **Struttura logica**: I componenti devono essere organizzati in modo che componenti che sono correlati siano nello stessi file.
  quindi pezzi di codice(**componenti logici**) che sono molto correlati fra di loro andranno nello stesso file, altrimenti divisi.
- **Struttura fisica**: Come i vari pezzi di codice(**componenti logici**) sono divisi nei file.
  questa suddivisone deve essere:
  - Consistente: Componenti **logici molto correlati** fra di loro devono essere nello stesso file per rendere consistente il progetto.
  - Comprensibile . Chiaro per chi lo legge.
  - Flessibile : Deve adattarsi a diverse esigenze.
- Molto importante anche dividere **l'interfacce** (es: firme delle funzioni) dalla sua **L'implementazione** (es: Quello che fa la funzione)

1.6 Header File

La direttiva al pre-processore `#include nomefile.hpp` può avere più forme:

```
#include <iostream>  //cerca la libereria nella cartella di default
#include "myheader.hpp" //controlera' nella cartella del file in cui e'
    steto incluso
```

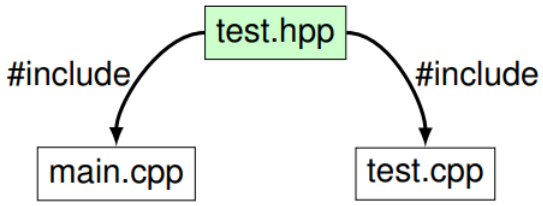
Nel **HeaderFile** vanno solo le dichiarazioni di funzioni, delle macro per il pre-processroe o struct. più generalmente vanno solo la dichiarazione dei vari elementi.

1.6.1 Inclusione Multipla

Può accadere avvolte che uno stesso file `.hpp` venga incluso piu volte in file diversi. facciamo un esempio:

- Abbiamo un file sorgente `test.hpp`
- abbiamo un secondo file `main.cpp`

Per ipotesi abbiamo incluso un file di nome `test.hpp` in tutti e due file.



Come possiamo risolvere? Basterà inserire nel file Header un **"flag"** che ci indichi se quel file è stato già inserito o meno.

```
#ifndef TEST_HPP
#define TEST_HPP
void test();
#endif
```

quando viene incluso il file `test.hpp` prima di essere inserito nel file controllerà se `TEST_HPP` non sia già stata dichiarata, nel caso in cui lo fosse non sarà incluso nulla.

1.6.2 Sudivisone File Haeder File

- gli Header contiene **l'interfaccia** di tutto il contenuto del file sorgente corrispondente

- Ogni file sorgente contiene un `#include` per l'header corrispondente e poi eventuali altri header per tutto ciò che utilizza e la cui implementazione è data in altri file
- creare un file header `.hpp` per ogni file sorgente `.cpp`

1.7 Fasi compilazione

In questa fase abbiamo 4 fasi che restituiscono i seguenti file(Nome file "`primo.cpp`"):

- **primo.i:** output Preprocessing
- **primo.s:** output fase di Compiling
- **primo.o:** output assembly
- **a.out:** output della fase di linking

1.7.1 Preprocessing stage

in questa fase succederà:

- rimozione commenti
- saranno esegui le macro (es. `#define`) , verranno sostituite al interno del programma
- i file saranno inclusi attraverso `#include` verranno inseriti le posizioni precisi dei file inclusi esegue tutte le azioni precedute dal #

1.7.2 Compiling stage

in questa fase viene creato il codice assembly. Ogni macchina ha la propria architettura e il proprio funzionamento. Per colpa di quest'ultima il codice assembly varia da processore a processore. Per questo ogni volta che il programma viene compilato sarà trasformato prima in assembly.

1.7.3 Assembly stage

- il **codice assembly** viene convertito in linguaggio macchina
- il file risultato è chiamato file oggetto
- **non risolve** chiamate a funzione.
- il file output non sarà visibile con normali tool di testo perché verranno codificati in `ASCII`
- il file oggetto `.o` può essere generato tramite il flag -c:
  `gcc -c primo.c` . questo comando farà tutti le fasi tranne il linking cosi non generando l'eseguibile.

1.7.4 Linking stage

- vengono risolte le **chiamate a funzione**. verranno uniti tutti i file oggetti fra di loro
- viene creato l'eseguibile

2 Tipi di dati

2.1 Tipi fondamentali

In C++ abbiamo dei **tipi fondamentali**, che sono implementati di base dal linguaggio. Si chiamano **Built-in(integrati)**
questi tipi sono:

- **Built-in (Integrati):**
  - **Boolean**
  - **char**
  - **int** , long int, long long int...
  - float , **double**, long double
  - **void**
  - **Puntatori**(es: **int***), **Array** e **Riferimenti**(**double&**)
- **User-defined:**
  - enum
  - **classes**
  - **tipi introdotti dalle librerie standard**

Un '**tipo**' serve per definire il **dominio** dei valori che può assumere quel elemento. Il '**tipo**' serve anche per definire l'insieme di **operazioni** che si possono essere fate su quel tipo.

poi intendiamo come **letterale** tutti i valori inseriti direttamente nel codice

```
1 int x = 10;          // Letterale numerico
2 string nome = "John";   // Letterale di stringa
3 bool flag = true;   // Letterale booleano
```

2.1.1 Sizeof()

Il `Sizeof()` permette di controllare lo spazio che occupa un determinato tipo o variabile. Il peso dei tipi va in base all'implementazione del programmatore. Quindi il programmatore può modificare le grandezze di default. questo può comportare dei vantaggi e dei svantaggi.

Vantaggi:

- Utilizzando grandezze personalizzate posso sfruttare a meglio il mio hardware(memoria)

Svantaggi:

- Il codice diventa meno portatile poiché diventerebbe più specifico per un certo hardware(meno portatile)
- rischio che con le versioni future del compilatore non potrebbe funzionare poiché si sono modificati valori di default

Quindi dobbiamo modificare questi valori solo quando è veramente necessario poiché a noi ci interessa la portabilità del codice

2.2 Inizializzazioni

Possiamo inizializzare una variabile in 4 modi:

1. Graffe(list initialization): `int a1 {15};`
2. Uguale + graffe: `int a2 = {15};`
3. Uguale: `int a3 = 15;`
4. Tonde: `int a4(15);`

Cosa utilizzare?

- La prima forma è **PREFERIBILE** poiché controlla che varie conversioni non perdano informazioni.
- Se non viene inserito nessun valore viene inizializzato a un valore di default, questo metodo di inizializzazione e più leggibile e permette meno errori.

```
1 int x2 = val; // if val==7.9, x2 diventa 7
2 char c2 = val2; // if val2==1025, c2 diventa 1
3
4 int x3 {val}; // error : possible troncamento
5
6 char c3 {val2}; // error : possible restrigimento
7
8 char c4 {24}; // OK: 24 can be represented exactly as a char
```

```
9
10 char c5 {264}; // error (assuming 8-bit chars): 264 cannot be
       represented as a char
11
12 int x4 {2.0}; // error : no double to int value conversion
```

Possiamo inizializzare anche una lista di elementi:

```
1 int a[] {2,3,4}
```

2.3 Tipo Booleano

Il tipo booleano può essere solo due valori `True` o `False`.
Possiamo subito dire che:

- Un qualsiasi valore intero diverso da zero è convertito in booleano come `True`, altrimenti un valore uguale a zero come `False`

```
1 bool b1 = 7; // 7!=0, so b becomes true
2 bool b2 {7}; // error : il bool non puo rappresentare 7
3 int i1 = true; // i1 becomes 1
4 int i2 {true}; // i2 becomes 1
```

- Un puntatore può essere convertito `True` se punta a qualcosa , altrimenti se non punta a nulla(`nullptr`) a `false`

2.4 Tipo carattere

Esistono vari tipi di caratteri (come i latini o gli ideogrammi cinesi ecc.) ognuno di essi ha la sua propia codifica.

Ovviamente non esiste solo il tipo `char`, Esistono numerosi tipi che supportano anche diversi tipi di codifica. nella maggior parte dei casi basta il `char` con i suoi `8 bits`.

Il `char` supporta la codifica ASCII ed è in grado di immagazzinare un solo carattere (1 Byte).

Esistono anche dei caratteri speciali che utilizzano il carattere " \" come "\n".

2.5 Tipo Intero

Abbiamo diversi tipi di intero:

- Con il segno(`unsigned`)
- Diverse dimensioni(`int , short int , long int ecc.`)
- utilizzare `unsigned` per risparmiare `1 bit` non ha senso, è quasi in rilevante in vasta scala.
- **letterali** sono interi e tutti espressi in decimale , esadecimale , ottale(es: base 10(1), base 16(0x1) , base 8(01))
- Il tipo di ogni letterale è deciso dal suo suffisso, se c'è (ad esempio: u o U per `usigned`, l o L for `long`, ll o LL for `long long`) e dal tipo di lunghezza minima che ne contiene il valore.

2.6 Tipo virgola mobile (floating point)

- `double` è il **default**. Quindi il tipo base per esprimere numeri con la virgola.
- I Sono letterali validi: 1.23, .23, -2.3, 0.23, 1., 1.0, 1.2e10, 1.23e-15 (niente spazi attorno alla e)
- Suffisso f se voglio che sia un `float (2.0f)`
- Suffisso L per `long double`.

2.7 Tipo void

- Non possono esistere oggetti di tipo void.
- Si usa solo in due casi:
  1. Per indicare che una funzione non restituisce nulla;
  2. Per indicare il tipo di un puntatore ad un oggetto di tipo sconosciuto

2.8 Tipo costante

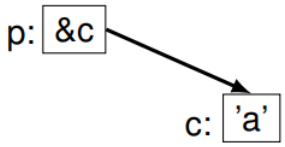
`const`: "Prometto di non cambiare questo valore" e il compilatore controlla che sia proprio così.
`constexpr` : "Da valutare a tempo di compilazione". Possono essere `constexpr` anche funzioni, ma solo se molto semplici (solo un'istruzione di `return`)
Distinzione sottile, che per il momento potete non considerare

2.9 Tipo punatore e riferimento

- Possiamo accedere a una variabile nei seguenti modi:
  - usando il nome
  - dal suo indirizzo di memoria, tenendo conto del tipo della variabile
  - dal suo riferimento
- Puntatori e riferimenti servono esattamente a questo, in due modi leggermente diversi.

Esempio di puntatore:

```
1 char c = 'a';
2 char* p = &c;
3 char c2 = *p;
```



- `c2` assume il valore puntato da `p`, nel senso che `c2` assume il valore memorizzato all'indirizzo contenuto in `p`
- quindi: la memoria viene letta sapendo il tipo della variabile, altrimenti sarebbe impossibile.
- L'oggetto puntato da `p` è `c`, il cui valore è `'a'`
- in conclusione: `c2` assume il valore del letterale `'a'`

Quando accediamo a un valore puntato da un puntatore(quindi trmite l'Indirizzo contenuto nel puntatore) stiamo effettuando la differenziazione tramite l'operatore unario `*`.

I puntatori hanno una aritmetica chiamata **aritmetica dei puntatori** che

utilizza l'indirizzamento della memoria. Ci permette di lavorare sugli indirizzi facendo delle operazioni(es: sommare un valore a un indirizzo) su questi indirizzi. Questa indirizzazione avviene tramite **bytes**.

Quindi il più piccolo oggetto che può venir allocato e indirizzato indipendentemente è il `char`, che corrisponde ad un byte. infatti il `bool` occupa almeno tanta memoria quanto un `char`.

Ottimo se ho bisogno di implementare applicazioni che scendano a quel livello di dettaglio (ad esempio, un sistema operativo)
Se invece voglio fare una buona applicazione portabile, meglio non usare queste operazioni di basso livello.

Per trasformare un qualsiasi tipo `X` nel corrispondente puntatore a `X` dobbiamo aggiungere al tipo il suffisso `*`

```
int* pi; // pi: un puntatore ad
intero
char** ppc; // ppc: un puntatore a puntatore a carattere
int* ap[15]; // ap: un vettore di 15 puntatori a int
```

Invece per le funzioni:

```
int(*fp)(char*); // fp: un puntatore a una funzione
                 // che prende in ingresso un puntatore
                 // a carattere e restituisce un intero

int* f(char*) // f:  prende in ingresso
                 // un puntatore a carattere e restituisce un puntatore ad
                     intero
```

2.9.1 Puntatori a void

- Se dichiariamo una variabile di tipo `void*`, le possiamo assegnare un puntatore ad un qualsiasi tipo di oggetto.
- Possiamo considerarlo un **puntatore ad un oggetto di tipo sconosciuto**
- **Però** non può essere né un puntatore a funzione né un puntatore a un membro di una classe (inclusi i membri funzione o metodi)
- Se ho due variabili di tipo `void*`, posso:
  - assegnare il valore dell'una all'altra
  - confrontarle (sia eguaglianza che diseguaglianza)
  - posso convertirle **esplicitamente** ad un altro tipo
- Qualsiasi altra operazione può essere **pericolosa** e quindi va evitata: dà **errore** a tempo di compilazione
- Per usare un `void*` dobbiamo convertirlo esplicitamente ad un puntatore di un dato tipo

```
1  void f(int* pi){
2  void* pv = pi; // ok: converto implicitamente a
3  // puntatore a intero
4
5  *pv; // errore: non posso accedere al valore puntato da void*
6  // si dice <<dereferenziare>>
7
8  ++pv; // errore: non posso incrementare il puntatore a void
9  // non conosco la dimensione dell'oggetto puntato!
10
11 int* pi2 = static_cast<int*>(pv);
12 // questo lo posso fare: cast esplicito
13
14 double* pd1 = pv; // errore
15 double* pd2 = pi; // errore
16 double* pd3 = static_cast<double*>(pv); // pericoloso
17 }
```

`int* pi2 = static_cast<int*>(pv);`
dove `static_cast<int*>` è un cast statico , questo cast aviene durante la compilazione e non durate il run-time

- In genere, usare un puntatore ad un tipo `X1` per puntare ad un oggetto di tipo `X2` è pericoloso(es: Il codi qui sopra). quindi usare il "cast" potrebbe creare problemi
- Pensiamo ad esempio che la dimensione di un oggetto può dipendere dall'implementazione . . .

- Cerchiamo quindi di usare questo operatore il meno possibile
- Il puntatore `void*` si usa per operazioni a basso livello sulla memoria
- In questi casi, gli usi più diffusi di `void*` sono:
  - voglio passare un parametro ad una funzione senza fare ipotesi sul tipo dell'oggetto puntato
  - la funzione deve tornare un puntatore ma senza fare ipotesi sul tipo dell'oggetto puntato
- Se l'operazione è a più alto livello, meglio utilizzare soluzioni basate su un progetto orientato agli oggetti

2.9.2 nullptr

Letterale che rappresenta il **puntatore nullo**, ovvero il puntatore che non punta a nessun oggetto I Può venir assegnato a qualsiasi tipo di puntatore, ma non agli altri tipi built-in:

```
1 int* pi = nullptr;
2 double* pd = nullptr;
3 int i = nullptr; // errore: i non e' un puntatore!
```

Un unico `nullptr` per qualsiasi tipo di puntatore

- Nessun oggetto può venir allocato all'indirizzo 0 (ovvero il pattern con tutti i bit a zero), quindi una volta si usava l'intero 0 al posto del puntatore nullo
- Addirittura si definiva una macro `NULL` per rappresentare il puntatore `null`
- Tuttavia
  - la definizione di `NULL` dipende dall'implementazione (ad esempio, può essere 0 o 0L)
  - la definizione usata in `C` (`(void*)0`) è illegale in `C++`
- Conclusione: usate `nullptr`!

2.10 Array

- Un array è una raccolta di dati di tipi omogenei fra di loro.
- Un array di Tipo **X** deve essere indicato nel seguente modo se è definito in modo statico **X NomeArray[size]**. l'indice va da `0` (index 0) fino a `size-1`(index -1)

```
float v[3]; // an array of three floats: v[0], v[1], v[2]
char* a[32]; // an array of 32 pointers to char: a[0] .. a[31]
```

- Possiamo usare l'operatore `[]` per accedere oppure l'operatore unario `*` (preferibile `[]` poiché `*` lavora troppo a basso livello)
- andare oltre alla lunghezza del array ha un comportamento indefinito e disastroso poiché non si sa cosa c'è dopo.
- quando inizializziamo un array la grandezza (`size`) deve essere ben definita cioè un letterale. altrimenti il comportamento è indefinito.
- possiamo invece usare il **Vector** all'interno della standard library. a differenza del array conserverà la sua grandezza e quindi può essere definita come una variabile.

```
void f(int n)
{
int v1[n]; // error : array size not a constant expression
vector<int> v2(n); // OK: vector with n int elements
}
```

Limitazioni Array built-in:

- Si tratta di una struttura inerentemente di **basso livello**, da usarsi essenzialmente per costruire strutture di più **alto livello**, quali le strutture **vector** e **array** della libreria standard.
  - Non è possibile assegnare un **array** ad un altro **array**. Questo a meno che **l'array** destinazione non sia un puntatore a il tipo del **array** che dovrà essere assegnato.

```
int v6[8] = v5; // error : can't copy an array (cannot assign an
    int* to an array)

v6 = v5; // error : no array assignment
```

  - Come conseguenza, non è possibile passare un **array** ad una funzione per valore(altrimenti si dovrebbe copiare l'intero **array**).
  - Il nome di un **array** viene implicitamente convertito ad un puntatore al suo primo elemento in molti contesti (poi vediamo meglio)
- Uno degli **array** più utilizzati è **l'array** di `char` terminato dal `char \0`: una stringa in stile C: conservata per compatibilità con librerie già esistenti.

2.10.1 Inizializzazione Array

Possiamo inizializzare un Array in diversi modi:

```
int v1[] = { 1, 2, 3, 4 }; //Grandezza ricavata dal numero
                           //di numeri al interno della lista

char v2[] = { 'a', 'b', 'c', 0 };
char v3[2] = { 'a', 'b', 0 }; // error : too many initializers
char v4[3] = { 'a', 'b', 0 }; // OK

//Questo:
int v5[8] = { 1, 2, 3, 4 };
//e' iquivalente a questo
int v5[] = { 1, 2, 3, 4 , 0, 0, 0, 0 }
```

- Nel caso in cui specifichiamo la dimensione e non inseriremo abbastanza elementi durante la inizializzazione allora saranno impostati a zero.
- Se non definiamo il numero di elementi che conterrà un array ma definiamo qual'è elemento ci dovrà essere allora il compilatore capirà da solo quanto spazzino allocare.

2.10.2 Letterali stringa

Un letterale stringa è un tipo costante (`const char`). Ogni letterale finisce con `'\0'`.
Ogni carattere occupa `1 byte` quindi un letterale con 5 caratteri sarà un `const char[5]`

- **letterali stringa sono costanti, quindi immutabili**.
- Se vogliamo usarla come inizializzazione e poi modificarla, dobbiamo usare l'array di caratteri (non costante):

```
char* p="Plato"; // errore da C++11 in poi. poiche' char* non e'
    costante

char p[]="Plato"; // p diventa un array di 5.
```

- letterali stringa sono **allocati staticamente**, quindi possono essere restituiti da funzioni in modo sicuro. Poiché dopo alla chiamata di funzione la stringa è allocata in modo statico non sarà persa.

```
const char* error_message(int i){
// ...
return "range_error";
}
```

- Il codice viene **ottimizzato**, quindi in alcuni casi può succedere che due letterali stringa identici non vengano duplicati. quindi viene usato lo stesso letterale.
- Questo significa che non possiamo ipotizzare con certezza cosa succede.

```
const char* p="Heraclitus";
const char* q="Heraclitus";

void g(){
if(p==q) cout << "one!\n";
}
```

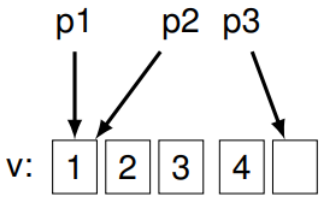
- L'espressione `p==q` confronta gli indirizzi, cioè i valori dei puntatori e non degli oggetti puntati.
- La lista vuota è `""`, ha tipo `const char[1]` e contiene il solo carattere `'\0'`.
- Per rappresentare i caratteri non grafici all'interno di una stringa possiamo utilizzare il `\`:
  `cout « "beep at end of message\a\n";`
- Un problema possono essere le stringhe contenenti il carattere nullo
  - Sono perfettamente legali.
  - La maggior parte dei programmi considererà solo la parte prima del primo carattere nullo: `Jens\000Munk` verrà considerato come `"Jens"`.
- Il carattere di **newline** non può essere incluso in una stringa: una stringa non può stare su più righe. Devo invece usare `'\n'`
- Se una stringa è troppo lunga può essere spezzata

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz"
                    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

//out -> "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

2.10.3 Array come puntatore

- Legame stretto tra array e puntatori: il nome di un array può venir utilizzato come **puntatore al suo primo elemento**.

```
int v[] = {1,2,3,4};

int* p1 = v; // conversione implicita:

// puntatore al primo elemento
int* p2 = &v[0]; // stessa cosa, ma esplicita
int* p3 = v+4; // puntatore all'elemento
// dopo l'ultimo
```



- **Puntare** ad un elemento immediatamente successivo alla fine dell'array non dà errore, purché non si legga o scriva.
- Quando passiamo un `array` come parametro non ci sarà modo di passare una copia. sarà sempre passato il **riferimento** , questa conversione implicita da `array` a `puntatore` farà perdere la grandezza del `array`. quindi non ci sarà modo di evitare questa conversione implicita.

```
int* p4 = v-1; // before the beginning, undefined: don't do it
int* p5 = v+7; // beyond the end, undefined: don't do it

char v[] = "Annemarie";
char* p = v; // implicit conversion of char[] to char*
strlen(p);
strlen(v); // implicita conversione di un char[] to char*, questo
    perche' strlen accetta un array di char.

v = p; // error : cannot assign to array
```

2.10.4 Navigazioni degli Array

Abbiamo due modi per navigare in un array:

- Utilizzare l'operatore `*`. quindi sommare al indirizzo del `array` **l'indice** che vogliamo raggiungere e poi fare la Dereferenziazione per ottenere il contenuto.
- Utilizzare le **parenesi quadrate** con al interno l'indice
- **nessuna delle due e più efficiente del altra**, per una questione di leggibilità e meglio con le parentesi quadrate.

```
void fi(char v[]){
    for(int i=0; v[i]!=0; i++)
        use(v[i]);
}

void gi(char v[]){
    for(char* p=v; p!=0; p++)
        use(*p);
}
```

2.10.5 Array Multidimensionali e Array come argomento di funzione

- Gli array multidimensionali sono rappresentati come array di array
- I Esempio 3x5:

```
int ma[3][5];

void init_ma(){
    for(int i=0; i<3; i++){
        for(int i=0; j<5; j++){
            ma[i][j] =10*i+j;
        }
    }
}

void print_ma(){
    for(int i=0; i<3; i++){
        for(int i=0; j<5; j++){
                cout << ma[i][j] << '\t';
            }
        count << '\n';
    }
}
```

- Come per gli `array` ad una sola dimensione, le dimensioni non sono memorizzate in alcun modo nella struttura dati `array`

- Un array non può venir passato ad una funzione per valore, ma con un puntatore al suo primo elemento

```
void comp(double arg[10]{
    for(int i=0; i < 10; i++)
    arg[i]+=99;
}

void f(){
    double a1[10];
    double a2[5];
    double a3[100];
    comp(a1);
    comp(a2); // disastro!
    comp(a3); // usa solo i primi 10
    elementi di a3
}
```

2.11 Riferimenti

- **Un riferimento** (reference) può essere visto come un nome alternativo per un oggetto, un alias.
- **Analogamente ai puntatori:**
  - è un alias per un oggetto
  - è implementato per conservare l'indirizzo di un oggetto
  - non richiede maggiori risorse computazionali
- **Diversamente dai puntatori, il riferimento:**
  - la sintassi è la stessa che avrei col nome dell'oggetto
  - si riferisce sempre all'oggetto con cui è stata inizializzata
  - **non esiste un "riferimento null"** da controllare: ogni riferimento si riferisce ad un oggetto

```
#include <iostream>

using namespace std;

int main(){
    int x = 10;

    // ref diventa un alias di x
    int& ref = x;

    // Il valore di x viene cambiato!
    ref = 20;
    cout << "x=" << x << '\n';
```

```
14
15      // Se cambio il valore di x, cambia anche
16      quello di ref 30
17      x = 30;
18      cout << "ref=" << ref << '\n';
19
20      return 0;
21  }
```

- Tra i **vantaggi** dei puntatori c'è quello di poter passare ad esempio ad una funzione una grande quantità di dati a basso costo.
- Però un puntatore ha una sintassi un po' pesante . . .
- e può cambiare valore nel tempo, Invece il **riferimento** si riferirà **SEMPRE** al oggetto a cui è stato inizializzato
- Inoltre bisogna sempre gestire la possibilità che il puntatore non stia puntando a nulla **(nullptr)**.
- Coi riferimenti non ho questi problemi.

3 Allocazione in memoria

La vita di un oggetto è determinata dal suo **scope**(è un blocco di codice tra parentesi graffe).

A volte ci occorre creare un oggetto che venga **allocato** nella free memory(heap) e venga **deallocato** solo quando noi lo vogliamo.

Quindi una variabile che vive indipendente dallo **scope**

Queste sono le parole **chiavi** per lavorare sulla **free memory**:

```
1  new            //per creare un oggetto
2  delete      //per distruggerlo
3  delete[ ] //se si tratta di un array
```

3.1 Key New

- `New` crea un oggetto sullo `heap`
- Se la memoria è insufficiente lancerà un eccezione `bad_alloc`
- in caso di buona allocazione sarà restituito un **puntatore** alla locazione di memoria allocata

```
// alloca un int, senza inizializzarlo
int* p1=new int;

// alloca 100 interi senza inizializzarli
int* p2=new int[100]

// dealloca singoli oggetti
delete p1;

// dealloca un array
delete[] p2;
```

- Gli oggetto allocati con **`new`** rimarranno allocati in memoria finché non verranno Deallocati **esplicitamente**

```
// alloca un int e lo inizializza a 7
int* p3=new int(7);

// dealloca singoli oggetti
delete p3;
```

- `new` un oggetto verrà allocato con la parola `new` e sarà richiamato il costruttore della classe(avviene di default anche se non specificato)

3.2 Key Delete

- `delete` è un operatore che può essere utilizzato solo sui **puntatori** ritornati da `new` oppure su puntatori a null(`nullptr`)
- L'Applicazione di `delete` su `nullptr` non avrà nessun effetto
- Se un oggetto è stato creato con `new` esso esiste fino a quando non viene esplicitamente distrutto con `delete`.
- Solo dopo aver usato `delete` a quel punto lo spazio in memoria occupato dall'oggetto può venir riutilizzato.
- Se `delete` viene utilizzata su una **classe** e in questa **classe** sarà definito un **distruttore** allora prima che la memoria venga liberata verrà eseguito il **distruttore**.

3.3 Key Delete[]

- Per liberare lo spazio allocato da una `new`, sia `delete` che `delete[]` devono conoscere la dimensione dell'oggetto allocato.
- A `delete` basta conoscere il tipo dell'oggetto puntato(facilmente reperibile).
- Ma `delete[]` deve conoscere anche il numero di oggetti puntati.
- Quando viene allocata memoria sullo **heap**, la `new []` tiene traccia di quanta memoria/numero di elementi allocata/i,
- di solito salvando l'informazione in un segmento che precede immediatamente la memoria allocata.
- Ne consegue che un oggetto array allocato mediante l'implementazione standard di `new` occuperà uno spazio leggermente superiore del corrispondente statico.
- Quanto meno, infatti, lo spazio necessario a contenere la dimensione dell'oggetto.

  > NB: **`sizeof`** restituisce solo la grandezza di un **tipo** specificato **NON** della memoria fisicamente allocata di una certa istanza.
  >
  > Quindi il calcolo della grandezza di un tipo di dato avviene alla **`Compilazione`** invece il numero di elementi di può sapere solo al tempo di **`Esecuzione`**.

- Questo vale solo per l'allocazione degli array: in tutti gli altri casi viene usato il tipo del puntatore.

3.4 Gestione memoria

I principali problemi della memoria **Libera**(Free memory) è:

- **Ogetti abbandonati(leaked objects)**: Allocare uno spazio di memoria con `new` e poi dimenticarsi di usare `delete` quando non serve più
- **Cancellazione prematura(Premature deletion)**: Quando cancelliamo una locazione(`delete`) di memoria in modo prematuro. quindi dealloco una porzione di memoria e poi in un altra parte del codice ho un puntatore a questa locazione ormai libera, quindi accendendo a una locazione liberata il comportamento è indefinito.
- **Doppia cancellazione(Double deletion)**: Un oggetto deallocato due volte poiché si chiama due volte il suo distruttore.

Esempio Cancellazione prematura:

```
int* p1 = new int{99};

int* p2 = p1; // poteziali proplemi, poiche se la locazione viene
            //deallocata abbiamo due puntatori a un locazione che
            // non esiste creando problemi

delete p1; // now p2 doesn't point to a valid object

p1 = nullptr; // Diamo un valore nullo per sicurezza

char* p3 = new char{'x'}; // p3 ora punta alla memoria puntata da p2

*p2 = 999; // Potrebbe causare problemi

cout << *p3 << '\n'; // non stampa x
```

NB: Perché `p3` punterà alla stessa locazione di `p2` ? Questo perché la **memoria** per allocare oggetti del nostro programma è **ben precisa**. Si inizia ad **allocare** dal primo **indirizzo** fino al ultimo della memoria dedicata al nostro programma. In questo esempio abbiamo allocato un `int`(che occupa i primi `8 bytes`) e subito dopo lo **de-allochiamo**. quando **allochiamo** una nuova porzione di memoria cerchiamo uno spazio libero per l'allocazione(che nel nostro deve poter contenere un `char`) , essendo la **memoria vuota** lo inserirà a partire dal primo indirizzo delle memoria dedicata al programma. Essendo che **punterà** proprio a quel indirizzo quando andremo a assegnare un `int(999)` sarà valutato come un `char` ed uscirà þ(Simbolo associato al codice ASCII).

3.4.1 Problemi: Doppia cancellazione

- Il problema nasce dal fatto che tipicamente il gestore delle risorse non è in grado di tenere traccia di quale parte di codice è proprietaria di una risorsa. quindi se faccio una doppia cancellazione io non so quella locazione di memoria effettivamente apparteneva originariamente apparteneva a qualche altra funzione
- Quindi il risultato di un doppio `delete` non è prevedibile

```
void sloppy(){
int* p=new int[1000];
// usa *p ...
delete[] p;
// ... Altro o attesa ...
delete[] p;
// la funzione non possiede *p!
}
```

- Tra il primo e il secondo delete[] la memoria può essere stata riutilizzata

3.5 Buone norme d'utilizzo

- Evitiamo di mettere oggetti sullo free store a meno che non sia strettamente necessario: meglio usare uno scope il più ristretto possibile
- Quando un oggetto viene costruito sullo free store, meglio inserirlo all'interno di un **gestore di oggetti** (handle) che preveda un distruttore
- Come regola pratica: usiamo `new` dove c'è un costruttore e `delete` dove c'è un distruttore

3.6 Prestazioni

- Una gestione poco attenta dello heap (alternanza di `new` e `delete`) può portare alla sua **frammentazione**: la formazione di buchi nella memoria disponibile.
- Questi buchi sono troppo piccoli per poter essere utilizzati per allocare nuovi oggetti.
- L'effetto è che la memoria che resta disponibile è molto meno di quello che ci si aspettava.
- Ne consegue anche **l'aumento dei tempi** di esecuzioni di nuovi `new`: diventa sempre più difficile andare a cercare dove poter mettere il nuovo oggetto.
- Cerchiamo di capire come funziona questo meccanismo per cercare di prevenirlo

```
1 while(/* ... */){
2     Msg *p = create();
3     // ...
4     Node *n1 = new Node;
5     // ...
6     delete p;
7     Node *n2 = new Node;
8 }
```

- In questo esempio si vede benissimo che ogni volta che cancello un messaggio e alloco un nodo si crea un buco poiché il nodo non è grande quanto la memoria libera creando un buco che non può essere riempito da messaggio. reiterando questo blocco di codice avremo sempre piu buchi non utilizzabili

3.7 Soluzioni alla framenttazione

- Due alternative:
  - Un **garbage collector** per tappare i buchi
  - Il programmatore evita di crearli
- I puntatori rendono difficile creare un **garbage collector**
  - poiché è difficile spostare le locazioni di memoria al interno della memoria rispetto ad altri linguaggi come **java** dove si posso spostare gli oggetti al interno della memoria più facilmente
- Come possiamo evitare la formazione dei buchi?
- A volte basta riorganizzare la chiamata alle `new` e `delete`
- Ma non è una soluzione generale
- **Prevenire:** evitare usi del free store che provocano frammentazione
- **Prima idea:** evitiamo l'uso del `delete`, almeno non rallentiamo nuovi `new`, almeno nella maggior parte delle implementazioni, anche se non è garantito dallo standard.

- **Seconda idea:** allochiamo tutta la memoria (statica e globale) all'inizio del programma e poi la usiamo.

  Svantaggi: struttura del programma non ottimale e uso di dati globali da evitare.

- Due **strutture** dati possono **aiutarci** in questo:
  - **Stack** : visto che si allunga e si accorcia solo da una parte non può causare frammentazione
  - **Pool** : una raccolta di oggetti della stessa dimensione. Essendo della stessa dimensione, non può esserci frammentazione.
- Con entrambe queste soluzioni, sia **l'allocazione** che la **deallocazione** hanno tempi prevedibili e veloci. Se non si gestisce la frammentazione potremo avere tempi lunghi di allocazione e non prevedibili

4 Tipi definiti dal utente

Abbiamo visto finora dei tipi **built-in(integrati)** che sono di molto **basso livello** è sono difficili per un programmatore gestirli e creare applicazioni di alto livello.
per questo C++ mette a disposizione **i meccanismi d'astrazione** con cui so costruiscono strutture di **alto livello**. Queste strutture vengono chiamate **tipi definiti dai utenti(user-defined types)**

Questi tipi definiti dai utenti sono:

- Classi
- Enumeratori
- Struct

4.1 Esempi di user-defined types

Un Esempio di tipo di dato definito dal utente è un vector:

```
1  struct Vector{
2  int sz; // numero di elementi
3  double* elem; // punt. ad array di elementi
4  };
5
6  Vector v; //creazione di una varabile di tipo vector
7
8  void vector_init(Vector& v, int s){ // definizione di funzione
                                       //per l'inizializzazione di un vecotor
9  v.elem = new double[s];
10 v.sz = s;
11 }
```

Questa è la prima versione di una migliore gestione del vettore.

Abbiamo:

- **Primo blocco** definiamo un tipo vector
- **Secondo blocco** creiamo una variabile di tipo vector
- **Terzo blocco** definiamo una funzione per l'inizializzazione
- Allochiamo l'array elem sullo heap.
- Si noti che il primo argomento (`Vector& v`) è passato per riferimento, in modo da poterlo modificare

Un utilizzo di questo **vector** può essere il seguente:

```
1  double read_and_sum(int s)
2  //legge s interi da cin e restituisce la somma
3  // s positivo
4
5  {
6      Vector v;
7      vector_init(v,s); // allocate s elements for v
8
9      for (int i=0; i!=s; ++i)
10     cin>>v.elem[i]; // read into elements
11
12     double sum = 0;
13     for (int i=0; i!=s; ++i)
14     sum+=v.elem[i]; // take the sum of the elements
15
16     return sum;
17 }
```

NB: in questo caso Il programmatore deve conoscere le strutture interne dell'oggetto

4.2 Classi

Una **classe** è un insieme di:

- Attributi
- Metodi

L'interfaccia della classe è data dalla **visibilità** dei suoi metodi e attributi(quindi da quelli **pubblici**). abbiamo anche dei attributi o metodi **privati** che sono visibili solo al interno della classe.

```
1  class Vector{
2  public:
3      Vector(int s) :elem{new double[s]}, sz{s} {
4          } // costruttore
5
6      double& operator[](int i) { return elem[i];} //nuovo operatore
7
8      int size() { return sz; }
9
10     private:
11         double* elem;
12         int sz;
13 };
14
15 Vector v(6);
```

- Abbiamo definito un costruttore che assegna alla variabile locale `elem` un puntatore a un vettore di `double` lungo **"S"** e a `sz` il numero **"S"**. Possiamo notare che per assegnare questi valori non c'è bisogno di fare inizializzazioni al interno del corpo del costruttore.
- abbiamo definito un **nuovo operatore** `[]`. questo operatore prenderà come input un `int i` e restituisce un **riferimento** a `double`

Quindi questa **classe** può avere viarie **in stanze** che potranno essere diverse fra di loro. Una cosa che le accomuna sarà la loro **struttura** e il loro **peso** che sarà uguale per tutti i tipi `Vector`.

- **La struttura** interna del **vettore** rimane nascosta(in questo caso).
- Notate come l'utilizzo del costruttore risolve il problema.

A cosa serve una classe ?

- Serve ad **astrarre** ancora di più un **oggetto**, rendendolo ancora più **generale** e più facile da utilizzare
- Infatti con le **struct** dobbiamo necessariamente sapere come è **formata** al suo interno , altrimenti non possiamo utilizzarla.
- con le **classi** non c'è bisogno di questo poiché ci sono **l'interfaccie**.
- **l'interfaccia** ci permette di utilizzare la **classe** anche se non sappiamo come è fatta veramente.

4.3 Enumerazioni

L'enumerazioni vengono usate per rappresentare piccoli insiemi di valori interi.
Esistono due tipi di enumeratoti: i `enum class` e i `enum`

4.3.1 Enum class e Enum

- Questi interi vivono all'interno dello **scope** (ambito di definizione) della loro **enum class**, cosicché lo stesso valore può venir usato in **enum class** senza confusione.
- Quindi possiamo vedere che il colore `red` di `color` non va in conflitto con quello di **Traffic_light** perché sono in **scope** diversi

```
1  enum class Color {red,blue,green};
2
3  enum class Traffic_light {green,yellow,red};
4
5  Color col=Color::red;
6
7  Traffic_light light=Traffic_light::red;
8
9  //Errori
10
11 Color x=red; // NO: quale red?
12 Color y=Traffic_light::red; // NO: tipo sbagliato!
13 Color z=Color::red; // OK
14
15 int i=Color::red; // No, tipo sbagliato
16 Color c=2; // No, tipo sbagliato
```

```
enum class Warning : char { green, yellow, orange, red };

//di default i valori dei enumaratori sono assegnati in modo
// crescente da 0
static_cast<int>(Warning::green)==0
static_cast<int>(Warning::yellow)==1
static_cast<int>(Warning::orange)==2
static_cast<int>(Warning::red)==3

// Se inzializzo un valore
enum class Warning{ green=12, yellow, orange, red };

static_cast<int>(Warning::green)==12
static_cast<int>(Warning::yellow)==13
static_cast<int>(Warning::orange)==14
static_cast<int>(Warning::red)==15
```

- La parola chiave **`class`** che segue **`enum`** specifica che:
  - questa enumerazione è **fortemente tipata** quindi non avvengono conversioni implicite
  - i suoi **enumeratori** hanno un ambito di definizione
- se togliamo **`class`** i valori saranno convertiti in interi normali implicitamente

```
enum Traffic_light { red, yellow, green };
enum Warning { green, yellow, orang e, red };

void f(Traffic_light x)
{
    if (x == 9) { /* ... */ }
                          // OK (but Traffic_light doesn't have a 9)

    if (x == red) { /* ... */ } // error : two reds in scope

    if (x == Warning::red) { /* ... */ } // OK (Ouch!)

    if (x == Traffic_light::red) { /* ... */ } // OK
}
```

4.3.2 Operaotri per `enum class`

- Per default, un enum class ha solo assegnamento, inizializzazione e confronto `(==,<,::)`
- Tuttavia, essendo un tipo definito dall'utente, si possono definire anche dei nuovi operatori:

```
Traffic_light& operator++(Traffic_light& t){
// prefix increment ++, mi restituisce il successivo poiche' incremento
    switch(t){
        case Traffic_light::green: return t=Traffic_light::yellow;
        case Traffic_light::yellow: return t=Traffic_light::red;
        case Traffic_light::red: return t=Traffic_light::green;
    }
}
```

5 iostream

- **iostram** è libreria standard permette di formattare i caratteri in
- **input** e in **output**.
- Le operazioni di input sono tipizzate ed estendibili cosi permettendo di gestire i tipi definiti dall'utente.
- Altri tipi di interazione con l'utente (grafico, ad esempio) non sono inclusi nello standard **ISO**(International Organization for Standardization) e quindi nemmeno in questa libreria. **ISO** è un organizzazione di standardizzazione, per non creare anarchia.

5.1 Output

- Un **ostream** converte un oggetto con tipo in uno **stream** di caratteri (bytes).
- Si può costruire l'output per ogni tipo definito dall'utente.
- Operatore « ("Output") definito per tutti gli oggetti di tipo **ostream** (es: lo standard output `cout`, lo standard error `cerr`, non bufferizzato o `clog`, bufferizzato).
- Per default, i valori scritti su uno stream di uscita sono convertiti in una sequenza di caratteri
- esempio: l'intero 10 sarà convertito nella sequenza '1','0'.
- Ogni operazione di « restituisce lo stream, di modo da poter concatenare diverse operazioni

```
int i=23;
cout << "Risultato_" << i << '\n';
```

- Un carattere che viene convertito ad intero sarà convertito nella nel valore della tabella ASCII.

```
void k()
{
    int b = 'b'; // note: char implicitly converted to int
    char c = 'c';
    cout << 'a' << b << c;
}
```

NB: `'b'` sarà convertito un ASCII quindi l'out sarà `a98c`

5.2 Input

- Un `istream` converte uno stream di caratteri in oggetti con tipo.
- `istream` per input di tipi `built-in`, estendibile a tipi definiti dall'utente.
- Operatore » ("metto in") definito sugli oggetti di tipo istream, tra cui `cin`.
- Il tipo dell'operando a destra di » determina:
  - quale ingresso accetta;
  - l'obiettivo dell'operazione.

```
// legge un intero e lo mette in i
int i;
cin >> i;
// legge un double e lo mette in d
double d;
cin >> d;
```

- Per leggere una sequenza di caratteri useremo `string`; tuttavia la lettura si ferma al primo carattere non alfanumerico:

```
// legge un intero e lo mette in i
string str; // std::string
cin >> str;
```

- Per leggere un'intera linea dobbiamo usare `getline()` (vedi esempio)
- Per leggere un solo carattere `get()`

5.3 iostream User-Defined Types

- La libreria `iostream` permette di definire operazioni di I/O per i tipi definiti dall'utente

5.3.1 Esmpio con Stream di output

```
1  struct Cliente{
2  string nome;
3  int numero;
4  };
5
6  ostream& operator<<(ostream& os, const Cliente& e){
7      return os << "{\"" << e.nome << "\"," << e.numero << "}";
8  }
9
10 int main(){
11     Cliente &c = *(new Cliente);
12     c.nome = 'a';
13     c.numero = 2;
14     cout <<  "C" <<c;  //cominciera' a stampare
15                        //da sinistra verso destra
16 }
```

- Nella definizione dell'operatore « per il nuovo tipo, lo stream di uscita
- viene preso, per riferimento, come primo argomento
- viene restituito come risultato

5.3.2 Esmpio con Stream di Input

```
struct Cliente{
    string nome;
    int numero;
};

istream& operator>>(istream& is, Cliente& e)
// read { "name" , number } pair. Note: for matted with { " " , and }
{
    char c, c2;
    if (is>>c && c=='{' && is>>c2 && c2=='"') { //legge finche'
                                      //non incontra '{' e '"'

        string name;   // il valore di default di una stringa e': ""

        while (is.get(c) && c!='"'){ // legge fino all secondo '"'
                                      //quindi avra letto tutto il nome
            name+=c;
        }
        if (is>>c && c==',') {   //legge fino alla virgola
            int number = 0;

            if (is>>number>>c && c=='}') { // legge il numero e'
                                        //subito dopo la '}'

                 e = {name ,number}; // assegna alla varabile Cliente
                    i valori

                return is;              //ritorna lo stream
            }
        }
    }
    return is;
}
```

5.4 Errori iostream

- Un iostream si può trovare in uno tra quattro stati:
  - **`good()`** : la precedente operazione di **iostream** ha avuto successo
  - **`eof()`** : arrivati alla fine dell'ingresso **("end-of-file")**
  - **`fail()`** : qualcosa di inaspettato (per esempio, mi ha aspetta una cifra e ho trovato un carattere
  - **`bad()`** : qualcosa di seriamente inaspettato (ad esempio, un errore nella lettura del disco)
- Qualsiasi operazione venga tentata su di uno stream che non termina con uno stato `good()` non ha nessun effetto.
- Un iostream può venire usato come condizione: `true` solo se si trova nello stato `good()`
- Dopo un errore di lettura, per pulire lo stream e procedere:

```
int i;
if(cin>>i){
    // fa quello che devi
    } else if(cin.fail()){
        cin.clear();
        string s;
        if(cin>>s){ // vediamo cosa contiene
            // procedi
        }
}
```

- In alternativa, possiamo usare le eccezioni . . .

6 String e Random

6.1 String

La **standard library** mette a disposizione il tipo `stringa` per completare i letterali `stringa`.
il tipo stringa fornisce utili **operatori** come:

- **la concatenazione(+):**

```
string compose(const string& name, const string& domain)
{
    return name + '@' + domain;
}

auto addr = compose("dmr","bell-labs.com"); //auto detecta in
                                            //automatico il tipo
```

  - **NB: auto** è una parola chiave che detecta il automatico il tipo che stiamo assegnato alla variabile `addr`.
  - Perché non usarlo sempre ? Altrimenti cererebbe ambiguità e difficoltà nella ricerca di eventuali errori

- **Operatore (+=)**

```
string addNewline(string& s1, string& s2){
    s1 = s1 + '\n';
    s1 += '\n';
}
```

Una **stringa** è `mutabile` quindi può cambiare nel tempo.
possiamo manipolarla utilizzando anche l'operatore `[]`.

Ecco alcuni esempi:

```
string name = "Niels_Stroustrup";
void func(){
    string s = name.substr(6,10); //"Stroupstrup"
    name.replace(0,5,"nicholas");
    name[0] = toupper(name[0])  //in posizione ci sara'
                                //la lettera in maiuscolo
}
```

- `sustr()` restituisce una string che è una copia della stringa indicata dagli argomenti (inizio e lunghezza).
- `replace()` sostituisce una sotto stringa con una stringa, anche di lunghezza diversa.
- Nei confronti tra stringhe o con letterali stringa si considera l'ordine lessicografico.

6.1.1 Costruttori

- **Costruttori senza puntatori:**

```
string s0; // la stringa vuota
string s1("Inizializzazione semplice");
string s2(s1); // copia una stringa
string s3(7,'a') // "aaaaaaa"
string s4(0) // in C ok, in C++ Pericoloso!
```

- **Costruttore con puntatori:**

```
string s5(nullptr); // pericolosissimo!
string s6(p); // dipende dal valore di p
string s7{"OK"}}; // OK puntatore a una
stringa in stile C
```

- **allucini metodi utili:**
  - **`s.size()`**: numero di caratteri in `s`
  - **`s.length()`**: lo stesso
  - **`s.clear()`**: per pulire
  - **`s.empty()`**: `bool`, è vuota?
  - **`s.front()`**: `s[0]`
  - **`s.back()`**: `s[s.size()-1]`

6.2 Random

La libreria `<radom>` ci permette di generare numeri **pseudo-casuali** tramite l'ausilio di due parti:

- Il **motore(engine)** che genera i numeri **pesudo-casuali** secondo una distribuzione.
- una **distribuzione** quindi come sono distribuiti i valori, di che tipo sono e l'intervallo. (es: `uniform_int_distribution`, `normal_distribution`, `exponential_distribution`, . . .)

```
1  // uso un alias per leggibilita '
2  using my_engine=default_random_engine;
3  using my_distribution= uniform_int_distribution <int>;
4
5  my_engine re {};
6  my_distribution one_to_six {1,6};
7  auto dado = bind(one_to_six,re);
8
9  // per usare il generatore dato
10 int x=dado();
```

NB: Per far funzionare quesro codice c'è bisogno della libreria `<radom>` e `<functional>`

7 Puntatore a funzione

- Come un qualsiasi dato, anche il body di una funzione viene **memorizzato in memoria**, e quindi ha un **indirizzo**.
- Ne consegue che possiamo avere **puntatori** a funzione esattamente come abbiamo puntatori ad un oggetto.
- Ma **non** possiamo modificare il codice di una funzione accedendovi attraverso il puntatore.
- Ci sono solo due cose che possiamo fare con una funzione:
  1. **chiamarla**
  2. **prenderne l'indirizzo**
- Il puntatore ottenuto prendendo l'indirizzo di una funzione può quindi essere utilizzato per chiamarla.

```
1  void error(string s) { /* ... */ }
2
3  void (*f1)(string) = &error; // OK: same as = error
4  void (*f2)(string) = error; // OK: same as = &error
5
6  void g(){
7      f1("Vasa"); // OK: same as (*f1)("Vasa")
8      (*f1)("Mary_Rose"); // OK: as f1("Mary Rose")
9  }
10
11 void (*pf)(string); // pointer to void(str ing)
12 void f1(string); // void(str ing)
13 int f2(string); // int(string)
14 void f3(int*); // void(int*)
15
16 void f(){
17     pf = &f1; // OK
18     pf = &f2; // error : bad return type
19     pf = &f3; // error : bad argument type
20     pf("Hera"); // OK
21     pf(1); // error : bad argument type
22     int i = pf("Zeus"); // error : void assigned to int
23 }
```

- Nel caso di puntatore a funzione, il dereferenziamento è opzionale (sia nell'assegnamento che nella chiamata)
- Ovviamente gli argomenti dichiarati per un puntatore a funzione seguono esattamente la stessa sintassi di quelli delle funzioni.
- Posso assegnare a un puntatore a funzione un altra funzione solo se quest'ultima e la stessa **segnature** del puntatore a funzione.

```
//alias utilizzato per p1 e p2
using P1 = int(*)(int*);
using P2 = void(*)(void);
```

- ho **rinominato** un puntatore a funzione che ritorna un intero e prende in input un intero come **P1**
- ho **rinominato** un puntatore a funzione che non prende nulla in input e non restituisce nulla come **P2**
- posso assegnare una funzione a questi due puntatori solo se la **firma** e la funzione combaciano

Un esempio di utilizzo di ptr a funzione: È un buon modo per parametrizzare gli algoritmi, ad esempio, ad un algoritmo di ordinamento posso passare la funzione che confronta due elementi. Cosi se volessi cambiare il funzionamento del confronto mi basterebbe passare come parametro L'Indirizzo di un altra funzione.

```
using ConfrontoInt = bool(*)(int, int);

void AlgoritmoX(ConfrontoInt f, int eta1, int eta2){
        if(f(eta1, eta2)){
            cout << "Ciao!";
        }
}

using ConfrontoInt = bool(*)(int, int);

bool Pippo(int a, int b){
    if(a>b){
       return true;
    }
    return false;
}

int PippoCattivo(int a, int b){
    if(a>b){
       return 1;
    }
    return 2;
}

int main(){
    AlgoritmoX(Pippo,2,1);          //OK!!
    AlgoritmoX(PippoCativo,2,1); //error: siganture  diversa del tipo
                                 //ConfrontoInt con PippoCattivo
}
//OUT = ciao
```

8 Eccezioni

Le **eccezione** provvede a dare delle informazioni sul errore. Lo scopo delle **eccezioni** è quello di portare l'informazione dal punto in cui un errore è stato **individuato** a dove può essere **gestito**. Questa **eccezione** deve essere gestita da qualcuno. solitamente non si può gestisce nella funzione in cui viene generata(si può ma qui ci concerteremo sul non farlo). Per questo motivo la funzione lancia(**throws**) l'eccezione. Sarà la funzione chiamante ad occuparsi del eccezione.

- le **librerie**: al momento della loro implementazione, lo sviluppatore può non conoscere nemmeno il tipo di programmi in cui verranno utilizzate.
  - Un errore può venir individuato all'interno di una libreria, ma lì non si sa come trattarlo (poiché non si sa dove sarà utilizzata la libererai). quindi si lancerà un **eccezione**, cosi sarà L'Utilizzatore della libreria a gestire l'errore a meglio, in base alle sue esigenze.
  - **L'utente** della libreria sa cosa fare **dell'errore**, ma non come individuarlo (altrimenti lo avrebbe già fatto fuori della libreria). per questo gestisce l'errore in un luogo separato perché è più semplice che identificarlo
- Ovviamente questo discorso è valido di **programmi grossi** e strutturati la cui esecuzione si prolunga nel tempo.
- Per catturare(**catch**) queste eccezioni utilizzeremo il blocco `try-catch`
  - Abbiamo un blocco `try` dove ci sono l'Istruzioni da eseguire, nel caso in cui avvenisse un errore(è stata lanciata un eccezione) ci saranno vari blocchi `catch` per catturare L'Eccezione. Saranno catturate solo quelle specificate.
  - invece per lanciare un eccezione si userà la parola chiave `throws`. utilizzato dalle funzioni che non posso gestire l'errore

```
1  void taskmaster()
2  {
3      try {
4          auto result = do_task();
5          // use result
6      }
7      catch (Some_error) {
8          // falisce di fare do_task: gestisce il probelma
9      }
10 }
11
12 int do_task(){
13     // ...
14     if (/* could perfor m the task */)
```

```
15          return result;
16      else
17          throw Some_error{};
18  }
```

- l'eccezione Può **essere di qualsiasi tipo** che ammetta la copia, ma è raccomandabile usare un tipo definito apposta, in modo da minimizzare il **clash**(scontrarsi) tra errori lanciati da librerie diverse. (es. due eccezioni che usano il tipo interno e utilizza lo stesso numero per errori diversi)
- La libreria `std` definisce una piccola gerarchia di eccezioni
- Un'eccezione può trasportare informazioni sull'errore che rappresenta.

8.1 Perché utilizzare L'Eccezione ?

- Noi utilizziamo gli L'Eccezione perché il modo tradizionale di gestione del errore non è efficiente e potrebbe portare a soluzione non corrette o mal funzionanti
- **caratteristiche gestione errori tradizionale:**
  - **Terminare il programma**
  - **Restituire un valore di errore** non sempre possibile (nessun valore disponibile)
  - **Stato di errore** Restituire un valore legale, ma lasciare il programma in uno stato di errore (occorre controllare esplicitamente)
  - Funzione che gestisce l'errore (si rimanda a come la funzione gestisce l'errore)
- **caratteristiche gestione errori con eccezioni:**
  - **Codice più leggibile** poi si può separare il codice ordinario dal quello per la gestione dei errori.
  - **il codice ordinario** rileva l'errore e lancia l'eccezione.

8.2 Gerarchie degli errori della Liberia standard

- `logic_error` possono essere individuati o prima che cominci l'esecuzione o attraverso dei test sugli argomenti di funzioni e costruttori.
  - `length_error`
  - `domain_error`
  - `out_of_range`
  - `invalid_argument`
  - `future_error`

- `runtime_error` tutti gli altri
  - `range_error`
  - `overflow_error`
  - `underflow_error`
  - `system_error`
- `bad_exception`
- `bad_alloc`
- `bad_typeid`
- `bad_cast`

8.3 RAII(acquisizione delle risorse è l'inizializzazione)

- C++ non ha un **garbage collecotr** quindi dobbiamo essere noi ad occuparci della **de-allocazione** dei oggetti allocati nel **heap**.
- Quando una risorsa è **troppo grande** per lo **stack** si alloca nel heap. avremo un **Proprietario** cioè la classe e incapsulato dentro di essa la risorsa
- essendo la classe proprietario di quella risorsa deve essere lei(calsse) a deallocare la risorsa quando non ci sarà più bisogno di quella risorsa

```
1  class widget
2  {
3  private:
4      int* data;
5  public:
6      widget(const int size) { data = new int[size]; } // acquisisce
7      ~widget() { delete[] data; } // release
8      void do_something() {}
9  };
10
11 void functionUsingWidget() {
12     widget w(1000000);  // lifetime automatically
13                         //quando si chiude lo scope
14                         // costruttore di w, include the w.data member
15     w.do_something();
16
17 } // automatic destruction and deallocation for w and w.data
```

NB: w è l'oggetto allocato nello stack, quando esce dal suo scope viene deallocato. Quando viene deallocato l'oggetto richiama il suo distruttore.

- Ogni risorsa viene **incapsulata** in una **classe**, in cui
  - il costruttore **acquisisce** la risorsa, e lancia un'eccezione se non ci riesce
  - il distruttore **rilascia** la risorsa senza mai lanciare eccezioni

9 Classi

Una classe può essere descritta brevemente come:

- **La classe** è un tipo definito dal utente(user-defined type)
- **Una classe** consiste in membri:
  - **Metodi** (function member)
  - **Attribuiti** (Data member)
- **I metodi** possono definire varie **funzionalità** come L'Inizializzazione(costruttore) , la copia, la pulizia (distruttore) ecc.

Alcune caratteristiche sono:

- Attributi e metodi si possono richiamare con **.**(dot) per i **puntatori** con **->**(arrow)
- gli **operatori** come **+**,**!** e **[]** possono essere ridefiniti
- La classe è un **namespace** che contiene i Membri(meber)
- **Le classi** offrono anche una **visibilità** dei loro membri tramite le seguenti parole chiavi:
  - **private:** Visibile solo nella classe di definizione.
  - **protected:** visibile anche ai figli.
  - **public:** visibile a tutti.
- Senza un **esplicita** definizione di **visibilità** sarà di **defualt** public

```
class X {
    private: // the representation (implementation) is private
        int m;
    public: // the user interface is public
    X(int i =0) :m{i} { } // a constructor (initialize the data member
         m)
    int mf(int i){// a member function
        int old = m;
        m = i; // set a new value
        return old; // return the old value
    }
};
X var {7}; // a variable of type X, initialized to 7
int user(X var, X* ptr){
    int x = var.mf(7); // access using . (dot)
    int y = ptr->mf(9); // access using -> (arrow)
    int z = var.m; // error : cannot access private member
}
```

Cos'è il **namespace**? è un blocco di codice con un nome

```
namescpace lasd{
    \\codice
}

Class X{
    \\codice
}
```

Come si utilizza ?

```
namescpace lasd{
    int X(int i){
        \\codice
    }

}

namescpace Prova{
    int X(int i){
        \\codice
    }

}

int main(){

    lasd::X(1);    //richiama X di lasd
    Prova::X(1); //richiama X di prova
    return 0;
}
```

Perché usarlo? Si utilizza perché ci potrebbero essere delle stesse funzioni col lo stesso nome ma inserendole in **namespace** diversi non andremo in conflitto, basterà specificare il **namespace** a cui si fa riferimento.

Copia di un oggetto(istanza della classe):

Per default gli oggetti possono venir copiati: quindi un'istanza di una classe può venir inizializzata con con una copia di un'altra istanza.

```
class Date{
    int d,m,y;
    public:
        void init(int dd, int mm, int yy);
        void add_year(int n);
        void add_month(int n);
        void add_day(int n);
}

Date my_birthday;
my_birthday.init(14,1,1988);
Date d1 = my_birthday;
```

NB: d1 è una copia del istanza di nome `my_birthday`

- **Per default**, la copia di un oggetto corrisponde alla copia di ogni singolo membro.
- Se si vuole che il **comportamento** sia diverso da questo, è necessario **definire** opportunamente la **classe**.
- Questo naturalmente vale anche per le **istanze** di una classe: l'assegnamento significa la copia **dell'istanza**.
- Anche qui, copia significa copia di ogni singolo membro, ma se si vuole un comportamento diverso basta ridefinire l'operatore di assegnamento.

9.1 Costruttore

Utilizzare init e poco elegante e soggetta ad errori per questo motivo ci sono **costruttori** sono dei metodi che hanno lo stesso nome della classe. **Il costruttore** impone delle **invarianti di classe** che tutte le **istanze** di quella classe devono avere. se questa proprietà non viene rispettata sarà **lanciata un eccezione**.

```
class Date {
    int d, m, y;
    public:
        Date(int dd, int mm, int yy); // constructor
        // ...
};

Date today = Date(23,6,1983);
Date xmas(25,12,1990); // abbreviated for m
Date my_bir thday; // error : initializer missing
Date release1_0(10,12); // error : third argument missing

Date today = Date {23,6,1983};
Date xmas {25,12,1990}; // abbreviated for m
Date release1_0 {10,12}; // error : third argument missing
```

NB: è consigliabile inzializare con le parentesi **{}** invece di **()** per i motivi descritti in **Inizializzazione(2.2, clicca)**

- Di solito abbiamo più costruttori, con **signature** diverse tramite **l'overloading**.
- **L'overloading** può essere fatto purché numero e/o tipo dei parametri sia diverso.
- Per non esagerare nel numero di parametri si può utilizzare il valore di default per uno o più parametri.

```
class Date{
    int d {today.d};
    int m {today.m};
    int y {today.y};
    public :
        Date(int dd):d{dd} {
            // controlla se la data e' valida
        }
        //overloading
        Date(int dd, int ddd):d{dd} :m{ddd}{
            // controlla se la data e' valida
        }
}
```

- Un **costruttore** può tuttavia inizializzare **uno o più** attributi
- dopo aver usato il primo **costruttore** il valore di d è `dd`, mentre `m` e `y` valgono rispettivamente `today.m` e `today.y`
- in questo esempio in oltre `d, m, y` sono già inizializzati di default

9.2 Definizione metodi

- Due **possibilità** per i metodi:
  1. **Dichiararli** dentro definizione della classe e **definirli** fuori.
  2. **Dichiararli** e **definirli** dentro la definizione della classe.
- **Preferibile il primo**, salvo che il metodo **sia piccolo**, modificato raramente e usato spesso: infatti viene considerato come `inline`.
- Ogni membro della classe può accedere a qualsiasi altro membro della classe indipendentemente da dove è stato definito: in altre parole, dichiarazione e definizione dei membri di una classe non dipendono dall'ordine.

```
class Date{
    public:
        void add_month(int n){m+=n;}
        // ...
    private:
        int d,m,y;
};

class Date{
    public:
        void add_month(int n);
        // ...
    private:
        int d,m,y;
    };

inline void Date::add_month(int n){ //definizone nella classe
                                    // dichiarazione fuori dalla classe
    m+=n;
}
```

9.3 inline

La **inline** parola chiave suggerisce che il **compilatore** sostituisce il codice all'interno della **definizione** della funzione al posto di ogni **chiamata a tale funzione**.

l'uso di funzioni **inline** può rendere il programma più veloce perché eliminano il sovraccarico associato alle **chiamate di funzione**. La chiamata a una funzione richiede il **push** dell'indirizzo restituito nello **stack**, il **push** degli argomenti nello **stack**, il passaggio al corpo della funzione e l'esecuzione di un'istruzione**return** al termine della funzione. Questo processo viene eliminato tramite **l'inlining** della funzione.

```
// account.h
class Account
{
public:
    Account(double initial_balance)
    {
        balance = initial_balance;
    }

    double GetBalance() const;
    double Deposit(double amount);
    double Withdraw(double amount);

private:
    double balance;
};

inline double Account::GetBalance() const
{
    return balance;
}

inline double Account::Deposit(double amount)
{
    balance += amount;
    return balance;
}

inline double Account::Withdraw(double amount)
{
    balance -= amount;
    return balance;
}
```

9.4 Funzioni costanti

- Si può aggiungere ai metodi la parola chiave per sottolineare che si tratta di metodi che non vanno a modificare lo stato dell'oggetto.

```
1  class Date{
2      int d,m,y;
3      public:
4          int day() const {return d;}
5          int month() const {return m;}
6          int year() const;
7          // ...
8      };
9
10 void Date::year() const {
11     return y;
12 }
```

- La parola chiave `const` è **obbligatoria** anche quando il metodo viene definito fuori della classe: fa parte della **signature** del metodo.
- Se l'oggetto è costante, non permette l'invocazione di metodi non dichiarati come costanti!
- Un metodo costante, ovviamente, può venire invocato anche su oggetti non costanti.

9.5 Static

- Le classi possono contenere membri dati **`static`i** e funzioni membro **`static`he**. Quando un membro dati viene dichiarato come **`static`**, viene mantenuta una sola copia dei dati per tutti gli oggetti della classe .
- Vale sia per gli **attributi**, che per **i metodi** (che, ad esempio, vanno a modificare gli attributi statici).
- I membri dati **`static`i** non fanno parte degli oggetti di un tipo **specifico** della **classe**. Di conseguenza, la dichiarazione di un membro dati statico non è considerata una definizione. Il membro dati viene dichiarato **nell'ambito della classe**, ma la definizione viene fatta **nell'ambito del file**. Questi membri **`static`i** hanno collegamento esterno. L'esempio seguente illustra questi concetti.

```cpp
// static_data_members.cpp
class BufferedOutput
{
public:
   // Return number of bytes written by any object of this class.
   short BytesWritten()
   {
      return bytecount;
   }

   // Reset the counter.
   static void ResetCount()
   {
      bytecount = 0;
   }

   // Static member declaration.
   static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;

int main()
{
}
```

- Nel codice precedente, il membro `bytecount` è dichiarato nella classe `BufferedOutput`, ma deve essere definito all'esterno della dichiarazione della classe.
- **Ai membri** dati **`static`** è possibile accedere senza fare riferimento a un oggetto di tipo classe. Il numero di **byte** scritti utilizzando oggetti `BufferedOutput` può essere ottenuto come segue:

```cpp
long nBytes = BufferedOutput::bytecount;
```

9.6 Distruttore

È importante **controllare** che tutte le risorse vengano **rilasciate**.
Questo è tra i compiti del **distruttore**.
Il distruttore ha lo stesso nome della classe ma prima del nome ha ~(tilde)

- Il distruttore viene chiamato
  - **implicitamente** quando l'oggetto esce dal suo ambito di definizione(**scope**)
  - **esplicitamente** con **`delete`**

```
class Date{
    int d {today.d};
    int m {today.m};
    int y {today.y};
    public :
        Date(int dd):d{dd} {
            // controlla se la data e' valida
        }
        //overloading
        Date(int dd, int ddd):d{dd} :m{ddd}{
            // controlla se la data e' valida
        }

        //distruttore
        ~Date(){
            // Libera risorse
        }

}
```

9.7 Ereditarietà o sottoclassi

Per indicare che `Employee` è una superclasse di Manager:

```
class Manager : public Employee{
    // ...
}
```

- Costruzione **bottom-up** (dalla classe base verso le derivate) e distruzione **top-down** (in senso inverso).
- I **costruttori non vengono** automaticamente ereditati (devono cambiare per forza di cose!)

9.8 Virtual

- Il `virtual` permette di **ridefinire** un metodo nelle proprie **sottoclassi**. `virtual` serve per creare un **interfaccia** cioè una struttura comune tra tutte le **classi** che la estenderanno. Visionando questa **interfaccia** possiamo sapere che metodi implementa una certa classe che estende questa **interfaccia**. questo concetto si rafforza quando la classe è **astratta**
- Aggiungere la pseudo inizializzazione `= 0` a un **metodo** lo farà diventare un `pure virtual` (non ha implementazione e deve essere quindi ridefinito).
- Una classe con una o più metodi virtuali puri si dice `astratta`: non si possono creare oggetti di una **classe astratta**.
- Se invece si aggiunge `final` dopo la dichiarazione quel metodo non può più venir ridefinito.
- Se una classe è **Astratta** quindi ha almeno un `Virtual` puro , allora ogni classe che la estende dovrà avere un implementazione per ogni metodo virtual puro

```
// Classe base
class Base {
public:
    // Funzione virtuale
    virtual void stampa() {
        std::cout << "Stampa dalla classe Base" << std::endl;
    }
};

// Classe derivata
class Derivata : public Base {
public:
    // Sovrascrittura della funzione virtuale
    void stampa() override {
```

```
15          std::cout << "Stampa dalla classe Derivata" << std::endl;
16      }
17  };
```

- quando si usa `virtual` e si implementa un metodo deve essere molto breve è semplice.
- Se dichiari un metodo **senza virtual**, le classi derivate possono comunque definire un metodo con lo stesso nome. Tuttavia, la chiamata al metodo dipenderà dal tipo del puntatore o della referenza utilizzata per fare la chiamata, non dal tipo dell'oggetto a cui il puntatore o la referenza puntano.
- Dichiarare il **metodo come virtual** consente il polimorfismo, ovvero il comportamento in cui la chiamata al metodo è determinata dal tipo effettivo dell'oggetto a cui il puntatore o la referenza puntano.

9.9 Template

```
// Funzione template per lo scambio di due valori di tipo generico
template <typename T>
void scambia(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    // Esempio di utilizzo della funzione scambia con interi
    int x = 5, y = 10;
    std::cout << "Prima dello scambio: x = " << x << ", y = " << y << std::
        endl;
    scambia(x, y);
    std::cout << "Dopo lo scambio: x = " << x << ", y = " << y << std::endl
        ;

    // Esempio di utilizzo della funzione scambia con double
    double a = 3.14, b = 6.28;
    std::cout << "Prima dello scambio: a = " << a << ", b = " << b << std::
        endl;
    scambia(a, b);
    std::cout << "Dopo lo scambio: a = " << a << ", b = " << b << std::endl
        ;

    return 0;
}
```

- in questo caso il **compilatore** quando andremo a inserire i parametri**attuali** andrà a sostituire a `T` il tipo corretto rispetto al **tipo** del parametro attuale.
- Un tipo o un valore diventa un parametro nella definizione di una classe, una funzione o un alias di tipo.

```cpp
// Definizione della classe template Pair
template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;

public:
    // Costruttore
    Pair(const T1& f, const T2& s) : first(f), second(s) {}

    // Metodo per ottenere il primo elemento della coppia
    T1 getFirst() const {
        return first;
    }

    // Metodo per ottenere il secondo elemento della coppia
    T2 getSecond() const {
        return second;
    }
};

int main() {
    // Creazione di una coppia di interi e una coppia di double
    Pair<int, double> pair1(10, 3.14);
    Pair<std::string, char> pair2("hello", 'A');

    // Stampare i valori della coppia di interi
    std::cout << "Primo valore di pair1: " << pair1.getFirst() << std::endl;
    std::cout << "Secondo valore di pair1: " << pair1.getSecond() << std::endl;

    // Stampare i valori della coppia di stringa e char
    std::cout << "Primo valore di pair2: " << pair2.getFirst() << std::endl;
    std::cout << "Secondo valore di pair2: " << pair2.getSecond() << std::endl;

    return 0;
}
```

NB: Tra <> andremo a indicare i tipi che dovranno essere sostituiti al interno della classe

10 Alberi Binari

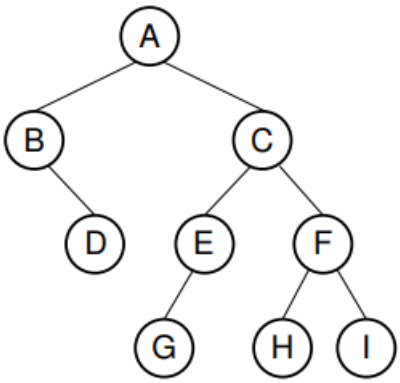
- **Gli alberi** sono un insiemi di **nodi** finti.
- **il nodo** da dove parte tutto è chiamata **root(radice)**
- Ogni nodo puoi avere dei figli(Max 2) questi figli sono dei sotto alberi del padre.
- Ricordiamo che un albero T_1 è un albero se ha un nodo n_1 e se **esistono** due **sotto alberi** T_2 e T_3 dove ognuno di essi ha un nodo che è diverso da n_1 e **l'intersezione** dei suoi sotto alberi T_2 e T_3 è **L'Insieme vuoto**
- Formalmente **Se** n_1 **è un nodo allora** T_1 **è un albero se** $\exists n_1 \in T_1$ **ed esistono** $\exists T_2, T_3 \subseteq T_1 \setminus \{n_1\}$ **tali che** $T_2 \cap T_3 = \emptyset$

10.1 Il percorso (Path)

Se abbiamo una **sequenza** di nodi $n_1, n_2, ..., n_k$ e n_i è il **padre** di n_{i+1} per $1 \leq i < k$. Questa sequenza è chiamata **percorso(Path). il Path è la sequenza di passi che impiego per arrivare da** n_1 **a** n_k , detto in altre parole: è **la sequenza di archi che attraverso** per arrivare a n_k. Per arrivare a n_k dobbiamo attraversare $k-1$ nodi poiché dobbiamo fermarci su n_k. quindi la lunghezza della Path sarà $k-1$

10.2 Discendente

Se esiste un percorso da un nodo n_k a un nodo n_{k+n} dove $n \in \mathbb{R}$ allora n_{k+n} è discendente di n_k.
Detto anche in altri modi se **esiste un percorso** da un nodo **R** a un nodo **M** allora **M** discende da **R**.



in questa immagine abbiamo la rappresentazione di un albero binario

10.3 Profondità(Depth) e Altezza(Height)

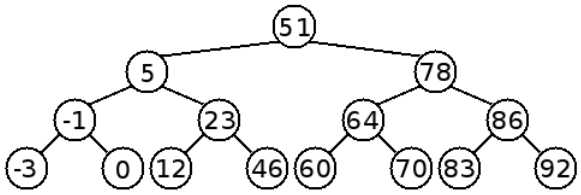
- La **profondità** è la lunghezza del **path** da un nodo **M** a un nodo **N**. Tutti i nodi a **profondità d** risiedono al **livello d**
- **L'altezza è**: Partendo da un nodo **N** l'altezza è il numero di **archi** che devo attraversare per arrivare a una foglia(nodo più in profondità) quindi possiamo dire che l'altezza è **Depth**. il percorso valido verso le foglie è quello più lungo.
- la differenza tra **Depth** e **l'altezza** che sono inverse, scendendo **Depth** aumenta invece **l'altezza** diminuisce
- il percorso valido verso le foglie è quello più lungo.
- Nel caso di un albero con un solo nodo allora si ha che la **Depth** e **l'altezza** sarà zero
- **P.S. :** La prof considera l'altezza come profondità + 1

10.4 Nodi interni e foglie

- **Un nodo interno** è un nodo che ha almeno un figlio
- **Una foglia (leaf)** è un nodo senza figli

10.5 Albero binario pieno

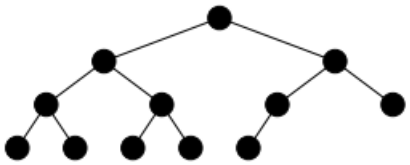
- Un albero è pieno quando per una certa altezza **h** ha il **numero massimo di nodi**, quindi ogni nodo interno ha sempre due figli.



10.6 Albero binario completo

Un albero è detto completo se rispetta queste tre proprietà:

1. è un **albero binario**
2. **le foglie** si trovano ad **altezza** h **o** $h-1$
3. **al più un nodo interno** può avere **meno di due figli**



NB: In questo corso per convenzione riempiamo un albero completo mettendo le foglie da **sinistra** verso **destra** in questo modo abbiamo la certezza che **tutti i livelli** siano pieni **eccetto** l'ultimo **livello** che può anche non essere pieno. L'albero di sopra rispetta questa proprietà.

10.7 Alberi binari Teoremi

un problema dei alberi binari e che non c'è nessun vincolo di come io **posso posizionare** i miei dati, quindi può accadere che si crei una **catena**. In una struttura in cui i **dati sono solo** conservati nelle **foglie** questo potrebbe essere un grande problema poiché consuma molto spazio.

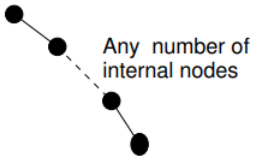


Figure 5.4 A tree containing many internal nodes and a single leaf.

Per ovviare a questo problema ci sono molte altre famiglie di alberi binari come i alberi binari pieni un teorema molto importante ci dice:

10.7.1 Teorema: numero massimo di foglie in un albero pieno

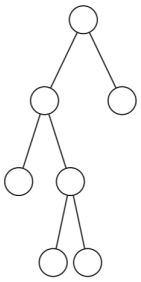
Enunciato:
Il numero di foglie in un albero pieno è uguale al numero di nodi interni più uno

Dimostrazione:

- Il teorema sarà dimostrato per induzione
- **Caso base:** Per un albero altezza zero ($h = 0$) abbiamo che c'è solo una sola foglia, quindi i nodi interni sono: zero più uno è uguale a uno ($0 + 1 = 1$). Invece per altezza uno ($h = 1$) avrò un nodo interno , quindi uno più uno è uguale a 2 ($1 + 1 = 2$), anche questa volta ci troviamo poiché essendo un albero pieno avremo due figli per l'unico nodo interno **(ovviamente un l'ipotesi è che stiamo lavorando su un albero binario pieno)**
- quindi osserviamo che sia per $h = 0$ e $h = 1$ le nostra formula per calcolare le foglie è corretta cioè numero di nodi interni $n + 1$
- **Ipotesi induttiva:** Ipotizziamo che ogni albero binario pieno T abbia $n - 1$ nodi interni e n foglie.
- **Passo induttivo:** Prendiamo un albero T con n nodi interni e prendiamo uno dei suoi nodi interni, lo chiamiamo I. Successivamente togliamo a questo nodo interno I i due figli rendendolo cosi foglia. cosi facendo avremo un **sotto albero** T' con $n-1$ nodi interni. Ora se prediamo T' per la nostra ipotesi induttiva sappiamo che T' ha n foglie poiché ha $n - 1$ nodi interni, se aggiungessimo a T' le foglie che abbiamo tolto a I cosa succederebbe? Avremo T' con $n+2$ foglie, dobbiamo però toglierne una poiché la foglia a cui furono sottratti i figli ora è padre(non togliamo la foglia fisicamente ma solo dal conteggio), cosi avremo n nodi interni e $n+1$ foglie. Abbiamo cosi dimostrato che anche aggiungendo due foglie(questo perché deve rimanere un albero binario pieno) avremo che il numero di foglie è il numero di nodi interni $+1$.

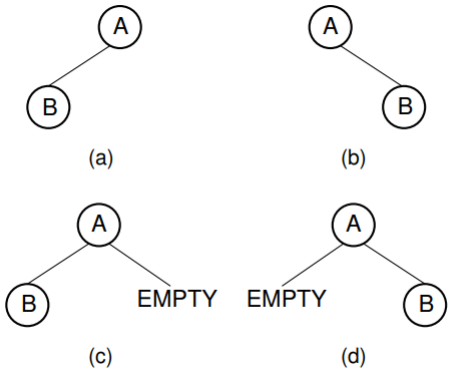
10.7.2 Nodi pieni

- **Un nodo è pieno** se ha due **figli** , se **non è pieno** è **foglia**(senza figli).
- quindi ogni nodo o è **interno** o altrimenti è **foglia** non può avere un solo figlio.



10.7.3 Definizione: alberi binari con figli non-vuoti e vuoti

In questa immagine possiamo vedere che la figura (a) è un albero binario con la root con un foglio **sinistro** non vuoto, la figura (b) è un albero binario con la root con un foglio **destro** non vuoto. Invece la (c) è un albero binario con la root con il figlio **sinistro** vuoto, la (d) è un albero binario con la root con il figlio **destro** vuoto



10.7.4 Teorema: quanti sotto alberi vuoti ha un albero binario

Teorema: *Il numero di sotto alberi vuoti in un albero binario è il numero di nodi + 1.*

Dimostrazione: Ogni albero binario T ogni nodo ha due figli quindi per n nodi un albero ha $2n$ figli dove n è il numero di nodi. L'unico nodo senza parente è la root quindi possiamo dire che $n-1$ nodi hanno dei parenti. sappiamo che $n-1$ nodi sono non vuoti. Togliendo al numero di figli totali il numero di nodi non vuoti $(n-1)$ quindi otteniamo cosi $n+1$ che è il numero di sotto alberi vuoti poiché so che n-1 non sono vuoti

10.8 Modi di visitare L'albero

- **Traversal**: ogni processo per attraversare tutti i nodi in un certo ordine e che eseguono una specifica azione(es. stampare, sommare ecc.)
- Ogni **Traversal** che elenca ogni nodo **una sola** volta viene chiamata **Enumerazione(visita)**
- Gli **attraversamenti** possono seguire diversi ordini, tra cui:
  - **Preorder**: visito prima me stesso poi a sinistra e poi a destra.
  - **Postorder** : visito prima sinistra poi a destra e poi me stesso.
  - **Inorder**: visito prima sinistra poi me stesso e poi destra.

- **Esempio per contare nodi in un albero:**

```
template <typename E>
int count(BinNode<E>* root) {
    if (root == NULL) return 0; // Nothing to
    count
    return 1 + count(root->left()) +
    count(root->right());
}
```

10.9 Spazio richiesto per albero binario

10.9.1 Struttura

- **Il nodo** deve contenere lo **spazio** per il dato e per i due **puntatori ai figli**
- **OverHead:** Il "overhead" per lo spazio di un albero si riferisce alla quantità di memoria aggiunta a quella utilizzata dai dati effettivi memorizzati nei nodi dell'albero.
- Quindi **l'overhead** è lo spazio richiesto per memorizzare tutto quello che non è il dato. quindi tutto quello che riguarda la struttura dati è **overhead**.
- infatti **l'overhead** dipende da:
  - in quale tipo di nodi memorizzo i dati (tutti, solo le foglie, . . .)
  - se le foglie contengono comunque puntatori nulli
  - se l'albero è a nodi pieni

10.9.2 Spazio Richiesto

- **Dato è piccolo**
  - **Costo Totale:** $n(2P + D)$
    n numero di nodi **P** sono i puntatori e **D** il dato
  - **OverHead:** $2nP$ senza il dato poiché è lo spazio richiesto in piu oltre al dato effettivo.
  - **Percentuale di overhead:** $2P/(2P + D)$ quindi se $P = D$ avrò il $2/3$ di spazio che è **overhead**
- **Se il dato è grande, nel nodo metto solo il puntatore**
- Ho quindi tre puntatori, tutti di **overhead**:
  - **Costo Totale:** $n(3P + D)$
    n numero di nodi **P** sono i puntatori e **D** il dato
  - **OverHead:** $3nP$ senza il dato poiché è lo spazio richiesto in piu oltre al dato effettivo.
  - **Percentuale di overhead:** $3P/(3P + D)$ quindi se $P = D$ avrò il $3/4$ di spazio che è **overhead**
- **Se invece l'albero è pieno e non ho puntatori sulle foglie l'overhead è:**
  - **Costo Totale:** Avremo $2P + D$ per circa $n/2$ **nodi interni**.
    Nelle foglie avremo D per circa $n/2$ **foglie**.

    Quindi in totale avremo $\frac{n}{2}(2P + 2D)$
  - **OverHead:** $\frac{n}{2}(2P)$
  - **Percentuale di overhead:**

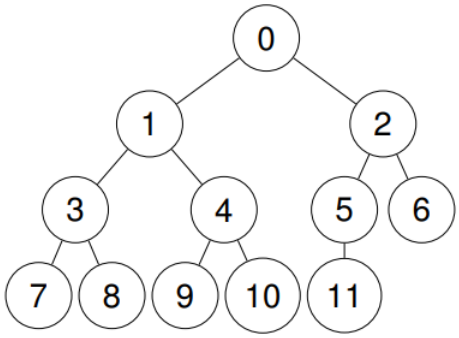
    $$\frac{\frac{n}{2}(2P)}{\frac{n}{2}(2P) + nD} = \frac{P}{P + D}$$
  - Se $D = P$ allora la percentuale di overhead è circa $1/2$

Puntatori ai dati solo nelle foglie:

- Nodi interni: solo puntatori ai figli
- Foglie: puntatore al dato
- Totale
$$\frac{n}{2}2P + \frac{n}{2}(P + D)$$
- Se $P = D$ ottengo $3P/(3P + D) = 3/4$
- Lo spazio total è diminuito, mentre è aumentato il rate di overhead perché adesso il dato sta solo nelle foglie

10.10 Implementazione di un albero binario (Vettore)

- Se conosciamo la struttura dell'albero, possiamo ridurre l'overhead legato ai puntatori.
- Se **l'albero binario è completo**, dato il numero di nodi **n** la sua struttura è unica.
- Questa **tecnica** funziona con i **alberi completi** poiché hanno un pattern ben preciso. Per funzionare questa tecnica ha bisogno che le foglie devono essere posizionate da **destra** verso **sinistra**.
- **Numero i nodi partendo dalla radice e scorrendo ciascun livello da sinistra verso destra in modo univoco**.
- **Questa numerazione mi dà la posizione all'interno dell'array:**
  - `Parent(r)` $= \lfloor \frac{r-1}{2} \rfloor$ se $r \neq 0$
  - `LeftChild(r)` $= 2r + 1$ se $2r + 1 < n$
  - `RightChild(r)` $= 2r + 2$ se $2r + 2 < n$
  - `LeftSibling(r)`[1] $= r - 1$ se r è pari
  - `RightSibling(r)` $= r + 1$ se r è dispari e $r + 1 < n$



Posizione	0	1	2	3	4	5	6	7	8	9	10	11
Genitore	-	0	0	1	1	2	2	3	3	4	4	5
Figlio sin.	1	3	5	7	9	11	-	-	-	-	-	-
Figlio dx.	2	4	6	8	10	-	-	-	-	-	-	-
Fratello sin.	-	-	1	-	3	-	5	-	7	-	9	-
Fratello dx.	-	2	-	4	-	6	-	8	-	10	-	-

[1] Da il fratello Sinistro di r, se r è il figlio destro LeftSibling(r) non restituisce nulla

11 Iteratori

- é un elemento che è in grado di indicare L'Elemento successivo in una struttura dati
- I Funzioni:
  - **`Lettura del dato nell'elemento;`**
  - **`Controllo di terminazione;`**
  - **`Successore (++);`**
  - **`Ripristinare la radice`** (ma solo nei `ResettableIterator`) o comunque il punto di inizio

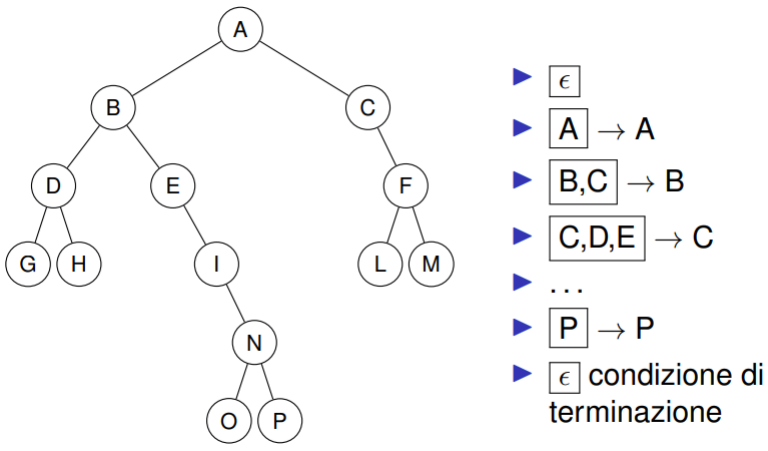
11.1 Iteratori per i alberi

Abbiamo diversi modi per visitare un albero:

- **Visita in ampiezza**
- **Visita in profondità**

11.1.1 Visita in ampiezza

- Abbiamo bisogno di una `coda` di supporto per:
  - **`Lettura del dato`**: metodo della classe;
  - **`Controllo di terminazione`**: controlla se la coda è vuota;
  - **`Sucessore`**: Dequeue del nodo visitato e Enqueue dei due figli del nodo corrente.



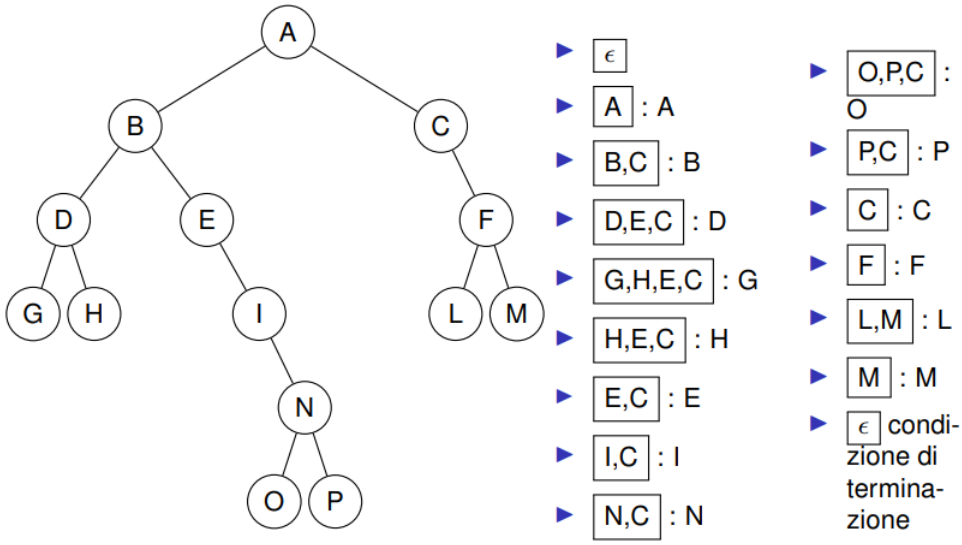
11.1.2 Visita in profondità

Abbiamo tre modi per visitare in profondità:

- **`Preorder`**
- **`Inorder`**
- **`Postorder`**

Ci serve uno `Stack` di supporto per la visita **`PreOrder`**:

- **`lettura del dato:`** metodo della classe;
- **`controllo di terminazione:`** `stack` vuoto;
- **`successore:`** `Pop` del nodo visitato e `push` dei due figli del nodo corrente.

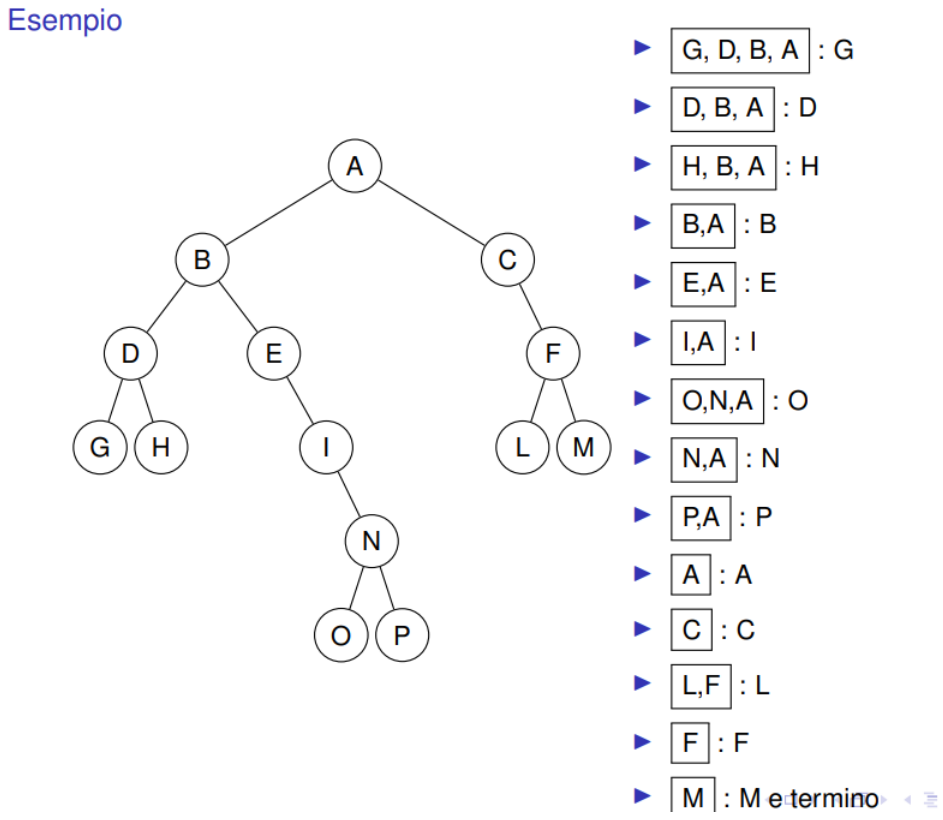


- **Codice Visita PreOrder:**

```
template <typename Data>
BTPreOrderIterator<Data> & BTPreOrderIterator<Data>::operator++(){
    if(Terminated()){
        throw std::out_of_range("Sei andato troppo oltre!");
    }

    if(corrente->HasRightChild()){
        Stk.Push(corrente->RightChild());
    }

    if(corrente->HasLeftChild()){
        Stk.Push(corrente->LeftChild());
    }

    if(Stk.Empty()){
        corrente = nullptr;
    }else{
        corrente = &(Stk.TopNPop());
    }

    return *this;

}
```

Visita `InOrder`:

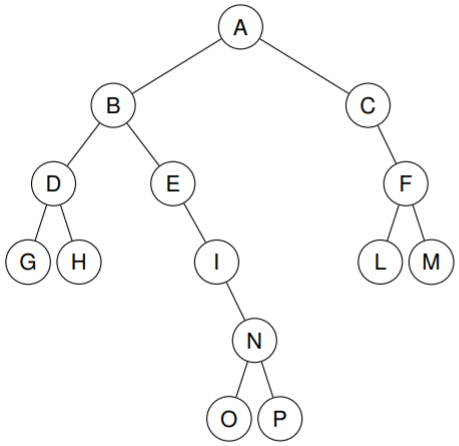
- In questo caso il **nodo corrente** viene scoperto, ma non **ancora visitato**: prima bisogna visitare il **sottoalbero sinistro**.
- Solito `stack` di supporto.
- `searchLeftMostNode` Ci serve anche una funzione che continua a scendere a sinistra fino a quando possibile inserendo man mano i nodi che incontra nello stack: si ferma al primo nodo il cui figlio sinistro è vuoto.
- scopri Quando scopriamo un nodo, ne facciamo il `push` nello `stack` e poi scendiamo a sinistra con `searchLeftMostNode`
  - **`lettura del dato:`** metodo della classe;
  - **`controllo di terminazione:`** stack vuoto;
  - **`successore:`** Pop e restituisco il nodo; visita il suo nodo destro, se c'è.
- Naturalmente parto dalla radice dell'albero.



Visita `PostOrder`:

- È un poi' più complicato, perché la strategia cambia a seconda che stia risalendo l'albero da destra o da sinistra: per capirlo devo tenere l'ultimo nodo restituito, che chiameremo `current`.
- La visita di un nodo implica fare il `Pop` del nodo e poi scendere a sinistra con `searchLeftMostList`: quando trova un nodo col figlio sinistro vuoto, salta al destro e rincomincia a sinistra, fino a quando non trova una foglia.
  - **`lettura del dato:`** metodo della classe;
  - **`controllo di terminazione:`** `stack` vuoto;
  - **`successore:`** ripeti fino a quando non arrivi ad un `Pop` e restituisci:
    - se `current == figlio sinistro` del `Top` dello `stack`, allora scopri il **figlio destro**;
    - se `current == figlio destro` del `Top` dello `stack`, `Pop` e **restituisci**
    - altrimenti (foglia), `Pop` e **restituisci**
- Naturalmente parto dalla radice dell'albero.

Esempio



- curr: ϵ; G, D, B, A : G
- curr: G; H, D, B, A : H
- curr: H; D, B, A : D
- curr: D; O, N, I, E, B,A : O
- curr: O P,N,I,E,B,A : P
- curr: P N,I,E,B,A : N
- curr: N I,E,B,A : I
- curr: I E,B,A : E
- curr: E B,A : B
- curr:B L,F,C,A : L
- curr:L M,F,C,A : M
- curr:M F,C,A : F
- curr:F C,A : C
- curr:C A : A e termino

12 HashTable

Le **hashtable** è un metodo per accedere alla array, il funzionamento è tramite delle chiavi(**keys**). Questo garantisce al momento della ricerca tempo constante poiché al record(l'informazione) è associata una chiave. Questa ricerca tramite la chiave per trovare il **record** viene chiamata anche **Hashing**. I **record** vengono ordinati nel array in modo che **soddisfino il calcolo dei indirizzi**(cosi la ricerca nel array è più semplice). non sono ordinati per frequenza o valore.

Terminologie:

- La funzione che mappa le chiavi e l'associa a una posizione nel array si chiami **`Hash function`** denotato con **h**.
- L'array che immagazzina i **recrod** viene chiamato **`Hash Table`** e viene denotata con la parola **HT**
- Una posizione nella **hash tabele** viene chiamato **`slot`**
- slot disponibili nella Hash Table viene denotato con **M**, i slot sono numerati da **0 a M-1**
- La **`Home`** è l'indice restituito dalla **Hash function**.

l'obbiettivo per le **hash** è quello di organizzare le cose in modo che: Per ogni **K** chiave e una qualsiasi funzione **h** abbiamo che $i = \mathbf{h}(K)$ dove i è l'indice del array dove è immagazzinata l'informazione(**lo slot dove risiede il record**) della chiave **K**. la funzione **h** restituisce un valore tra $0 <= h(K) < M$. L'Obbiettivo è avere che **HT[i]** contiene il record della chiave **K**

12.1 Debolezze generali

- Le **hashTable** non sono molto utili quando ci sono più **record** con la stessa chiave (quando permesso);
- **hashTable** non è molto efficiente a cercare in un determinato range
- non è efficiente a trovare il massimo o il minimo di un valore di una chiave oppure visitare i record in ordine di chiave

12.2 Utilità

- Quando è utile ? quando vogliamo cercare ogni **record** con un certa chiave **K**. le hash vengono utilizzate per i database infatti vengono immagazzinate non solo sulla memoria centrale ma anche sulle memorie come hardisk o ssd.

12.3 Osservazioni

- Nel caso ideale avremo un **hash table** con **M solt** e avremo delle chiavi **k** che vanno da **0 a M-1**. Cosi facendo avremo che con la chiave **k** nella **Hashtable** avrò **HT[k]** che contiene il **record** a cui è associato la chiave **k**. nel caso ideale usiamo tutti le chiavi. quindi a ogni chiave è associato un solo elemento.
- ovviamente questo se L'Intervallo delle possibili chiavi combacia con il numero di **slot**
- nel caso reale questo non avviene, se abbiamo che il range di chiavi è tra 0 e 65.535 dovremmo avere come minimo 65.535 slots però se abbiamo un aspettativa di utilizzare solo 1000 chiavi allora avremo altri 64.535 slots che non saranno mai utilizzatati.

12.4 Risoluzione

- Quindi dobbiamo creare una funzione hash **h** che può mappare un grande intervallo di chiavi nel più piccolo numero di **slot** quindi in un **hash table** più piccola possibile.
- Questo perché di solito ci sono **sempre più possibili chiavi** che **slot**

12.4.1 Problema(collisioni)

- Questa soluzione può portare a **collisioni** perché per due chiavi K_1 e K_2 po accadere che $\mathrm{h}(K_1) = \beta = \mathrm{h}(K_2)$ dove β è uno **slot** della **tabla hash**
- La ricerca di un record con valore chiave **K** in un database organizzato mediante **hashing** segue una procedura in due passaggi:
  - 1. Calcolare la posizione della tabella **h(K)**.
  - 2. A partire dallo slot **h(K)**, individuare il **record** che contiene la **chiave K** utilizzando (se necessario) un criterio di **risoluzione delle collisioni**.

12.5 Funzione di Hash

- Lo scopo della **funzione Hash** è di utilizzare meno **slot** possibili per un certo **intervallo di chiavi**, ovviamente è impossibile evitare le **collisioni** in questo caso.
- in modo probabilistico meno **slot** abbiamo è più aumenterà la mobilita di **collisioni** poiché meno spazio disponibile. inoltre al aumentare delle chiavi aumenta di conseguenza la probabilità di **collisioni**.
- quindi **diminuire** i slot e **aumentare** le chiavi aumenta le **collisioni**

- consideriamo, ad esempio, un'aula piena di studenti. Qual è la probabilità che una coppia di studenti condivida lo stesso compleanno (cioè lo stesso giorno dell'anno, non necessariamente lo stesso anno)? Se ci sono 23 studenti, allora le probabilità sono pari che due condivideranno lo stesso compleanno. Questo nonostante il fatto che ci siano 365 giorni in cui gli studenti possono avere compleanni (ignorando gli anni bisestili), nella maggior parte dei quali nessuno studente della classe compie gli anni. Con più studenti, aumenta la probabilità di un compleanno condiviso.

Una funzione Hash si può considerare:

- **Indifferente:** va bene qualsiasi funzione che prende una chiave e restituisce uno slot.
- **Pessimo:** per qualsiasi chiave restituisce lo stesso slot: la tabella di hash non ci aiuta nella ricerca.
- **Ottimo:** ogni slot ha la stessa probabilità di venir colpito.

12.6 Distribuzione delle chiavi

- Per poter controllare la **distribuzione degli slot** (quante volte viene colpito uno slot o possibilità che un slot possa essere colpito), **dovremmo conoscere la distribuzione delle chiavi**(la possibilità che una chiave esca).
- Di solito conosciamo il range di valori che le chiavi possono assumere.
- A volte la distribuzione delle chiavi all'interno del range ci è favorevole:
  - **Esempio** distribuzione uniforme delle chiavi(il che significa che ogni possibile chiave avrebbe la stessa probabilità di uscire): basta dividere il range in parti uguali. Questa frase si riferisce a un esempio di distribuzione uniforme delle chiavi all'interno di un certo intervallo di valori.

    Immagina di avere un intervallo di valori interi da **0 a 1000** e di voler distribuire le chiavi all'interno di questo intervallo in modo uniforme. **Dividere il range** in parti uguali significa che puoi suddividere l'intervallo totale in **segmenti** di uguale dimensione.

    Ad esempio, se vuoi distribuire le chiavi uniformemente su **10 segmenti o slot (o "parti uguali")**, puoi fare quanto segue:

    * **Segmento 1:** da 0 a 100
    * **Segmento 2:** da 101 a 200
    * **Segmento 3:** da 201 a 300
    * **Segmento 10:** 901 a 1000

Ogni segmento ha la stessa dimensione (in questo caso, 100) e contiene un decimo dell'intervallo totale di chiavi. Distribuire le chiavi in questo modo assicura che ogni segmento abbia la stessa probabilità di essere selezionato, garantendo così una distribuzione uniforme delle chiavi all'interno dell'intervallo.

- Purtroppo la fortuna è rara, e in genere le chiavi risultano ammucchiate in alcune parti e rare altrove.
- In questi casi la scelta della funzione di hash è critica,
- ma dovrebbe partire dalla conoscenza della distribuzione delle chiavi.

12.6.1 Perché distribuzioni non uniforme

avvolte le cattive distribuzioni non sono solo sfortuna:

- 1. Distribuzione di Zipf (Es: città grosse e piccole).
- 2. La raccolta dei dati introduce bias (Es: arrotondamenti dei numeri).
- 3. La prima lettera delle parole inglesi (e italiane e . . .) non è uniforme.

12.6.2 Esempi di funzioni di Hash

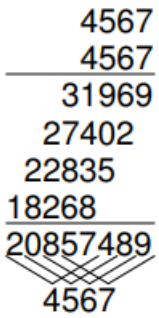
16 slots:

```
int h(int x) {
    return x % 16;
}
```

- Lo slot dipende solo dai quattro bit meno significativi, che possono avere una cattiva distribuzione.
- Esempio di una scelta tipica: %M: la scelta della dimensione della tabella di hash ha spesso un effetto importante sulle prestazioni.

Un altro esempio:

- Metodo mid-square:
- Fai il quadrato della chiave.
- Prendi gli r bit centrali.
- Ottieni uno slot in $[0, 2^r - 1]$.
- Esempio:
  - Chiavi: numeri decimali di 4 cifre.
  - $M = 100$ (2 cifre decimali $\Rightarrow r = 2$)
  - **Chiave**: 4567, al quadrato: 208**57**489
  - Tutte le cifre contribuiscono a **57** quindi ci sarà più varianza e meno probabilità che esca lo stesso numero



12.7 Gestione delle collisioni

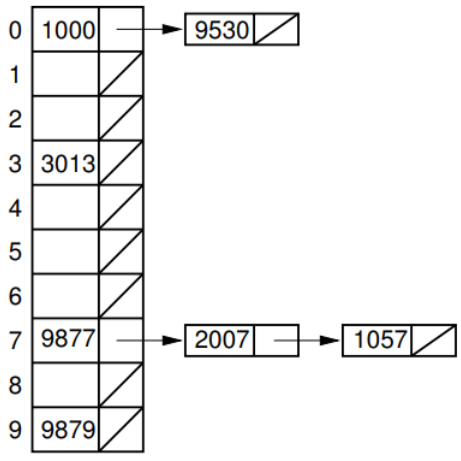
- In pratica, le **collisioni** non possono venir escluse a priori, quindi vanno gestite.
- Due classi principali, con:
  - **Hashing Aperto**(o con catene separate): le collisioni vengono immagazzinate fuori della tabella di **hash**.
  - **Hashing Chiuso** : dentro la tabella di hash, in un altro slot

12.8 Open Hashing (indirizzamento chiuso)

Nel **`Open Hashing`** per ogni **slot** avremo un **record** , nel caso in cui ci fosse una **collisione** si creerebbe una lista **linkata**. in uno **slot** potremmo avere più **record linkati** fra di loro.

Benefici:

- Questo metodo è molto semplice da gestire, di solito si può ordinare gli elementi della linked list tramite dei criteri, del tipo:
  - Per numero di accessi
  - Per ordine dei chiave
  - a caso
- ovviamente dare un ordinamento può ritornare utile poiché il **tempo medio potrebbe beneficiarne**. se esempio accedo molto spesso alla chiave **"cane"** mediamente il mio θ medio sarà migliore.



Se abbiamo **M** slot e **N** record da immagazzinare avremo che in ogni slot ci sarà $N \div M$. se invece stiamo nel caso in cui abbiamo **più slot** che **record** si spera che avremo **meno slot** possibili **con più di un record.**
se ogni slot ha solo un record il tempo medio di accesso è $\theta(1)$.

Svantaggi:

- La **linked list** o comunque delle strutture che hanno dei salti da un blocco al altro della memoria sono efficienti solo nella memoria primaria
- Nella **memoria secondaria** sarebbe poco efficiente poiché dobbiamo **accedere** più volte alla **memoria secondaria**(Sappiamo che la memoria secondaria è molto più lenta).

12.9 Closed Hashing (indirizzamento aperto) Con Buckets

L'Hashing chiuso utilizza i **Buckets** quindi per **M** slot e **B** buckets avrò che ogni **Bucket** avrà $M \div B$ slots.
Ogni volta che la **Hash function** darà un indice questo sarà il numero del **bucket**. nel momento del inserimento del record dovrò partire dal inizio del **bucket** e scorrere finché non trovo uno slot libero. se il **bucket** è pieno il record sarà inserito in un **overflow bucket** che ha spazio "infinito"

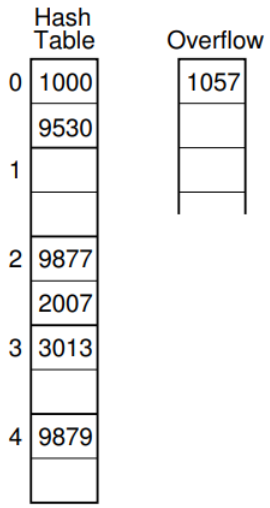


Figure 9.4 An illustration of bucket hashing for seven numbers stored in a five-bucket hash table using the hash function $\mathbf{h}(K) = K \bmod 5$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to bucket 0, three values hash to bucket 2, one value hashes to bucket 3, and one value hashes to bucket 4. Because bucket 2 cannot hold three values, the third one ends up in the overflow bucket.

Una **buona** implementazione della **Hash function** e riempire sempre tutti i **bucket** e cercare di andare meno possibile nel **overflow bucket**

12.9.1 Ricerca dei record

Quando dobbiamo ricercare i record nel **HashTable** dobbiamo utilizzare la La **Hash Function** con la chiave **K** per determinare il **bucket**. Possiamo ricadere in vari casi:

- Ricercare il **Bucket** dove risiede il **record** associato alla **chiave K**(la Chieve **K** ci serve a trovare il **bucket** tramite la **funzione di hash**, ma **K** ci servirà a trovare il corretto **record** nel **bucket**). Successivamente aver trovato il **bucket** si ricercherà il **record** con la chiave **K** nel **bucket**. Se **ricercando** troviamo una posizione vuota e non abbiamo ancora trovato il **record** allora quest'ultimo non è presente.
- Se ricercando finisce il **bucket** dovremo andare in quello di **overflow**. per determinare se il **record** c'è oppure no dobbiamo navigare tutto il **bucket di overflow**

12.9.2 Problemi e variante del bucket

Il problema principale è che iniziando sempre a inserire dal primo elemento del **bucket** c'è più probabilità di **collisione** , di conseguenza più accessi da fare. Quindi accedo al primo slot se c'è **collisione** scendo al secondo slot , controllo il secondo slot se c'è **collisione** vado avanti ... e cosi via.
La variante:

- Quando inserisco un **record** accederò a uno **slot casuale** del **bucket** cosi diminuendo la probabilità di **collisione**
- Poi scorro finché non trovo una posizione libera, se arrivo alla fine del **bucket** risalgo al inizio e continuo , questa ricerca di inserimento finirà quando torno al punto di partenza.
- il **bucket** quindi è "circolare"

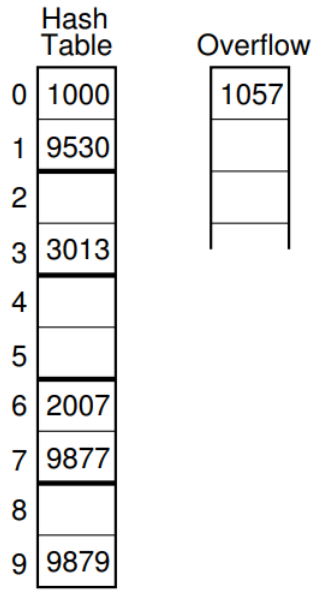


Figure 9.5 An variant of bucket hashing for seven numbers stored in a 10-slot hash table using the hash function $\mathbf{h}(K) = K \bmod 10$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Value 9877 first hashes to slot 7, so when value 2007 attempts to do likewise, it is placed in the other slot associated with that bucket which is slot 6. When value 1057 is inserted, there is no longer room in the bucket and it is placed into overflow. The other collision occurs after value 1000 is inserted to slot 0, causing 9530 to be moved to slot 1.

12.10 Closed Hashing (indirizzamento aperto) Con Linear Probing

Questo metodo non ha i **bucket**.

12.10.1 Inserimento:

Quando andiamo alla **Home** e accade una collisione durante l'inserimento si andrà a cercare nello slot successivo della **home**, se c'è uno **slot** libero si inserisce altrimenti si va avanti. Quindi **l'inserimento** cercherà dalla **home** in poi uno slot libero dove inserire l'elemento(La HashTable è circolare).

Passaggi:

- Primo passo ottenere la **Home** con la funzione di **Hash**.
- Partendo dalla **home** trovare uno slot libero (`EMPTYKEY`) per ottenere **l'offeset** da aggiungere alla **home** si richiama la **probe function**($\mathbf{P}(K, i) = i$ dove i è **L'offset**)
- l'inserimento non parte propino se non ho almeno due slot liberi , questo perché lo slot libero ci servirà nella ricerca.

Inserimento

```
// Insert e into hash table HT
template <typename Key, typename E>
void hashdict<Key, E>::hashInsert(const Key& k, const E& e) {

    int home; // Home position for e
    int pos = home = h(k); // Init probe sequence

    //ripeto finche non trovo uno slot dove inserire
    for (int i = 1; EMPTYKEY != (HT[pos]).key(); i++) {
        pos = (home + p(k, i)) % M; // probe

        //non inserisce se la chiave e' gia presente
        Assert(k != (HT[pos]).key(), "Duplicates not allowed");
    }

    //Crea un record dove E e' l'elemento e K e' la chiave
    KVpair<Key,E> temp(k, e);
    HT[pos] = temp;
}
```

La sequenza di slot dalla **Home** in poi viene chiamata **`probe sequence`** la funzione che restituisce **l'offset** da aggiungere alla **Home** è chiamata **`probe Function`**

12.10.2 Ricerca

Se si segue la **probe sequence** la fase di ricerca è possibile. questo perché se nel inserimento non si segue il **probe sequence** non abbiamo la sicurezza che se incontro uno spazio vuoto sono sicuro che la chiave **K** non c'è nella hash.

Passaggi:

- Trovo la **Home** con la funzione di **Hash**
- Se la **home** non contiene **K** vado al successivo, quindi utilizzando la **probe Function** calcolo offset è vado avanti.
- mi fermerò quando lo trovo oppure quando trovo uno slot vuoto, questo indicherà che non c'è l'emento con chiave **K**

Ricerca

```
// Search for the record with Key K
template <typename Key, typename E>
E hashdict<Key, E>::hashSearch(const Key& k) const {
    int home; // Home position for k
    int pos = home = h(k); // Initial position is home slot

    //quando trovo la chiave o quando trovo un slot vuoto esco dalla ricerca
    for (int i = 1; (k != (HT[pos]).key()) &&
        (EMPTYKEY != (HT[pos]).key()); i++){
            pos = (home + p(k, i)) % M;
    }

    // controllo se lo slot e' quello che cerco
    if (k == (HT[pos]).key()) return (HT[pos]).value();// Found it

    else return NULL; // k not in hash table
}
```

12.10.3 Primary Clustering

Il **Proping primario** è il peggior metodo per risolvere le **collisioni**, perché soffre del **Primary Clustering**. Questo problema tende a far raggruppare due insieme di record.

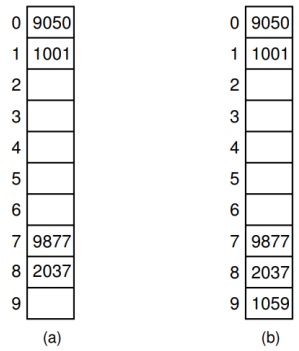


Figure 9.8 Example of problems with linear probing. (a) Four values are inserted in the order 1001, 9050, 9877, and 2037 using hash function $\mathbf{h}(K) = K \bmod 10$. (b) The value 1059 is added to the hash table.

Come notiamo nella figura sopra i due blocchi si tendono ad unirsi poiché le probabilità che il prossimo record sia inserito nella posizione **2** è del **0.6** quindi 6/10 invece per tutti i **slot** da **3 a 6** è una possibilità su **10 quindi 0.1** questo perché se colpisco **7,8,9,0 o 1** andranno tutti a inserire nello **slot 2** poiché L'Inserimento andrà avanti finché non trova uno slot vuoto.

1. **Aumento del Tempo di Ricerca:**

   Quando si verifica clustering primario, le operazioni di ricerca richiedono più tempo perché più elementi devono essere esaminati. Se molte chiavi si trovano nello stesso cluster, bisogna scorrere una lunga sequenza di celle per trovare l'elemento desiderato.

2. **Riduzione delle Prestazioni di Inserimento e Cancellazione:**

   Le operazioni di inserimento e cancellazione diventano meno efficienti perché richiedono il trattamento di collisioni. In presenza di un grande cluster, un nuovo elemento che collide con questo cluster deve essere inserito alla fine del cluster, aumentando così la lunghezza della sequenza.

3. **Degradazione dell'Equilibrio della Tabella Hash:**

   L'accumulo di elementi in poche aree della tabella riduce l'efficacia della funzione di hash nel distribuire uniformemente le chiavi. Questo porta a un uso inefficiente dello spazio della tabella hash e può richiedere ridimensionamenti più frequenti.

4. **Effetto Cascata:**

   Una volta che si forma un cluster, è probabile che questo cluster cresca più velocemente di altre aree della tabella, poiché le collisioni successive tenderanno ad aggregarsi ulteriormente al cluster esistente.

12.11 Miglioramento gestione collisioni (Hashing chiuso)

Ci sono vari modi per risolvere il **primary clustering**, il metodo più veloce è quello di aggiungere una costante moltiplicativa al **offset**.

- Quindi la **funzione di Proping** diventa nel seguente modo $\mathbf{P}(K, i) = ci$ dove **c** è la costante moltiplicativa.
- La **probe sequence** sarà del tipo $(\mathbf{h}(K)+ic)$ **mod** M. il modulo di M per far in modo che il numero sia sempre al interno del numero di slot.

Un buona **probe sequence** dovrebbe fare in modo che tutti i slot vengano visionati prima di ritornare alla **home**. Sfortunatamente in questo caso non è cosi.

Esempio

- Se ho $c = 2$, la home position è pari e Il numero di slot M è pari allora avrò che capiterò sempre e solo sui slot pari, al contrario se la home position dispari andrò solo i quelli dispari.
- quindi non andrò mai a visionare tutti i slot ma solo una parte.

Essendo **M** variabile potrei avere dei problemi. Osserviamo che abbiamo **due blocchi**: il blocco dei **numeri pari** e quello dei **numeri dispari**. Se uno dei due si riempisse troppo e l'altro rimarrebbe mezzo vuoto avremo che da una parte le **prestazioni sono ottime** dal altro **pessime**. Questo non ci piace perché va a impattare in modo **negativo** mediamente su tutta la struttura. Detto in poche parole il gioco non vale la candela!

12.11.1 Risoluzione:

Per risolvere questo problema basterà prendere una **c** che non abbia divisori in comune con la **M**(oppure prendere M primo) cosi facendo si prenderanno tutti i **solt**. Quindi se ho $M = 10$, **c** dovrà essere **1,3,7** oppure **9**. Se invece $M = 11$ basterà un numero **tra 0 e 10**. Osserviamo che questo però non ci salverà dal **clustering primario** poiché con $\mathbf{c = 2}$ le chiavi k_1 e k_2 potrebbero avere una **`probe sequence`** che converge nella stessa. Se $\mathbf{h}(k_1) = \mathbf{3}$ la sequenza è **3,5,7,9** e cosi via. Se $\mathbf{h}(k_2) = \mathbf{5}$ la sequenza è **5,7,9** e cosi via. come possiamo vedere dopo un certo punto è la stessa **`probe sequence`**. in sintesi questo metodo non risolve il **clustering primario**

12.11.2 Risoluzione Definitiva Clustering Primario

Esistono due modi per risolvere **Clustering Primario** il primo è avere un **offset** casuale , cosi facendo è statisticamente improbabile che due chiavi hanno la stessa **`probe sequence`**. Questo è impossibile poiché se avessimo un offset casuale non potremmo più ricercare un record poiché per una stessa chiave abbiamo un offset sempre diverso. Quindi posiamo usare dei numeri **pseudo-casuali**, questa tecnica viene chiamata **`pseudo-random probing`**

Funzionamento:

- Avremo quindi che $\mathbf{(h}(K) + r_i\mathbf{)\ mod}\ M$ sarà lo slot del inderimento/ricerca
- r_i è il valore nella **i-esima** cella del vettore che conterrà valori casuali ma fissi, cioè dopo aver deciso i numeri casuali quest'ultimi non cambieranno. non sono valori casuali ma permutazioni dei valori da 1 a M-1.
- **`probe Function`** è $\mathbf{p}(K, i) =$ **`Perm[`**$i - 1$**`]`** dove **`Perm`** è un array di lunghezza **M-1** contando la **permutazione casuale** dei valori fa **1 a M-1**. **Quindi per chiavi diverse non avrò mai la stessa sequenza di probing.**

Example 9.9 Consider a table of size $M = 101$, with `Perm`$[1] = 5$, `Perm`$[2] = 2$, and `Perm`$[3] = 32$. Assume that we have two keys k_1 and k_2 where $\mathbf{h}(k_1) = 30$ and $\mathbf{h}(k_2)$ = 35. The probe sequence for k_1 is 30, then 35, then 32, then 62. The probe sequence for k_2 is 35, then 40, then 37, then 67. Thus, while k_2 will probe to k_1's home position as its second choice, the two keys' probe sequences diverge immediately thereafter.

Il secondo metodo per risolvere il **clustering primario** è **quadratic probing** dove la **funzione di proping** è $\mathbf{p}(K, i) = c_1 i^2 + c_2 i + c_3$.
il caso più facile di è $\mathbf{p}(K, i) = i^2$ quindi $(\mathbf{h}(K) + i^2)$ **mod** M

Example 9.10 Given a hash table of size $M = 101$, assume for keys k_1 and k_2 that $\mathbf{h}(k_1) = 30$ and $\mathbf{h}(k_2)$ = 29. The probe sequence for k_1 is 30, then 31, then 34, then 39. The probe sequence for k_2 is 29, then 30, then 33, then 38. Thus, while k_2 will probe to k_1's home position as its second choice, the two keys' probe sequences diverge immediately thereafter.

il problema del **clustring quadratico** è che alcuni slot non verranno mai visionati, se per esempio avessimo una **hashtable** di **size 3** quindi abbiamo i slot **0,1 e 2** possiamo dire con certezza che **2** non sarà mai **visionato**. un modo semplice di risolvere è di avere **M** come numero primo cosi sicuramente meta tabella potrà essere riempita. oppure usare una **funzione di proping** del tipo $\mathbf{p}(K, i) = (i^2 + i)/2$ e M $= 2^S$(M come potenza di 2) cosi facendo ogni **slot** sarà riempito.

12.11.3 Clustering Secondario

Il **`secondary clustering`** si genera con le due nuove **funzione di proping**.
Il **`secondary clustering`** non genera dei cluster contigui ma delle **chiavi diverse K** possono avere la stessa **`probe sequence`** e cosi facendo avendo dei cluster, però non contigui.

Clustering Secondario

Il **clustering secondario** si verifica nelle tabelle hash quando si utilizza una strategia di probing come il probing quadratico o il doppio hashing per risolvere le collisioni. A differenza del clustering primario, il clustering secondario non crea blocchi continui di celle occupate, ma piuttosto crea gruppi di celle occupate che seguono un modello di collocazione ripetitivo.

Esempio di Clustering Secondario con Probing Quadratico

Immaginiamo di avere una hash map con 10 slot (indice da 0 a 9) e utilizziamo il probing quadratico per risolvere le collisioni. La funzione di probing quadratico può essere definita come:

$$\text{indice} = (h + i^2) \% \text{dimensione\_tabella}$$

Dove h è il valore hash iniziale e i è il numero di tentativi (0, 1, 2, ...).

1. Inseriamo una chiave `A` che ha un valore hash iniziale `3`. La chiave `A` va nella cella `3`.
2. Inseriamo una chiave `B` che ha un valore hash `3`. La cella `3` è occupata, quindi usiamo il probing quadratico:
   - $i = 1$: (3 + 1^2) % 10 = 4. `B` va nella cella `4`.
3. Inseriamo una chiave `C` che ha un valore hash `3`. La cella `3` è occupata, quindi:
   - $i = 1$: (3 + 1^2) % 10 = 4. La cella `4` è occupata.
   - $i = 2$: (3 + 2^2) % 10 = 7. `C` va nella cella `7`.

Ora, se un'altra chiave `D` ha un valore hash `4`, e finisce seguendo una sequenza simile:

1. Inseriamo una chiave `D` che ha un valore hash `4`. La cella `4` è occupata, quindi:
   - $i = 1$: (4 + 1^2) % 10 = 5. `D` va nella cella `5`.

Notiamo che chiavi con hash iniziali diversi (3 e 4) possono finire in celle che formano un pattern di clustering secondario basato sulla funzione di probing.

12.12 Risoluzione Clustering Secondario

Per risolvere il **clustering secondario** si utilizza una **seconda funzione hashing** , il risultato di questa seconda funzione viene usata come **costante moltiplicativa**. quindi avremo $\mathbf{p}(K, i) = (i * h_2(K))$. questa h_2 restituisce un valore tra $1 <= h_2 <= M - 1$

Example 9.11 Assume a hash table has size $M = 101$, and that there are three keys k_1, k_2, and k_3 with $\mathbf{h}(k_1) = 30$, $\mathbf{h}(k_2) = 28$, $\mathbf{h}(k_3) = 30$, $\mathbf{h}_2(k_1) = 2$, $\mathbf{h}_2(k_2) = 5$, and $\mathbf{h}_2(k_3) = 5$. Then, the probe sequence for k_1 will be 30, 32, 34, 36, and so on. The probe sequence for k_2 will be 28, 33, 38, 43, and so on. The probe sequence for k_3 will be 30, 35, 40, 45, and so on. Thus, none of the keys share substantial portions of the same probe sequence. Of course, if a fourth key k_4 has $\mathbf{h}(k_4) = 28$ and $\mathbf{h}_2(k_4) = 2$, then it will follow the same probe sequence as k_1. Pseudo-random or quadratic probing can be combined with double hashing to solve this problem.

---

questo ci garantisce di non avere `probe sequence` uguali per chiavi diverse

12.13 Risoluzione problema cancellazione

La cancellazione deve affrontare due problemi:

- **Cancellare un record** potrebbe creare dei problemi nella ricerca. Quando noi inseriamo un dato seguiamo una sequenza di **probing**, nel momento in cui **cancelliamo** un elemento potremmo rompere questa sequenza. Questo significa che nel inserimento siamo passati per un elemento che ora nella ricerca non c'è più(poiché cancellato) quindi troveremo uno **slot vuoto** prima del dovuto, cosi facendo non visiteremo tutta la sequenza di **probing**.
- Il secondo problema che dobbiamo far in modo che lo slot che vogliamo cancellare sia riutilizzabile anche dopo la cancellazione

La risoluzione è molto semplice: utilizzeremo delle **tombstone**(pietre tomabali). metteremo una **TombStone** quando cancelliamo, che sarà valutato come uno **slot pieno**nella ricerca, come **slot vuoto** nel inserimento. Quindi nel inserimento se **troviamo** una **tombstone** inseriamo, invece nella ricerca continuiamo.

12.14 Considerazioni

Quindi quando si fa un **HashMap** si deve guardare:

- Un ottima distribuzione dei slot quindi avere meno collisioni possibili
- una buona cosa di solito quando si vuole una ottima distribuzione è usare i **numeri primi**, poiché i **numeri primi** non hanno divisori in comune con nessuno. quindi i numeri che hanno gli stessi fattori produrranno valori **hash** con **pattern ripetitivi**. Usando un **numero primo**, si **riduce** questa ripetizione. questo perché si farà h_1(k) % **M**
- diciamo che per una distribuzione non uniforme delle chiavi dovrebbe ridurre il numero di collisioni
- ricordiamo che le chiavi nel caso reale non hanno una distribuzione uniforme.

13 Grafi

Il grafo è un insieme di coppie di vertici dove:

- $< V, E >$ Dove $E = \{A \subseteq V || A| = 2\}$ questo significa che **E** è un insieme formato da coppie di **vertici V**
- Possiamo indicare un grafo come **G** quindi possiamo scrivere $G =< V, E >$
- Quindi **E** è la **relazione binaria** su **V** ossia $E \subseteq V \text{x} V$ quindi **E** è L'insieme degli archi che sono presenti nel grafo **G**
- In un grafo **orientato** le coppie in **E** sono ordinate , quindi $(X, Y) \neq (Y, X)$ in un grafo **non orientato** questa differenza non c'è.
- Diciamo che $0 \leq |E| \leq |V|^2 - |V|$, facciamo **meno |V|** poiché non considerammo i **self loop**
- Due vertici che hanno una **relazione** si chiamano **adiacenti**, quindi se abbiamo la coppia (X, Y) allora **X e Y** sono adiacenti.
- Un **Grafo è etichettato** se ogni vertice ha una **"Etichetta"** che lo distingue dai altri.
- un **ciclo è semplice** se e solo se il primo è l'ultimo vertici sono uguali
- un grafo senza cicli viene chiamato aciclico
- Il **grado massimo di un grafo** è il **numero massimo di archi uscenti da un nodo**

13.1 Cammini

- Un percorso chiamato anche **Path** è una sequenza di vertici $v_0, v_1, ..., v_n$ purché esistono dei archi tra questi vertici , quindi $(v_i, v_{i+1}) \in E$ dove $1 \leq i \leq n - 1$
- Un **Path** è **Semplice** se tutti i vertici del percorso sono distinti
- La lunghezza del percorso è data dal numero di archi attraversati
- Un **Path** è **ciclico** se il **primo e l'ultimo vertice si ripetono**, un percorso per essere ciclico deve avere almeno una lunghezza ≥ 3

13.2 Sotto Grafi

- Presi Due grafi **G** e **G'** dove $G =< V, E >$ e $G' =< V', E' >$
- Diciamo che **G'** è un **sotto grafo** se $V' \subseteq V$ e $E' \subseteq E$
- Dciamo che **G'** è un **sotto grafo indotto** se $V' \subseteq V$ e $E' = E \cap V' \text{x} V'$

13.3 Componenti concesse e fortemente connesse

- Un **grafo** o un **sotto grafo non orientato** si dice **connesso** se ogni vertice del grafo o sotto grafo può raggiungere tutti i altri
- formalmente Se **G non orientato** si dice **connesso** se:

$$\forall v, w \in V | (V, W) \in Reach(G)$$

  dove **Reach** è L'insieme di **coppie di vertici che sono raggiungibili** Quindi se abbiamo (V,W) significa che V raggiunge W e che W raggiunge V(dipende anche dal grafo)

- Un **grafo** o un **sotto grafo orientato** si dice **fortemente connesso** se ogni vertice del grafo o sotto grafo può raggiungere tutti i altri
- formalmente Se **G orientato** si dice **fortemente connesso** se:

$$\forall v, w \in V | (V, W) \in Reach(G)$$

  dove **Reach** è L'insieme di **coppie di vertici che sono raggiungibili** Quindi se abbiamo (V,W) significa che V raggiunge W e che W raggiunge V(dipende anche dal grafo)

- Invece quando parliamo di **componenti** stiamo parlando di un **sottoinsieme massimale**, quindi **Componenti concesse e fortemente connesse** in più hanno solo che **sono massimali**

13.4 Metodi di Rappresentazione

Abbiamo due metodi di rappresentazione di un grafo:

- **Tramite matrice di adiacenza**
- **Tramite liste di adiacenza**

13.4.1 Matrice di Adiacenza

- Matrice $|\mathbf{V}| \times |\mathbf{V}|$ di bit: 1 se esiste l'arco, 0 altrimenti.
- Per rappresentare matrici pesate, ogni elemento contiene un numero.
- In ogni caso, richiede spazio $\theta(|V|^2)$
- Tempo richiesto per aggiungere o rimuovere un altro vertice è di $\theta(|V|^2)$ poiché devo **creare** una nuova matrice e poi copiare gli elementi della **vecchia alla nuova**
- La ricerca di adiacenti di **un vertice** è **lineare su $|\mathbf{V}|$** quindi $O(|V|)$, invece per una **visita completa** è $\theta(|V|^2)$
- cancellazione di un arco è constante.

13.4.2 Liste di Adiacenza

- Vettore di **|V|** liste linkate (o altri contenitori più adeguati): la lista nella posizione **i-esima** contiene i vertici adiacenti a V_i (e in questo modo rappresenta gli archi).
- il vettore piu li liste linkate richiedono uno spazio di $\theta(|V|+E)$, ovviamente perchè l'array di **|V|** più le listelinkate che tutte insieme possono essere massimo **|E|**.
- Per aggiungere o rimuovere un vertice il costo sarà di $\theta(|V|)$ poiché si dovrà allocare un vettore e non una matrice
- Per una ricerca dei **ADJ** di un vertice è $\theta(|E|)$
- Per una ricerca di tutto il grafo allora $\theta(|V|+|E|)$ quindi io navigo tutto l'array **|V|** , dopo aver visionato tutto il grafo avrò visitato **|E|** archi. Diciamo che la somma di tutte liste di adiacenza di ogni vertice è **|E|**
- La cancellazione di un arco non è più constante può essere nel caso peggiore $O(|E|)$ nel caso migliore $\Omega(1)$

13.4.3 Quando usere Una matrice o una lista di adiacenza

Un grafo si dice:

- Il grafo è **Denso:** se il numero di archi si avvicina a $\theta(|V|^2)$
- Il grafo è **Sparso:** se il numero di archi è più o meno $\theta(|V|)$

Ci conviene usare una **matrice** se il grafo è **denso** poiché occuperemo meno spazio , questo perché la lista di adiacenza per colpa dei puntatori per una grande quantità di archi avrà tanti puntatori e cosi pesando di più.
Al contrario se un **grafo è sparso** allora ci conviene usare le **lista di adiacenza**, poiché sprecheremo meno spazio.

13.5 Metodi Accessori

- In genere gli algoritmi prevedono di visitare i nodi adiacenti al nodo dato.
- Daremo supporto a questa necessità con due funzioni:
  - **first(v)** restituisce il primo vertice adiacente al nodo **v**, quindi il primo della lista
  - **next(v,n)** restituisce il vertice adiacente a **v** immediatamente dopo n nella lista dei nodi adiacenti; **n** = **|V|** a fine lista.

```
1 for(w=G->first(v); w < G->n(); w=G->next(v,w))
```

13.6 Visite

- Come per gli alberi, anche per i grafi esistono delle visite **(traversals)** standard.
- Ogni vertice viene visitato una sola volta.

13.6.1 Problemi

- **Due problemi principali:**
  1. grafo **non connesso**: non è possibile raggiungere tutti i vertici da quello scelto per partire;
  2. presenza di **cicli**, che possono portare, se non controllati, a loop infiniti.
- Entrambi i problemi possono venir **risolti** con dei **marcatori** sui vertici (un bit per vertice):
  1. **cicli**: evito di visitare vertici già visitati;
  2. grafo **non connesso**: alla fine controllo per vedere se ci sono vertici ancora non visitati (se ci sono, riparto con la visita da uno di questi)

13.7 Implementazione Visita

Quindi per vistare tutti i vertici dobbiamo **scorrere tutto il vettore** è se il **vertice non è stato vistato** allora vistare tutti i suoi **ADJ**

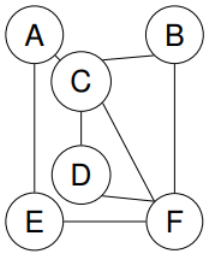
```
1  void graphTraverse(Graph* G) {
2      int v;
3      for (v=0; v<G->n(); v++)
4          // Initialize mark bits
5          G->setMark(v, UNVISITED);
6
7      for (v=0; v<G->n(); v++)
8          if (G->getMark(v) == UNVISITED)
9              doTraverse(G, v);
10 }
```

- Dove **doTraverse()** può implementare una visita in **ampiezza o in profondità.**

13.8 Visita in ampiezza(BFS)

- **Breath-first search (BFS).**
- Prima di procedere visiti tutti i vertici collegati al vertice corrente.
- Struttura di supporto: **coda**.

```
1  void BFS(Graph* G, int start, Queue<int>* Q) {
2      int v, w;
3      Q->enqueue(start);
4      // Initialize Q
5      G->setMark(start, VISITED);
6      while (Q->length() != 0) { // Process all vertices on Q
7          v = Q->dequeue();
8          PreVisit (G, v); // Take appropriate action
9          for (w=G->first(v); w<G->n(); w = G->next(v,w))
10             if (G->getMark(w) == UNVISITED) {
11                 G->setMark(w, VISITED);
12                 Q->enqueue(w);
13             }
14         }
15     }
16 }
```



Coda	Azioni
A	enq(A)
C E	deq(A), enq(C), enq(E)
E B D F	deq(C), enq(B), enq(D), enq(F)
B D F	deq(E)
D F	deq(B)
F	deq(D)
	deq(F)
	end

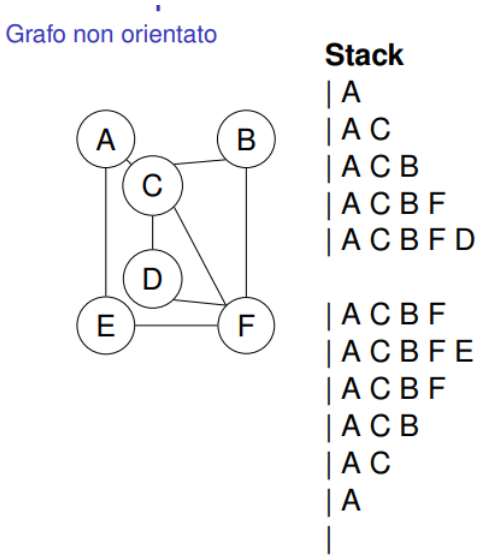
13.9 Visita in profondità(DFS)

Lo scopo della **DFS** è seguire il primo percorso del grafo finché non termina, nel momento in cui è terminato comincerà a risalire lo stack di esecuzione o di supporto finché non troverà una biforcazione. dopo aver trovato la biforcazione farà la stessa cosa che ha fatto al inizio , cioè percorrere tutto il percorso finché non termina. farà questo finché non visiterà tutto l'albero.

- **Ricorsiva:** ogni volta che si visita un vertice **v** si visitano anche i suoi vicini non ancora visitati. però appena si trova un vicino non visitato entreremo in quel vicino e visteremo i suoi vicini, ripetiamo questo finché non troveremo nessun altro da esplorare
- **Iteratori**:
  - si inseriscono in uno **stack** tutti gli archi che escono da **v**;
  - per **trovare il prossimo** vertice da visitare, si estrae e segue un arco dallo **stack**.
- L'effetto è di seguire un ramo nel grafo fino alla sua conclusione prima di risalire le biforcazioni.
- Si costruisce così un **albero di ricerca in profondità.**
- **Vale sia per i grafi orientati che per quelli non orientati**

13.10 DFS Ricorsiva

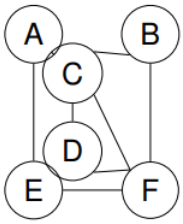
```
void DFS(Graph* G, int v) { // Depth first search
    PreVisit(G, v); // Take appropriate action
    G->setMark(v, VISITED);
    for (int w=G->first(v); w<G->n(); w = G->next(v,w)){
        if (G->getMark(w) == UNVISITED){
            DFS(G, w);
        }
    }
    PostVisit(G, v); // Take appropriate action
}
```



13.11 DFS Iterativa

```
void DFS() {
    init(stack,visitedVertices,adjIterators);
    for(int v=vertices.first(); vertices.hasMore();
    v=vertices.Next())
    if(!v.isVisited()){
        PreVisit(v);
        stack.push(v);
        while(!stack.isEmpty()){
            v = stack.top();
            v.markVisited();
            while(adjIterators[v].hasNext() and
                (w = adjIterators[v].Next()).isVisited())
                    // do nothing;

            if(!w.isVisited()){
                PreVisit(w);
                stack.push(w);
            }else{
                stack.pop();
                PostVisit(v);
            }
        }
    }
}
```

Post-order

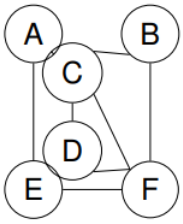


Stack

| A
| A C
| A C B
| A C B F
| A C B F D

| A C B F
| A C B F E
| A C B F
| A C B
| A C
| A
|

Pre-order



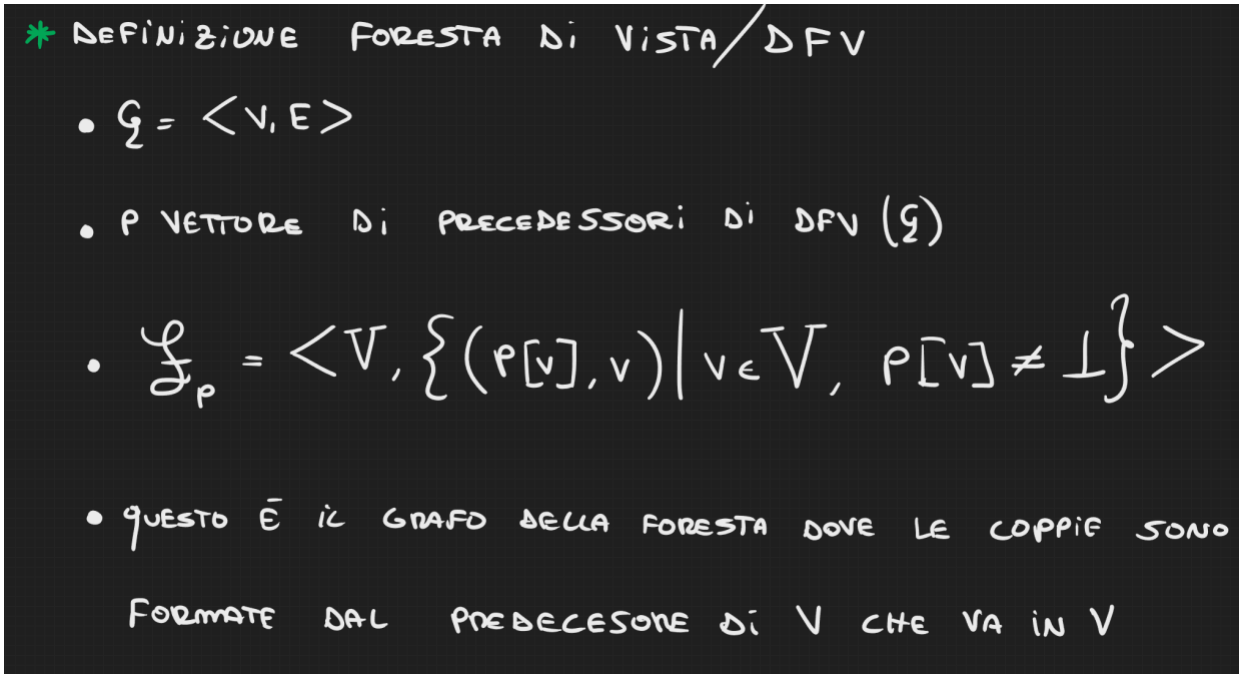
Stack

| A
| A C
| A C B
| A C B F
| A C B F D

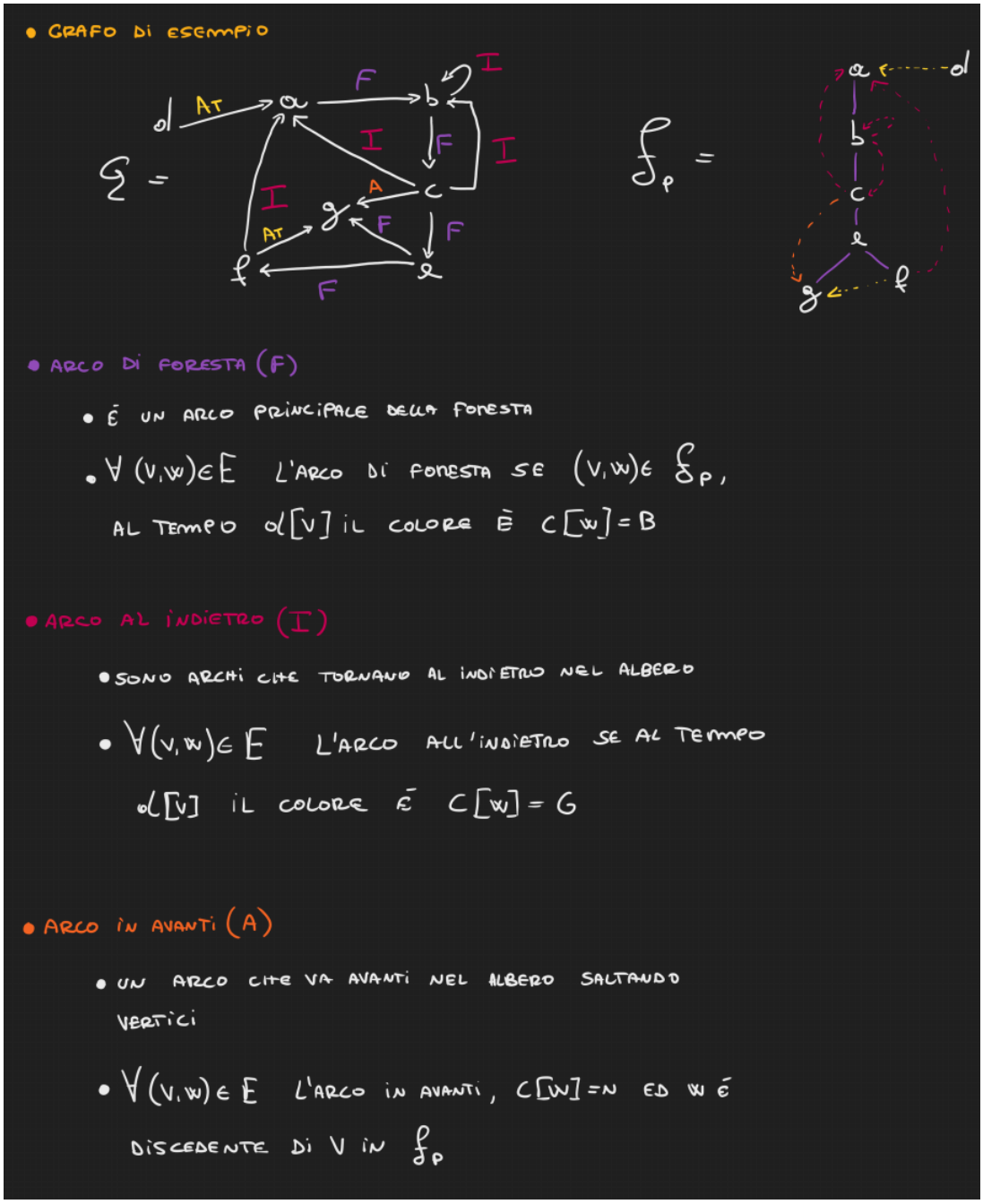
| A C B F
| A C B F E
| A C B F
| A C B
| A C
| A
|

Lo **stack non cambia** ma cambia solo l'ordine in cui viene effettuato il lavoro sul vertice

13.12 Foresta di visita



13.12.1 Archi di Foresta



14 Intercorso

In questo paragrafo raggrupperò tutto quello che c'è da sapere sulle tre prove intercorso

14.1 Using

`Using` è una **parola chiave** simile al `typedef` ma più versatile , si puo dire che è L'Evoluzione. **`Using`** può essere usato per alias a:

1. **Alias di tipo:** dare un nome a un tipo

```
using NewTypeName = ExistingTypeName; //caso generale

#include <iostream>
#include <vector>

// Alias di tipo per un vettore di interi
using IntVector = std::vector<int>;

int main() {
    IntVector vec = {1, 2, 3, 4, 5};
    for (int n : vec) {
        std::cout << n << ' ';
    }
    return 0;
}
```

2. **Alias per le funzioni**: utilizzato per le funzioni anonime

```
#include <functional>

// Alias di tipo per una funzione che prende
//due parametri e restituisce un Accumulator
template <typename Accumulator>
using FoldFun =
    std::function<Accumulator(const Data &, const Accumulator &)>;
```

3. **Alias di Namespace:** Dare nomi ai Namespace

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

int main() {
    cout << "Hello,_world!" << endl;
    return 0;
}

#include <iostream>
using std::cout;
using std::endl;

int main() {
    cout << "Hello,_world!" << endl;
    return 0;
}
```

4. **Alias di tamplate:**

```cpp
template <typename T>
using Vec = std::vector<T>;

Vec<int> intVec = {1, 2, 3};
Vec<double> doubleVec = {1.1, 2.2, 3.3};
```

Quindi abbiamo 4 tipi di alias: **Tipo, funzioni, Namespace e tamplate**

14.2 lvalue e rvalue

14.2.1 lvalue

Un **lvalue** (left value) è un termine utilizzato in C++ per descrivere un'espressione che rappresenta un **oggetto** che ha **un'identità** e un **indirizzo di memoria accessibile**. In altre parole, un **lvalue** è qualsiasi cosa che possa apparire alla sinistra di un'assegnazione (anche se non è l'unica caratteristica distintiva).

Caratteristiche di un **lvalue**:

- **Ha un'identità**: Un lvalue rappresenta un oggetto che **risiede** in una **posizione** specifica nella **memoria.**
- **Può essere modificato**: Un lvalue **non const** può essere assegnato, quindi può essere modificato.
- **Può essere preso per riferimento**: Si può ottenere un riferimento a un lvalue.

```
int x = 10;
x = 20; // x e' un lvalue


int arr[3] = {1, 2, 3};
arr[0] = 10; // arr[0] e' un lvalue

int y = 30;
int* ptr = &y;
*ptr = 40; // *ptr e' un lvalue

int z = 50;
int& ref = z;
ref = 60; // ref e' un lvalue
```

14.2.2 rvalue

In C++, un **rvalue** (right value) è un termine che descrive un'espressione che rappresenta un valore temporaneo, che non ha un'identità persistente o un indirizzo di memoria specifico. Gli **rvalue** sono tipicamente valori temporanei che non possono essere assegnati ad altre variabili (non possono essere lvalue).

Caratteristiche di un **rvalue**:

- **Temporaneità:** Gli **rvalue** sono valori che esistono solo per la **durata di un'espressione**. **Non hanno un indirizzo di memoria stabile** a cui ci si può riferire dopo l'esecuzione dell'espressione.
- **Non modificabili:** Gli rvalue non possono essere assegnati; puoi usarli per ottenere un valore, ma non puoi **modificarli direttamente**.
- **Utilizzo in Espressioni:** Gli **rvalue** sono comunemente usati in espressioni matematiche, risultati di funzioni, e valori letterali.

indichiamo un **rvalue** con && (es. `f(Data && s)` s è un rvalue)

```cpp
int x = 10; // 10 e' un rvalue

int y = x + 5; // x + 5 e' un rvalue

 // "hello" + std::string("world") e' un rvalue
std::string s = "hello" + std::string("world");

int getValue() {
    return 42;
}
int z = getValue(); // getValue() e' un rvalue
```

14.3 std::move()

std::move è una funzione della libreria **standard di C++** che effettua il **cast** di un oggetto a un **rvalue reference**. Viene utilizzata principalmente per abilitare il trasferimento **(move)** di risorse da un **oggetto** a un **altro** senza **effettuare una copia**, ottimizzando così le prestazioni e l'efficienza della gestione delle risorse.

Cosa Fa **std::move**?

- **Cast a rvalue reference:** std::move effettua il **cast di un lvalue** (un oggetto con un nome e un indirizzo identificabile) a un **rvalue reference**. Un **rvalue reference** è un tipo di riferimento che può essere legato a oggetti temporanei, permettendo l'implementazione delle semantiche di movimento.
- **Non Sposta Effettivamente:** std::move **non sposta** effettivamente i dati o risorse; si limita a **indicare** che l'oggetto **può essere "spostato"**. L'effettivo spostamento delle risorse avviene nelle funzioni di movimento (**move constructor o move assignment operator**).

> **NB:** Dopo un'operazione di **movimento**, l'oggetto sorgente è lasciato in uno **stato valido** ma **indeterminato**. Deve essere pronto per la distruzione, ma **non deve** essere usato ulteriormente senza essere **riassegnato** o **ricostruito**. **No std::move su Oggetti Const** non ha senso.

In sintesi, **std::move** è uno strumento potente per migliorare l'efficienza del codice C++ consentendo il trasferimento delle risorse invece della loro copia.

```cpp
#include <iostream>
#include <string>
#include <utility> // Include std::move

class MyClass {
public:
    std::string data;

    // Costruttore
    MyClass(const std::string& str) : data(str) {
        std::cout << "Constructed with data: " << data << std::endl;
    }

    // Move constructor
    MyClass(MyClass&& other) noexcept : data(std::move(other.data)) {
        std::cout << "Move constructed with data: " << data << std::endl;
    }

    // Move assignment operator
    MyClass& operator=(MyClass&& other) noexcept {
        if (this != &other) {
            data = std::move(other.data);
            std::cout << "Move assigned with data: " << data << std::endl;
        }
        return *this;
    }

    // Prevent copy
    MyClass(const MyClass&) = delete;
    MyClass& operator=(const MyClass&) = delete;
};

int main() {
    MyClass obj1("Hello");
    MyClass obj2 = std::move(obj1); // Uses move constructor

    MyClass obj3("World");
    obj3 = std::move(obj2); // Uses move assignment operator

    return 0;
}
```

14.4 Lambada function

Una **funzione lambda**, o **funzione anonima**, in C++ è una funzione senza nome che può essere definita direttamente nel contesto in cui viene utilizzata. Le funzioni lambda sono state introdotte in C++11 e offrono una sintassi concisa per definire funzioni al volo, specialmente utili quando si lavora con funzioni di **ordine superiore**(es. fold, map ecc.).

La sintassi di base per una **lambda** in C++ è la seguente:

```
[ capture ] ( params ) -> return_type { body }
```

- **capture:** Specifica quali variabili del contesto circostante la lambda può catturare e utilizzare.
- **params:** Parametri della funzione lambda, simili a quelli di una funzione normale.
- **return_type:** (Opzionale) Il tipo di ritorno della lambda. Se omesso, il compilatore deduce il tipo di ritorno.
- **body:** Il corpo della funzione lambda.

```cpp
//ES 1
//mettiamo auto poiche' la funzione e' un tipo complesso
//quindi lasciamo il compito al compilatore
auto sum = [](int a, int b) {
    return a + b;
};

int result = sum(3, 4); // result e' 7


//ES 2
int x = 10;
int y = 20;

auto add = [x, y]() {
    return x + y;
};

int result = add(); // result e' 30


//ES 3
int x = 10;

auto increment = [&x]() {
    x += 1;
};

increment(); // x e' ora 11
```

```cpp
//ES 4
include <iostream>

int main() {
    // Definizione di una lambda che calcola la media di due interi
    auto average = [](int a, int b) -> double {
        return (a + b) / 2.0;
    };

    // Utilizzo della lambda per calcolare la media di 4 e 7
    double result = average(4, 7);

    std::cout << "La media di 4 e 7 e': " << result << std::endl; // Output
        : La media di 4 e 7 e': 5.5

    return 0;
}
```

14.5 Problema del Diamante

Il **Problema del diamante** è un problema comune in linguaggi di programmazione orientati agli oggetti che supportano l'ereditarietà multipla, come C++. Si verifica quando una classe eredita da due classi base che a loro volta ereditano da una stessa classe base comune. Questa situazione può creare ambiguità su quale versione dei membri della classe base comune deve essere utilizzata, poiché esistono più **"percorsi"** per accedere a quei membri.
Significa che quando in stanzio un oggetto e devo percorrere la strada verso il padre non saprò che strada percorrere.

Esempio:

```cpp
#include <iostream>

class Base {
public:
    void show() {
        std::cout << "Base_class" << std::endl;
    }
};

class Derived1 : public Base {
    // ...
};

class Derived2 : public Base {
    // ...
};

class DerivedFinal : public Derived1, public Derived2 {
    // ...
};

int main() {
    DerivedFinal obj;
    // obj.show(); // Errore: ambiguita'
    return 0;
}
```

- **In questo esempio**, `DerivedFinal` eredita da `Derived1` e `Derived2`, entrambe ereditano da `Base`. Se proviamo a chiamare `obj.show()`, il compilatore non sa quale versione di `show` chiamare, quella ereditata da `Derived1` o quella ereditata da `Derived2`, perché ci sono due **percorsi** per arrivare alla funzione `show`.

14.5.1 Risoluzione tramite virtual

In C++, possiamo risolvere il **Problema del diamante** usando **l'ereditarietà virtuale**. Con **l'ereditarietà virtuale**, diciamo al compilatore che ci sarà **una sola copia** della classe `base` condivisa tra tutte le classi derivate. **Questo elimina l'ambiguità**. elimina L'Ambiguità perché ci sarà solo una strada per la classe `base`.

Esempio:

```cpp
#include <iostream>

class Base {
public:
    void show() {
        std::cout << "Base_class" << std::endl;
    }
};

class Derived1 : virtual public Base {
    // ...
};

class Derived2 : virtual public Base {
    // ...
};

class DerivedFinal : public Derived1, public Derived2 {
    // ...
};

int main() {
    DerivedFinal obj;
    obj.show(); // Ora funziona correttamente, non c'e' ambiguita'
    return 0;
}
```

- **Ereditarietà Virtuale:** Le classi `Derived1` e `Derived2` ereditano da `Base` usando **`virtual public Base`**. Questo indica che `Base` deve essere condivisa tra tutte le **classi derivate** che utilizzano **l'ereditarietà virtuale**.
- **Ambiguità Risolta**: Ora, `DerivedFinal` ha una singola copia di `Base`, **eliminando l'ambiguità**. La chiamata a `obj.show()` funziona correttamente.

Quindi quando noi instanziamo una **classe** verrà instanziato anche la **classe padre**. Nel **`Problema del diamante`** avevamo che una classe ereditava da due classi che avevano due istanze diverse della stessa classe padre(cioè `Base`). mettendo **virtual** usiamo la stessa istanza per entrambi.

14.5.2 Quando usare l'Ereditarietà Virtual?

L'ereditarietà virtuale è utile quando:

1. **Si ha una gerarchia di classi con ereditarietà multipla**: Specialmente se si prevede che più percorsi di ereditarietà possano portare alla stessa classe base.
2. **Si desidera evitare copie multiple della stessa classe base**: Questo riduce il consumo di memoria e previene ambiguità.

14.5.3 Funzionamento Tecnico

Quando una classe **eredita un'altra classe in modo virtuale** oppure la classe che stiamo estendendo **contiene metodi virtuali** oppure la stessa classe contiene metodi virtuali allora abbiamo i seguenti tre componenti:

- **vptr (puntatore alla tabella dei metodi virtuali)**: Ogni oggetto che contiene almeno un **metodo virtuale** (o **eredita** da una **classe con metodi virtuali**) ha un **vptr**. Questo **vptr** punta alla **vtable della classe stessa**.
- **vtable (tabella dei metodi virtuali)**: La **vtable** è una struttura dati globale per ogni classe che ha almeno un metodo virtuale. **Contiene i puntatori** ai metodi virtuali definiti per quella classe e per le sue classi base(da cui ha ereditato). Nel caso di un **OverRiding** il puntatore non punterà più a un metodo ereditato ma al metodo che lui stesso a fatto **L'override**. **vtable** non è una struttura dati globale nel senso che è condivisa tra tutte le istanze di quella classe. Ogni istanza della classe ha il proprio **vptr (puntatore alla vtable)**, che punta alla **vtable** specifica della classe stessa. vtable risolve le chimate a funzione quando c'è di mezzo metodi **virtual**.
- **Offset**: Gli **offset** per l'accesso alle classi base(quelle ereditate) **ereditate virtualmente** non sono direttamente memorizzati nella **vtable**. Questi **offset** sono **calcolati** dal **compilatore** e gestiti internamente per determinare l'indirizzo corretto dell'oggetto della classe base all'interno di un oggetto della classe derivata. Quindi nello **spazio** di **indirizzi logici** di un **istanza di una classe** risiede un **riferimento** per ogni **istanza di classe padre(classi da cui eredita)**, cosi facendo se più classi ereditano dalla stessa **non si crea ambiguità e ridondanza**, Quindi ci sarà un unica istanza per tutte L'istanze delle classi figlio. Per **raggiungere** questo riferimento **si usa l'offset**, quindi sommando **l'offset al indirizzo di base del oggetto ottengo il riferimento all'istanza della classe padre.**

14.5.4 Esempi:

```cpp
#include <iostream>

class A {
public:
    virtual void foo() {
        std::cout << "A::foo" << std::endl;
    }
};

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

class C : public virtual A {
public:
    void foo() override {
        std::cout << "C::foo" << std::endl;
    }
};

class D : public B, public C {
public:
    void foo() override {
        std::cout << "D::foo" << std::endl;
    }
};

int main() {
    D d;
    d.foo(); // Output: D::foo

    A* a = &d;
    a->foo(); // Output: D::foo, risolto dinamicamente attraverso la vtable

    B* b = &d;
    b->foo(); // Output: D::foo

    C* c = &d;
    c->foo(); // Output: D::foo

    return 0;
}
```

```
vtable di A:
+---------+
| A::foo  |
+---------+

vtable di B:
+---------+
| B::foo  |
+---------+

vtable di C:
+---------+
| C::foo  |
+---------+

vtable di D:
+---------+
| D::foo  |  <-- Risolve la chiamata a ‘foo‘
+---------+
```

Se togliessimo **foo** da D avremmo:

```
vtable di A:
+---------+
| A::foo  |
+---------+

vtable di B:
+---------+
| B::foo  |
+---------+

vtable di C:
+---------+
| C::foo  |
+---------+

vtable di D:
+---------+
| B::foo  |  <-- Risolve la chiamata a ‘foo‘
+---------+
```

c'è **B** è non **C** poiché viene risolto prima **B** e poi **C** , questo perché **B** è stato scritto prima quindi in ordine di compilazione **B** viene risolto prima.

```cpp
#include <iostream>

// Classe base con un metodo virtuale
class A {
public:
    virtual void foo() {
        std::cout << "Chiamato foo() di A" << std::endl;
    }
    virtual void bar() {
        std::cout << "Chiamato bar() di A" << std::endl;
    }
};

// Classe B che eredita virtualmente da A
class B : public virtual A {
public:
    void foo() override {
        std::cout << "Chiamato foo() di B" << std::endl;
    }
};

// Classe C che eredita virtualmente da A
class C : public virtual A {
public:
    void bar() override {
        std::cout << "Chiamato bar() di C" << std::endl;
    }
};

// Classe D che eredita virtualmente da B e C
class D : public B, public C {
public:
    void foo() override {
        std::cout << "Chiamato foo() di D" << std::endl;
    }
    void bar() override {
        std::cout << "Chiamato bar() di D" << std::endl;
    }
};

int main() {
    D d;

    // Puntatore a D come classe base A
    A* ptrA = &d;

    // Chiamata ai metodi virtuali tramite il puntatore di tipo A
    ptrA->foo(); // Chiamato foo() di D
    ptrA->bar(); // Chiamato bar() di D

```

```
    return 0;
}
```

```
vtable per A:
|------------|
| A::foo     | ----> Puntatori ai metodi virtuali di A
|------------|
| A::bar     | ----> Puntatori ai metodi virtuali di A
|------------|

vtable per B:
|------------|
| B::foo     | ----> Puntatore al metodo foo() di B
|------------|
| A::bar     | ----> Puntatore al metodo bar() di A
|------------|

vtable per C:
|------------|
| A::foo     | ----> Puntatore al metodo foo() di A
|------------|
| C::bar     | ----> Puntatore al metodo bar() di C
|------------|

vtable per D:
|------------|
| D::foo     | ----> Puntatore al metodo foo() di D
|------------|
| D::bar     | ----> Puntatore al metodo bar() di D
|------------|
```