Differenza copy-constructor e copy-assignment:

* Constructor: bisogna stare attenti a creare copie di oggetti a cui punta l’oggetto principale, per non far puntare più oggetti ad uno stesso oggetto
* Assignment: ho già un oggetto e sto clonando un altro oggetto su di lui. Prima devo eliminare tutti gli oggetti collegati al primo

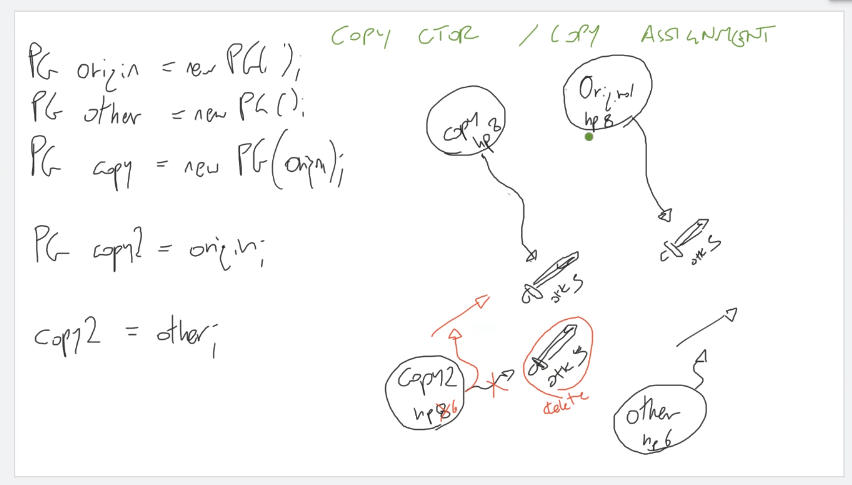
Pg origin = new pg();

Pg other = new pg();

Pg copy = new pg(other);

Pg copy2 = origin;

copy2 = other;



**Move constructor e move assignment**

Permettono ad esempio di SPOSTARE oggetti da un array ad un altro (senza copiarli)

In questo caso se ho un guerriero che punta ad una spada, spostandolo farò un nuovo guerriero, che però punta sempre alla stessa spada (il primo guerriero verrà delatato)

Move constructor:

Personaggio(Personaggio&& pg); <- gli sto passando un r value

Int a = 5;

* A detto l value
* 5 detto r value

Se faccio

Int a = b+c;

Ho “b+c” come r value, che è un valore temporaneo

stampaSomma(5, 3); ha 5 e 3 come r value

Gli r value si dichiarano come type&& nomevar

//inserire codice di Personaggio.cpp con il Personaggio::Personaggio(Personaggio&& pg)

Il move-assignment restituisce Personaggio& (inserire pure sto codice va)

Inoltre quando faccio move-assignment parto da un oggetto che già esiste, quindi quello devo liberarlo (e poi copiarci l’altro)

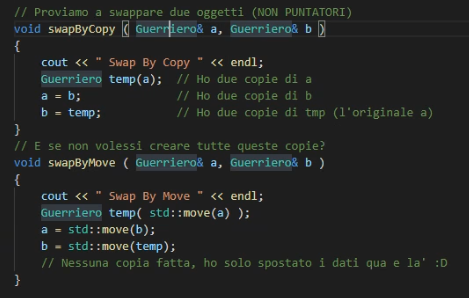
**NB:** il move constructor di una classe figlia, deve prima richiamare quello della madre

Ascoltare a 28 minuti che è importante

void stampaSomma(int m, int n);{roba}non ha r value, ma variabili locali che sono diventate l value qui dentro

std::move(g) trasforma g in un r value, serve per richiamare il move constructor (se passi un r value parte il move constructor, se passi un l value parte il copy constructor)-> riga 12 guerriero.cpp

//A 34 minuti continua ad essere importante



* Le reference non fanno copie, ma dentro sawpByCopy copio di tutto quindi lol
* Facendo a = std::move(b); richiamo il moveconstructor invece del copyassignment

**NB:** non serve fare il delete di oggetti locali di una funzione, tanto una volta eseguita vengono eliminati

**REGOLA DEL 5**

Se implemento anche uno solo tra distruttore move ecc devo implementarli tutti

**STRINGHE**

Non serve liberarle, tranne se si fa un puntatore ad una stringa

Stringa.substr(indice, per quanti caratteri);

Stringa.replace(indice, per quanti caratteri, con quale stringa);

Stringa.find(“O”); resituisce l’indice nel quale si trova la prima volta la stringa che gli ho passato

Stringa.at(); //rivedere, eccezioni?

Const char\* stringdichar = stringa.c\_str(); //Passa da string a char[]

Stringa s = R”(stringalol\ncaio\nboom)” // stringa literal, ignora gli \n (o meglio semplicemente non li tratta come spazi ma come caratteri)

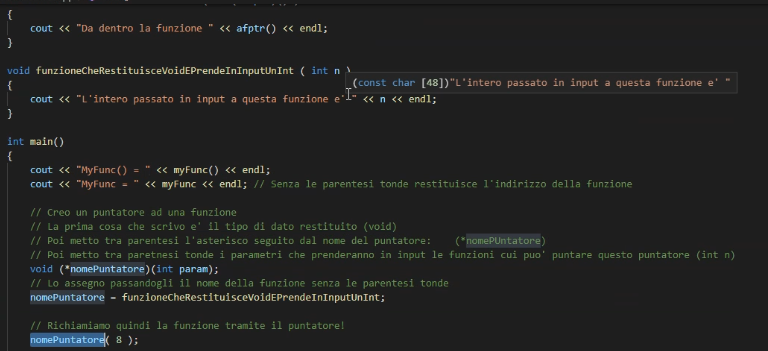
**PUNTATORI DI FUNZIONI**

MyFunc(){ return 2};

int a = MyFunc(); //Da il valore di return della funzione

int b = MyFunc; //Da l’indirizzo della funzione

//Minuto 57 importante

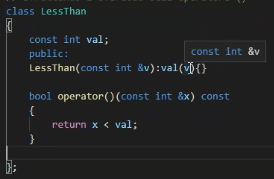


//ascoltare

**FUNCTORS**

Oggetti usati come funzioni, ossia faccio overload dell’operatore ()

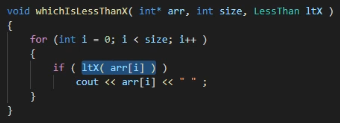
Es:



Controlla se il numero che gli si passa è minore del numero che gli ho dato quando l’ho istanziata

Utilizzo:

* Per passarlo ad un’altra funzione

Es: 

**SMARTPOINTERS**

* Unique pointer (è per forza unico, solo un puntatore punta a quell’oggetto)  
  devo passarlo via reference ad una funzione, perché sennò quella prova a farne una copia (e non si può fare)
* Shared pointer (sa anche quanti sono i puntatori shared che puntano all’oggetto che punta, se il contatore arriva a 0, distrugge anche l’oggetto **NON SI STA PIU’ ATTENTI AL DELETE DAJE**  
  **Contro:** nn o kpt)

//1 ora e 22 minuti boh vedere

Il move sullo shared sposta la ownership (quindi il contatore non si abbassa, rimane uguale)