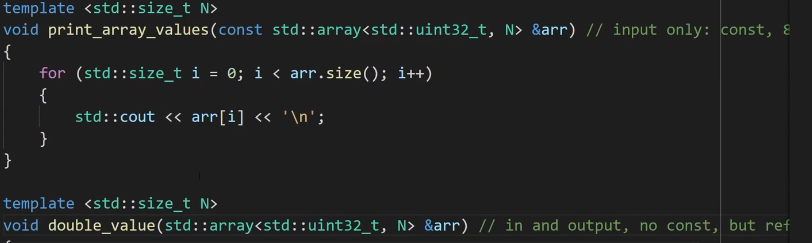
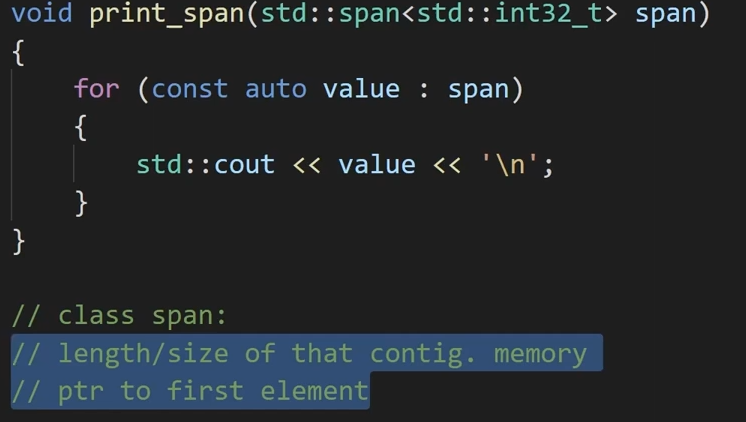
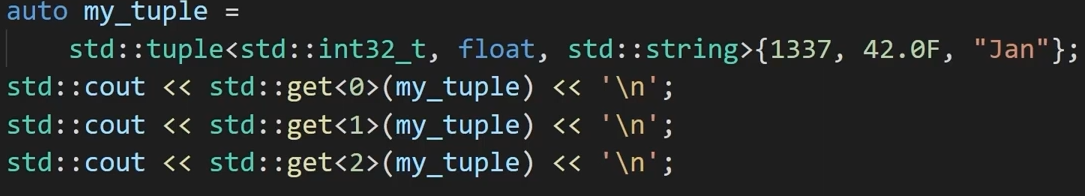
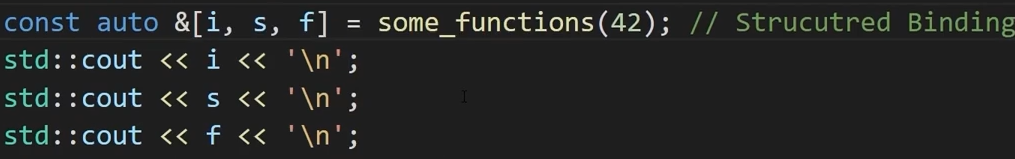
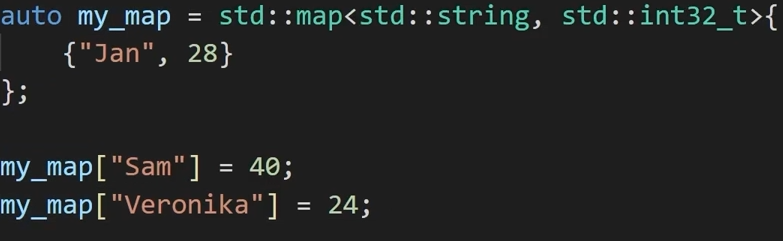
* **Structs:**
* Only use when I want to store data of different types, without the need of methods (if I need methods class is better)
* **Const:**
* Means the variable/object cannot be modified after initialization (Read only)
* The value can be set at runtime or compile time.
* **Constexpr**:
* Stronger guarantee: the value must be computable at compile time (when possible).
* Ensures the expression is a constant expression.
* Useful for array sizes, template parameters, switch cases, etc.
* **Anonymous namespace:**
* A modern C++ way to defining a function private to a single source file. (works like global variables) A screen shot of a computer code

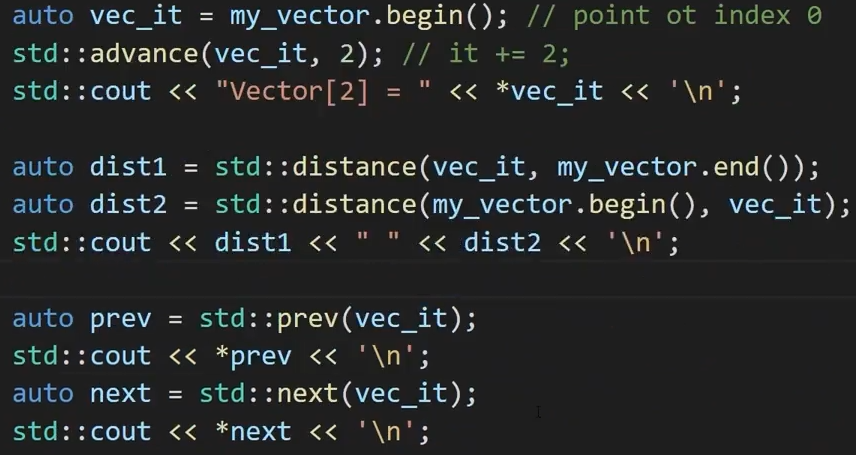
  AI-generated content may be incorrect.
* **Template:** is used to initialize arrays of size “N” (not a prefixed size)
* **Reference to array (pointer):**
* 
* Reference here is made in both cases but const for input only (read only) and in and output
* Use a pointer when you have to – otherwise use reference
* False of a pointer == **nullptr**, true of a pointer != nullptr (use in ifs)
* **lvalues**: An expression that refers to a specific object in memory (has an identifiable address).
* **rvalues**: A temporary value or a value that does not persist beyond the expression (no permanent address).
* **Templates:** Acts as Generics from Java:



**Span class:** does not own memory, it holds a reference to another container’s memory. So it can’t make a copy because it is a lightweight object which just holds a pointer to the first element of a (ANY – array, vector...) contiguous memory (an array/vector) 

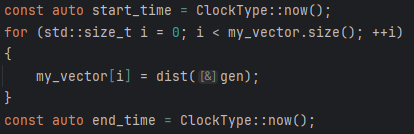
* **Pair type:** Acts similar to a struct – define a pair of variables with type of choice (don’t have to be of same type) mainly used for map (pairs of key value)
* **Tuple type:** Acts similar to struct aswell. Defined like this: 
* **Structured binding:** alternative to “get “ for each variable in a pair or a tuple or struct or .... with a single line command: 
* A computer screen with text

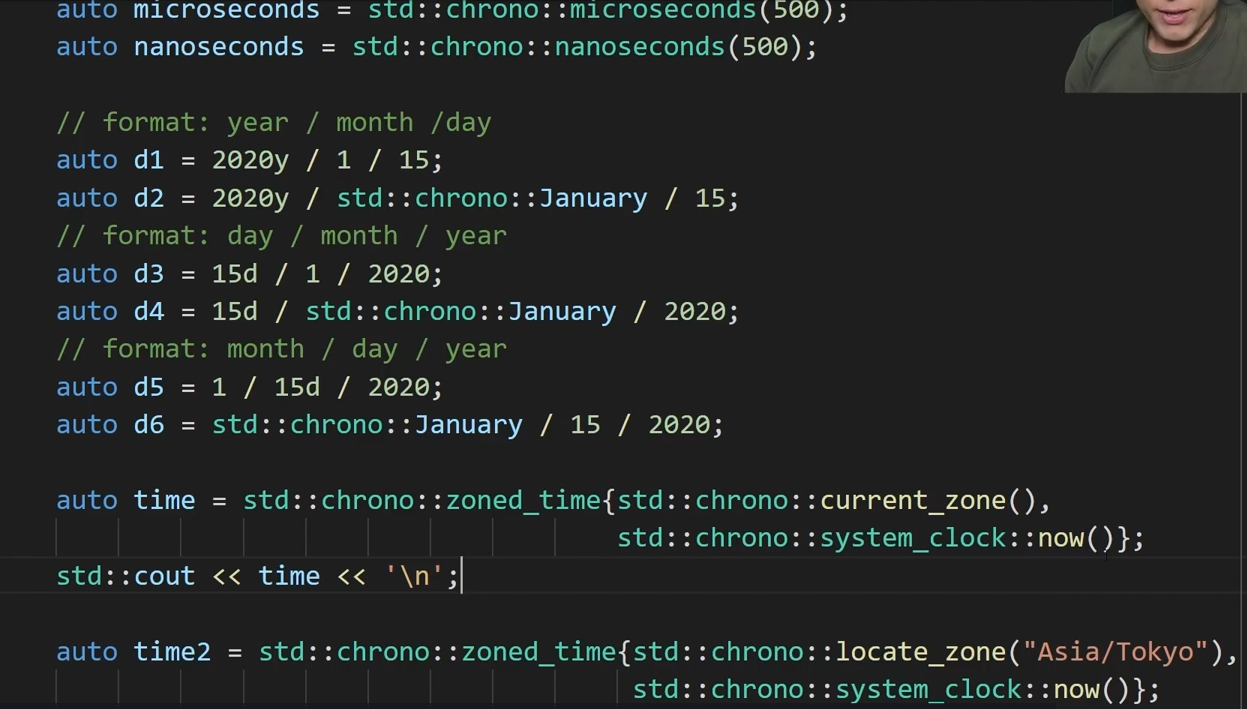
  AI-generated content may be incorrect.
* **Map**: 
* Iterator commands that works for \*all\* containers in CPP

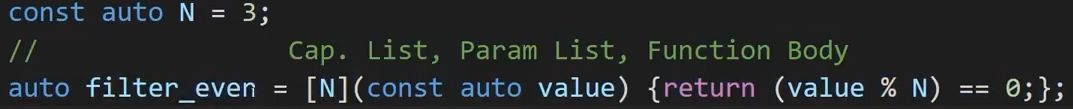


* **Strings:** Are mutable – can replace characters in strings.
* **std::string\_view** : when reading with “const std::string &s” as a function parameter – instead we should use std::string\_view – higher optimization.
* **Randomly generate numbers**:

std::int32\_t gen()  
{  
 static auto seed = std::random\_device{};  
 static auto generate = std::mt19937{seed()};  
 static auto distribution = std::uniform\_int\_distribution<std::int32\_t>{-10, 10};  
  
 return distribution(generate);  
}

* **Chrono library**: handles time management – use to measure runtime by sampling before and after execution of a code block -

Also provides date and more time options: 

* **Equality operator between floats**: When comparing between 2 floats we will use almost\_equal method instead of equal method since the floats might differ by a very small diff (epsilon)
* **Lambda functions**: Shorten ‘spontaneously’ defined functions in which we will use to perform actions that we will create separate functions to perform them. The syntax of lambda function: 

Looking at the structure – the () and {} acts as a standard function, the captions list is for input variables from the outside into the logic of the lambda function (wtf?)

(Note – in C++20 we don’t have to apply N in the Caption list)

* **std::generate**: calls a function on a series of elements in a defined range, can also be lambda function. Use case:
* auto my\_vector = std::vector<std::int32\_t>(*NUM\_ELEMENTS*, 0U);  
  std::generate(my\_vector.begin(), my\_vector.end(), gen);  
  // print\_vector(my\_vector);

(same gen from earlier in that file)

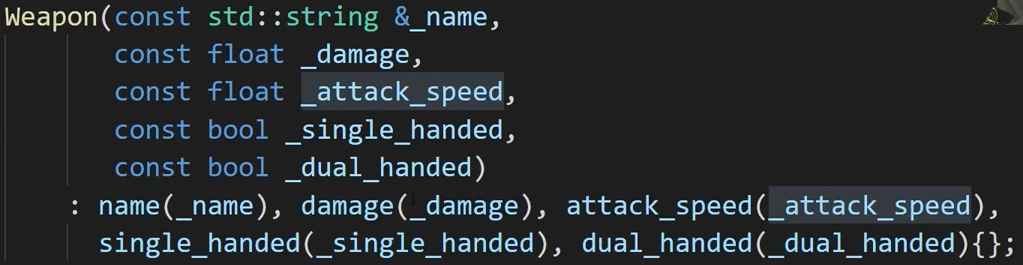
* **std::transform**: a method that allows changing the values of a container with a (lambda or not) function of choice.,
* **std::accumulate**: a method that allows summing up container values of choice, with a starting sum. Note: if you want to apply a calculation for each value before accumulating it to the sum, can also add a binary operator (can be a lambda function),
* **std::remove, std::remove\_if,,**
* **std::replace, std::replace\_if**: a method that iterates through a container and looks to replace a value (if exists) with a different value of choice. The replace\_if allows having a unary (single) function to live up to the “if”,
* **std::sort**: allows sorting a container in a pre declared sorting condition,
* **std::all\_of, std::any\_of, std::none\_of:,**
* **std::function**: a container that allows holding a set of functions like this:

**Classes**

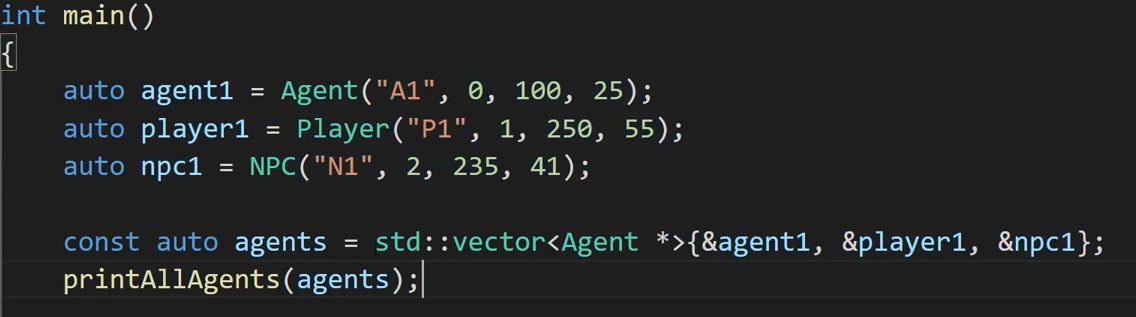
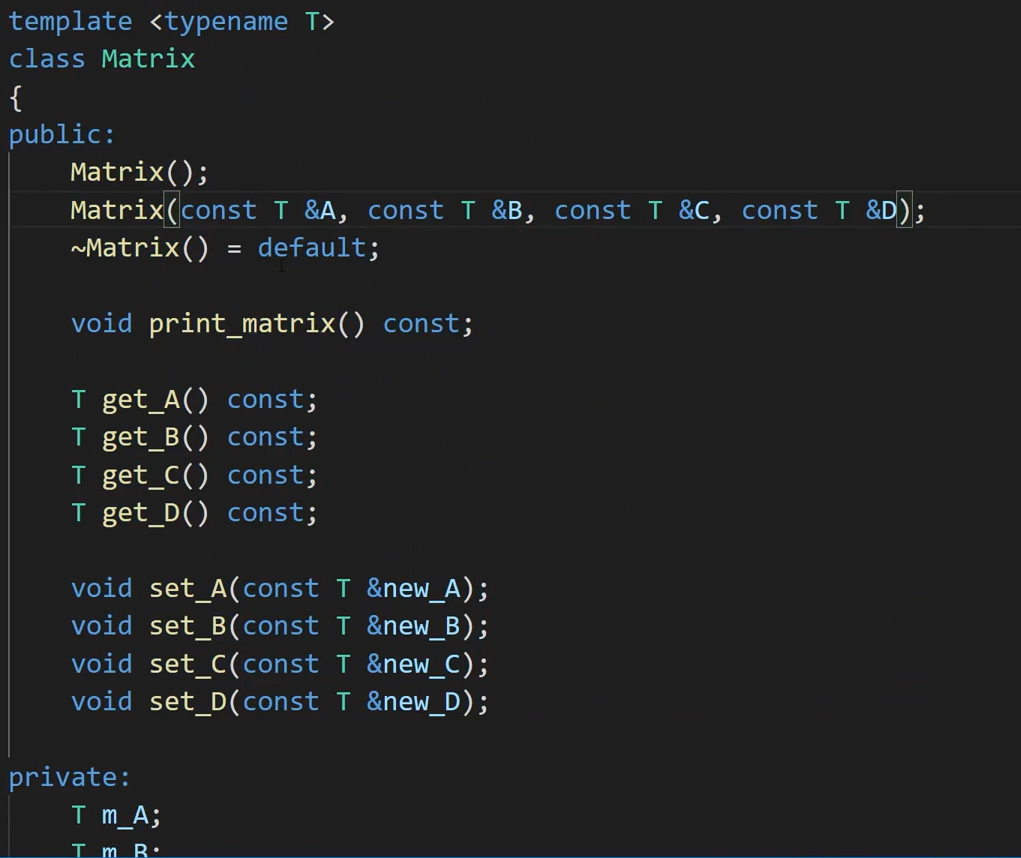
A computer screen shot of a program code

AI-generated content may be incorrect.

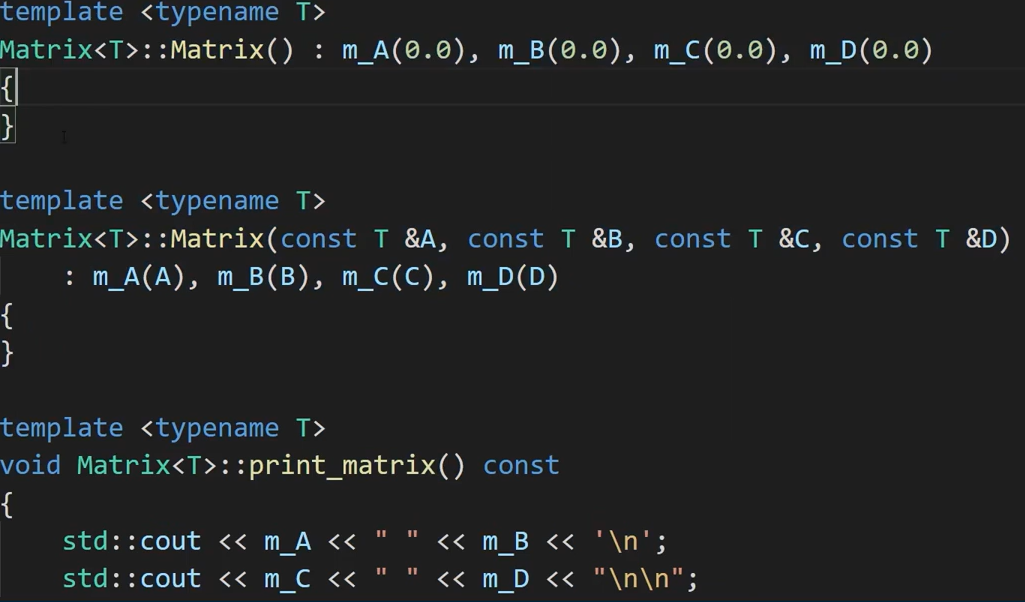


* **Interface (abstract in CPP):** if we have a “virtual” keyword in a prototype we are in an abstract class.
* **Abstract and Pure abstract**: pure abstract class objects can not be instantiated (like interface in Java) but Abstract class objects can be instantiated. (like abstract class in Java)
* **Constructors**: looks like in the following image, must implement the class initializer list in order of initialization in class attributes

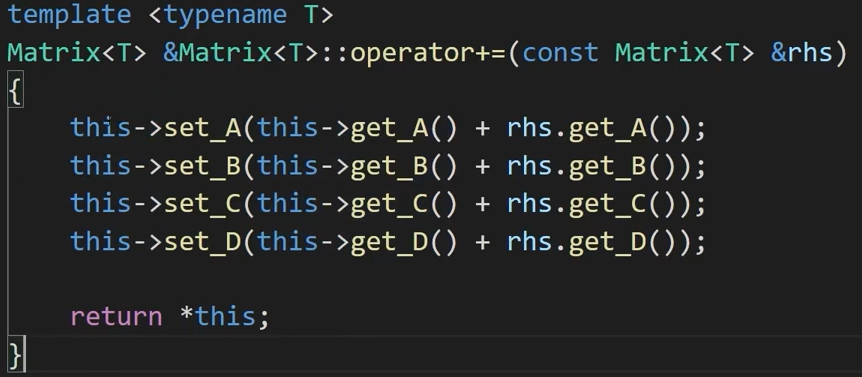


* **final and override keywords:** when inheriting methods from a class you can define them as final or override – final is when it is the final definition for the class and cannot be changed afterwards, override: if you have a child class that will override the method you can leave it as override. 
* **polymorphism:** In the following example Player and NPC Inherits from Agent and overrides (with “final” keyword) the print method – and with the use of pointers it will print each object using the respective print method.****
* **Template Class**: An example for a matrix template class – defined this way in the header file (like all classes – they are defined in header file) 

An example for initializing template class methods:



An example for defining operators within the template class:



**End of Classes**