**C++**

**Structure Of Program**

* **<<** is used for concatenation.
* In **C++**, main function must be compulsory **int main()**.

**Doing Math**

* Letter **f** after typing value is compulsory to specify float, not double.



* In C++ floats when printed, are shown upto the decimal point necessary, unlike C.

**For example:**

**In C++ 5.180 -> 5.18**

**In C 5.180 -> 5.180000**

**More On Variables**

* Keyword **auto** in C++ = Keyword **var** in C#
* **Pascal case:** TheBadassEngineer
* **Camel case:** theBadassEngineer

**Taking Input**



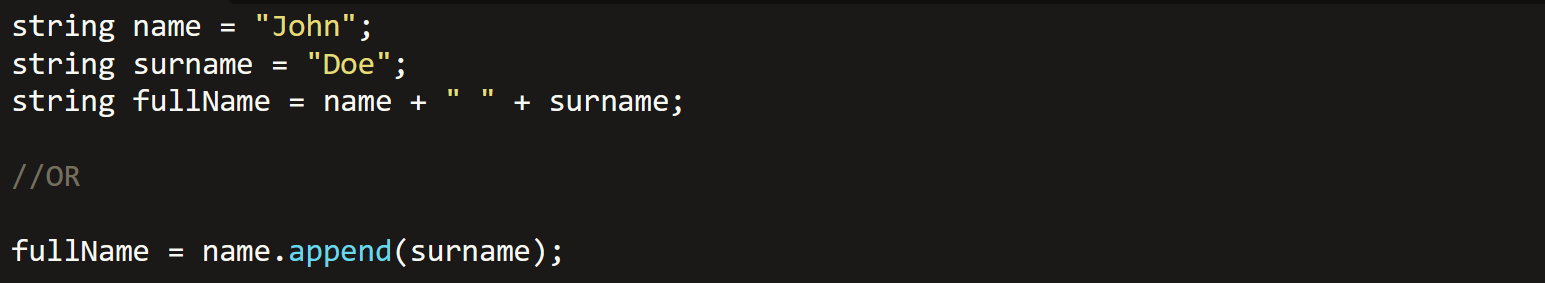
For reading whole **string** (and **not** stopping at **space**):-



**The Switch Statement**

* Forgetting to add break at case end, results in executing all cases despite wrong value.

**String**



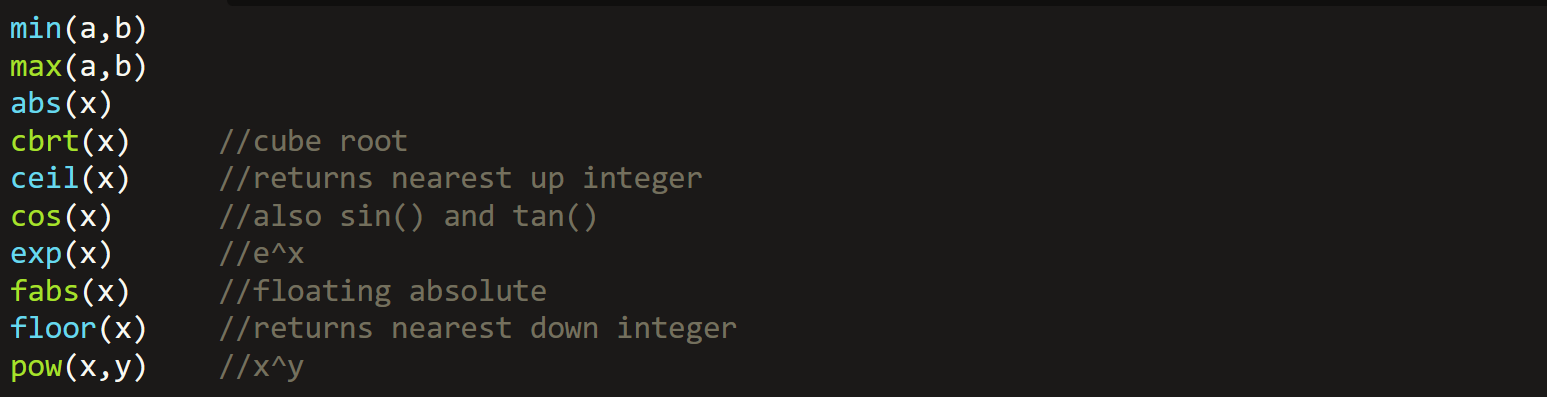
* The append function above appends **surname** to **string name** also.
* Using **<string>** header:-



* Namespace **std** is used for **string** and **cout** functions.

**Math**

* Using **<cmath>** header:-



* **Not-defined** answers return a **string “nan”**.
* The inputs for **trigonometric functions** must be in **radians**.
* In **pow()**, if raised power is **not** a whole number **neither** float with only zeroes after decimal point, then **“nan”** is returned.

**Array**

* This returns size of array **in bytes**, not as per the current number of elements in array but as per **defined size** of array:



**Structures**



* When **pointers** are defined for structures **without typedef**, then the pointers are blamed by the compiler & debugger.

**Function Parameters**

* We can also pass whole array as parameter:-



**Classes/Objects**

* Access modifier in C# = Access specifier in C++
* Class member in C# = Attribute in C++
* Semicolon is necessary after class block.





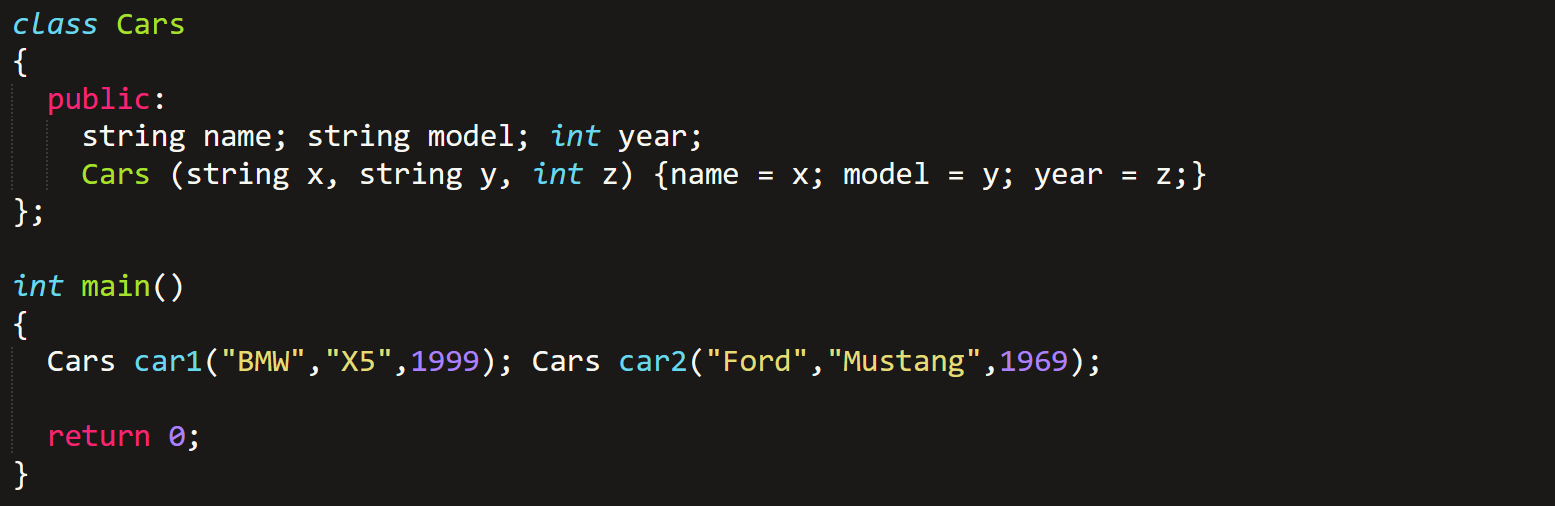
**Class Methods**



* To write method the way shown above, it must be first declared in class the **regular** way.
* **Void function** can **never** have a return type, whereas **non-void** functions may **omit** return statements if they want to.

**Constructors**

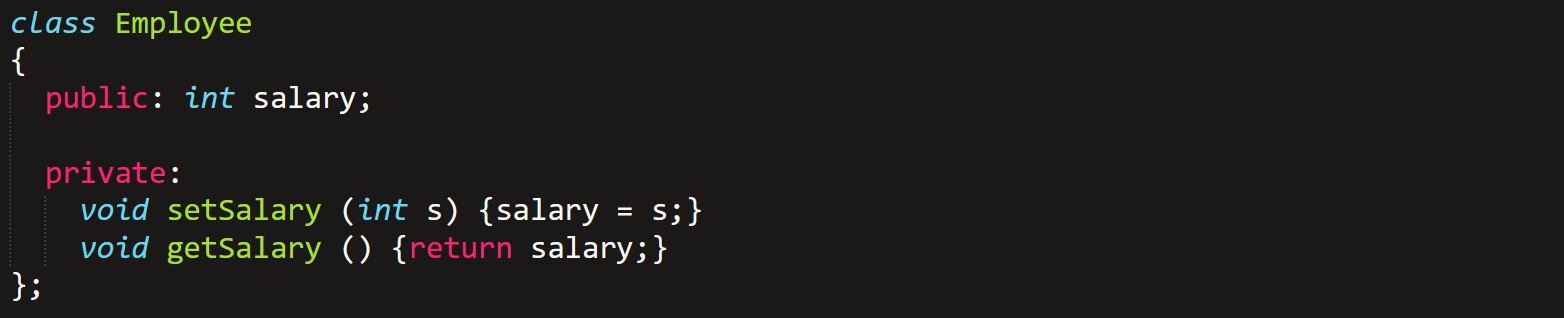
* Are always public.
* **Can’t** have a return value.
* It **is a** special **method**!
* Assigning value to object attribute (shortcut):-



* It is advised to **not** keep the name of **arguments** same as **attributes**.
* There exists a ***this*** keyword!

**Encapsulation**

* **Set & get** methods to access and modify private attributes:-



* It however is **not** working at all, I tried.

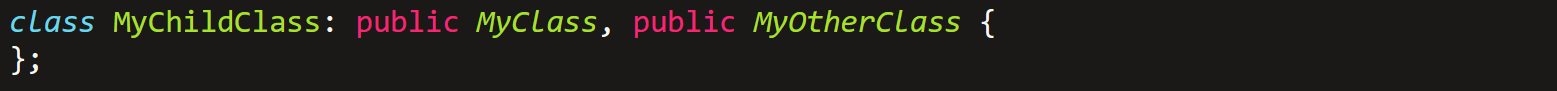
**Inheritance**

* We call it **derived class (child)** and **base class (parent)** in C++.
* Done the same way as in C#.
* But you should declare inherited class as **public**, to allow operations on it. As shown below:-
* Inheritance **+** providing access specifier to the class:-



**Multiple Inheritance**

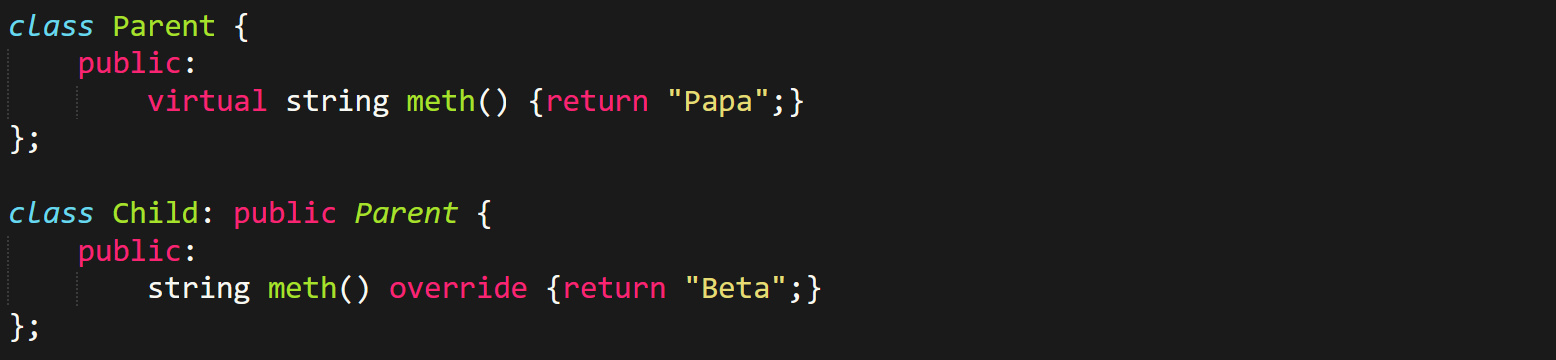
* Deriving a child class from multiple parents.



* In this, there **must be no common attribute** among parents.
* Constructors are **not derived**.
* When deriving from multiple parents, it is advised to **not involve** any parent attribute in any parent constructor.

**Polymorphism**

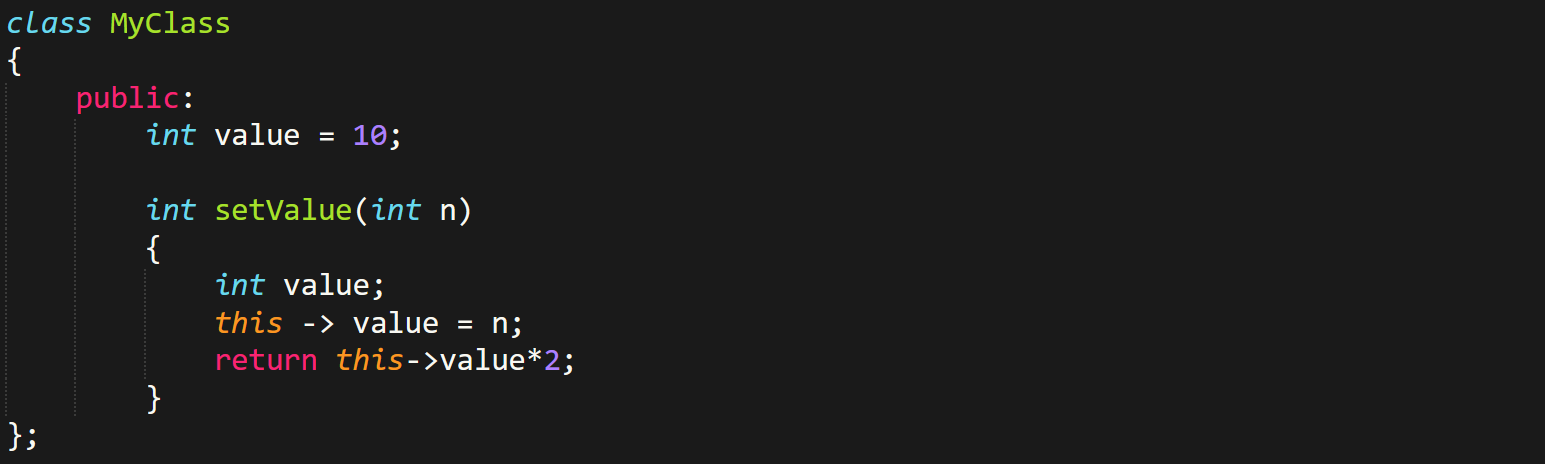
* Parameter’s type and quantity **must be same** in each class’s overridden methods.
* **virtual** & **override** keywords are used.



* Never use any **iostream/std** function like **cout** or **cin** for **void methods**, else your life will become **hell**!
* Just call **void** methods **directly**.
* And methods with **return type** must use them.

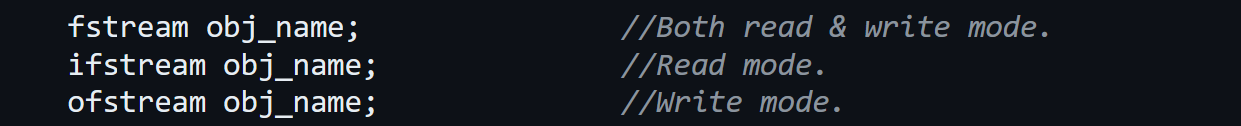
**This Keyword**

* Used to distinguish between **class attribute** & a **method variable**.
* Like shown below, value is both an **attribute** & a **local variable** in **setValue()**:

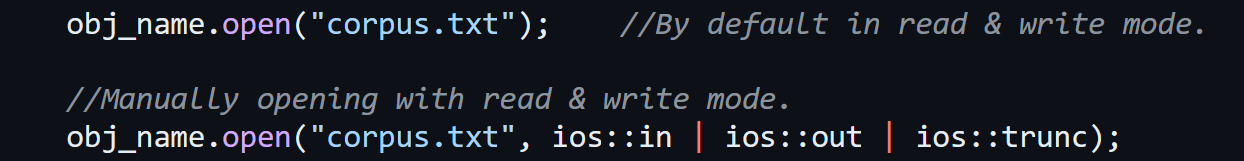


**File Handling**

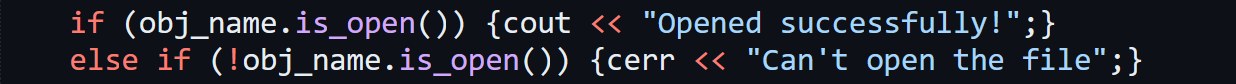




* ***obj\_name*** in C++ is similar to ***fptr*** in C.

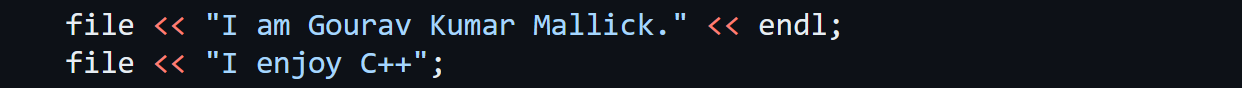


* ***ios::trunc*** overwrites a file if it **exists** & creates if it **doesn’t** exist.
* **Not** using it **won’t** create a file.
* ***ios::app*** is used for append.
* One of the both ***(ios::trunc or ios::app)*** must be selected.

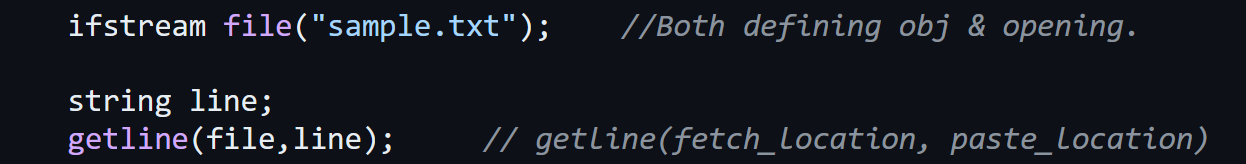




Writing to file using ***ofstream***:-

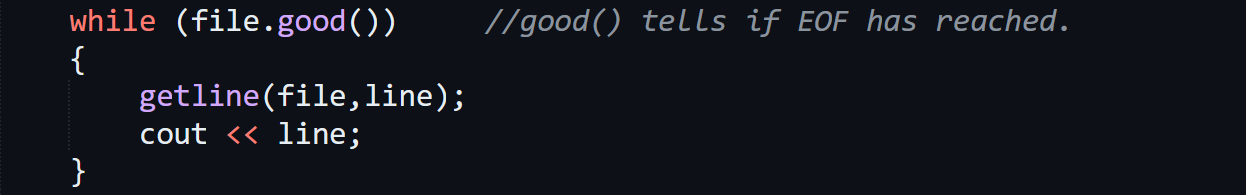


Reading from file using ***ifstream***:-



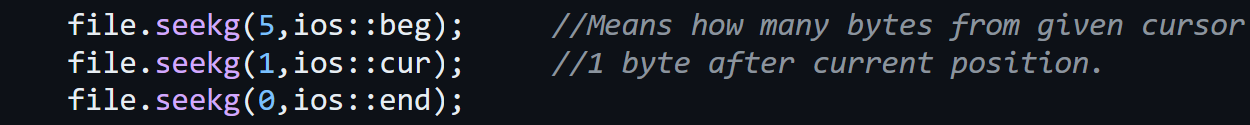


Getting all the lines:-



Moving cursor somewhere:-





Check cursor positions:-

* Read & write are done using separate cursors.

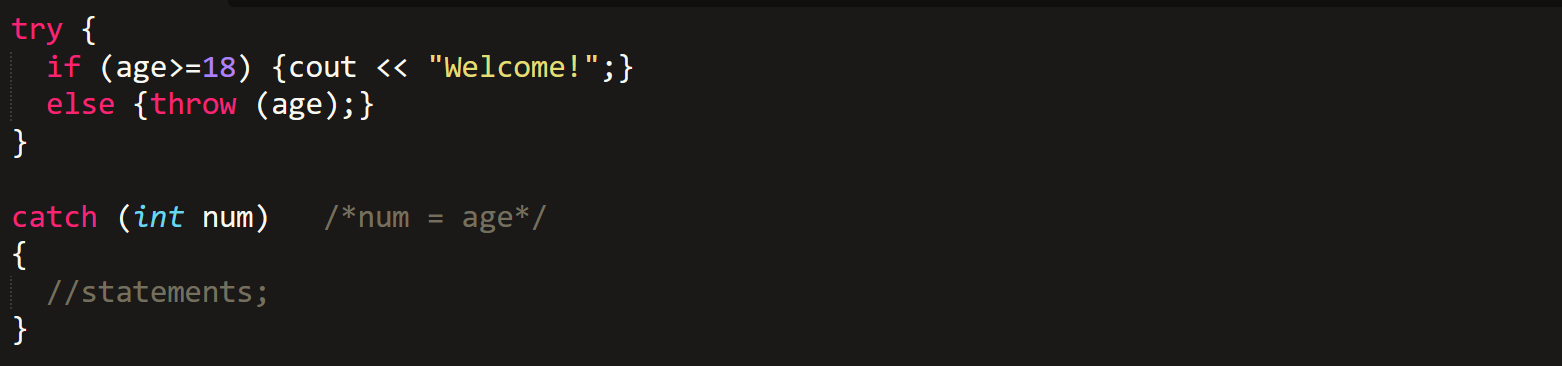


Binary mode:-

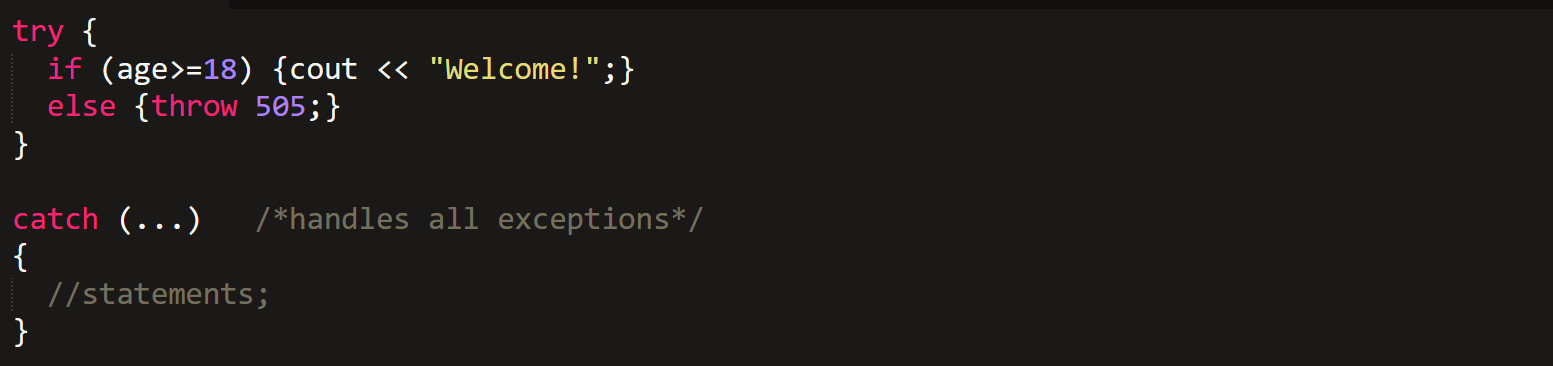


* Notice the ***ios::binary*** written.

**Exceptions**

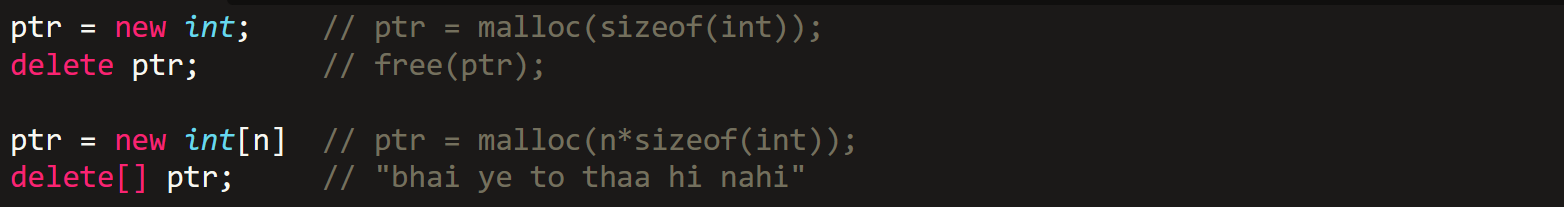


* If our throw is not an identifier, then we can use three dot (…) in catch:-



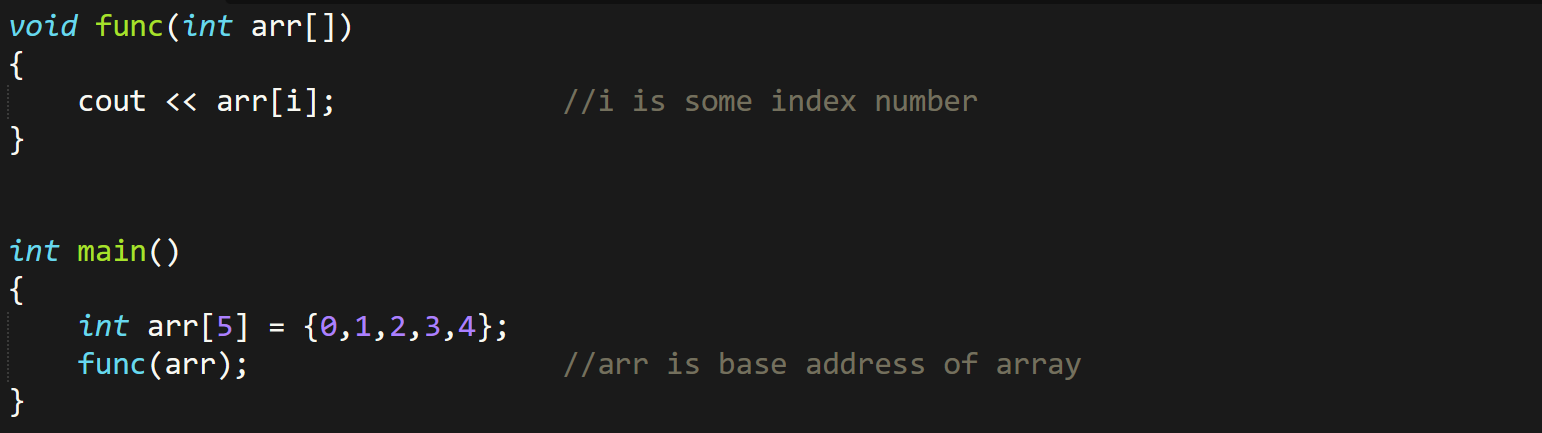
**Memory Management**

* We use **<cstdlib>** to include C’s standard library into C++.



**Passing Array Argument**

1. **With reference:-**



* **arr[]** is same as **\*arr**.

1. **Without reference:-**

* Pass as an element or just use your brain buddy!

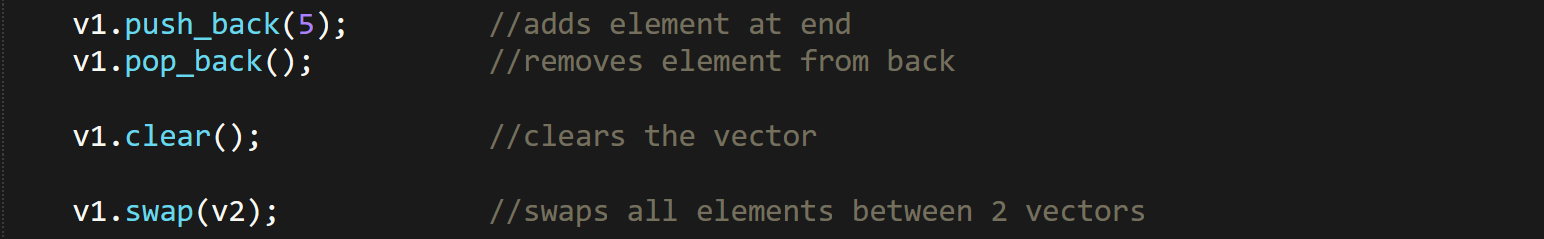
**Vectors**

* Declare header file ***<vector>***.
* Vector traversal **isn’t** done from beginning, just like **arrays** & unlike **linked lists**.
* So vectors are **faster** than linked list.

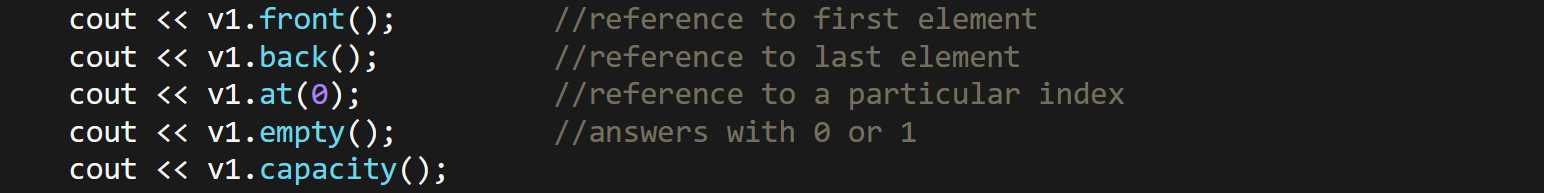
1. **Declaration:-**



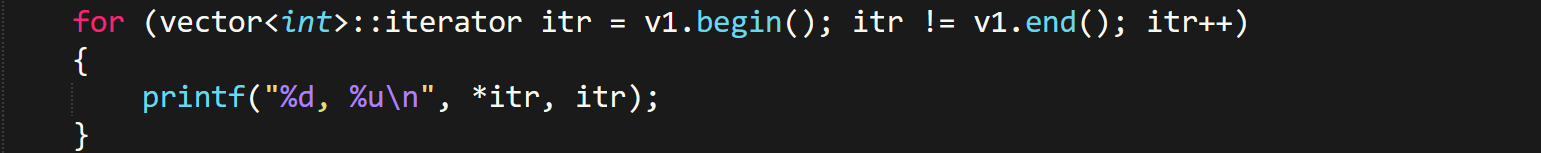
1. **Functions with void data types:-**



1. **Functions with return types:-**



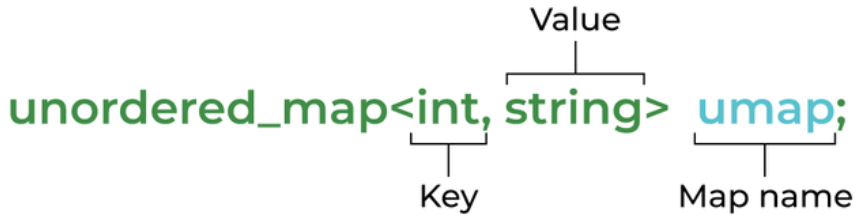
1. **Iterating/traversing through vectors:-**



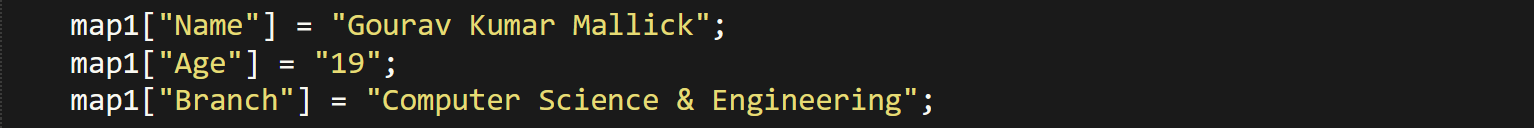
**Unordered Map**

* Uses **<unordered\_map>** library.
* Values are hashed to keys & keys are hashed to indices.

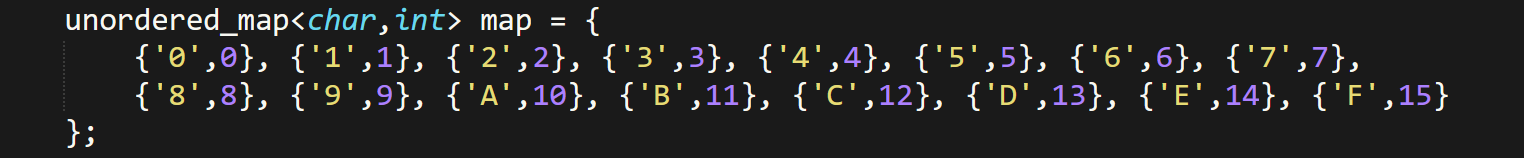
1. **Declaration:-**



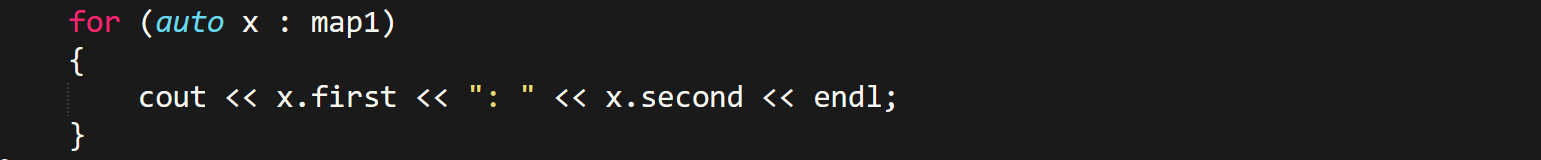
1. **Putting values:-**



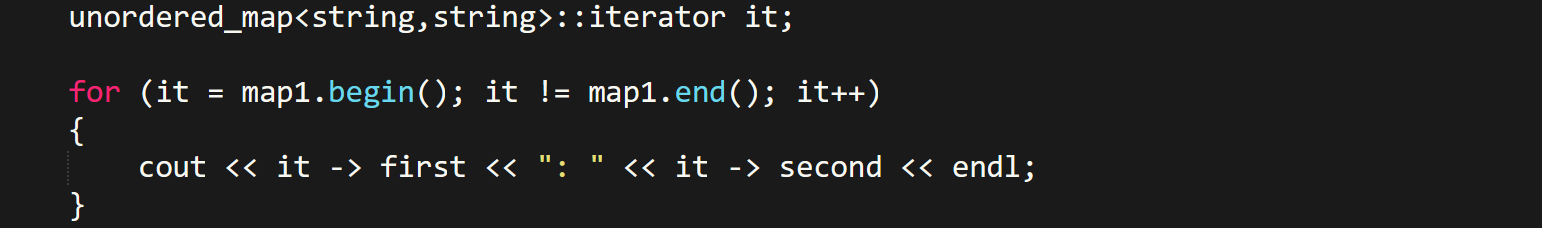
**OR**



1. **Traversal/iteration:-**



**OR**

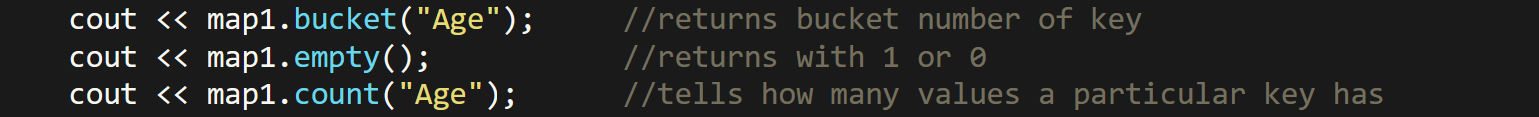


* **begin()** and **end()** are used with an **iterator only** as shown above.
* **end()** is place after crossing the last pair in memory.
* It traverses **in reverse**.
* Address operations **don’t** work on unordered maps.
* Time complexity of unordered map is **O(1)**, contrary to **map** with **O(log(n))**.

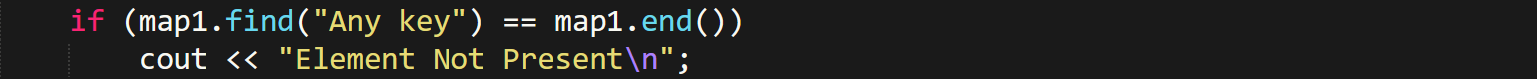
1. **Function with no return type:-**



1. **Functions with return type:-**



1. **Functions to be used in control statements:-**



**Generic**

* **Generic:** Class or function that is defined as a **template**.

1. **Template declaration:-**



**OR**



* Both lines must be written together like this.
* Must be written after **headers** & **using namespaces** (if any).

1. **Calling function:-**



**OR**

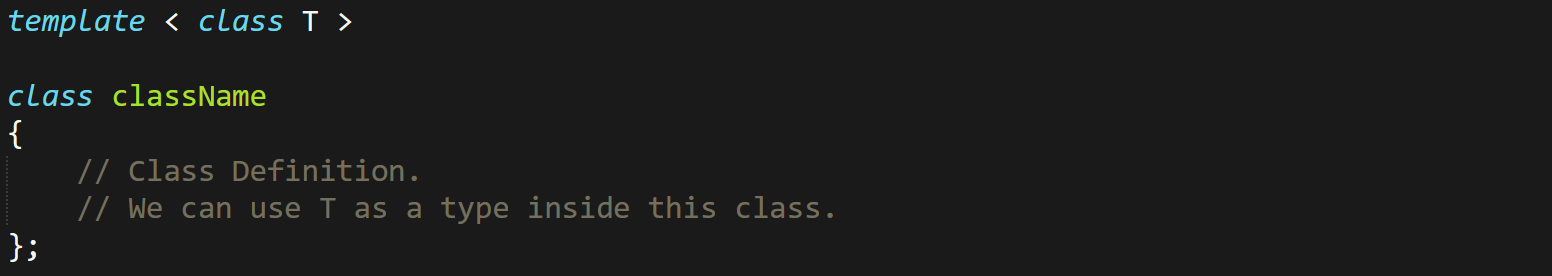


* In this one, compiler decides the data type by itself.
* ***auto*** keyword can’t be used.
* **Same template** data type (**T** here) can be used in multiple generics.
* Template **overloading** is possible.

1. **For passing objects as arguments:-**



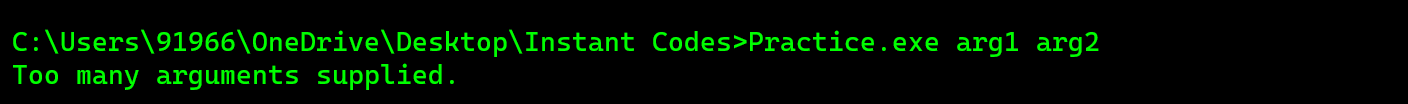
1. **Using templates for classes:-**



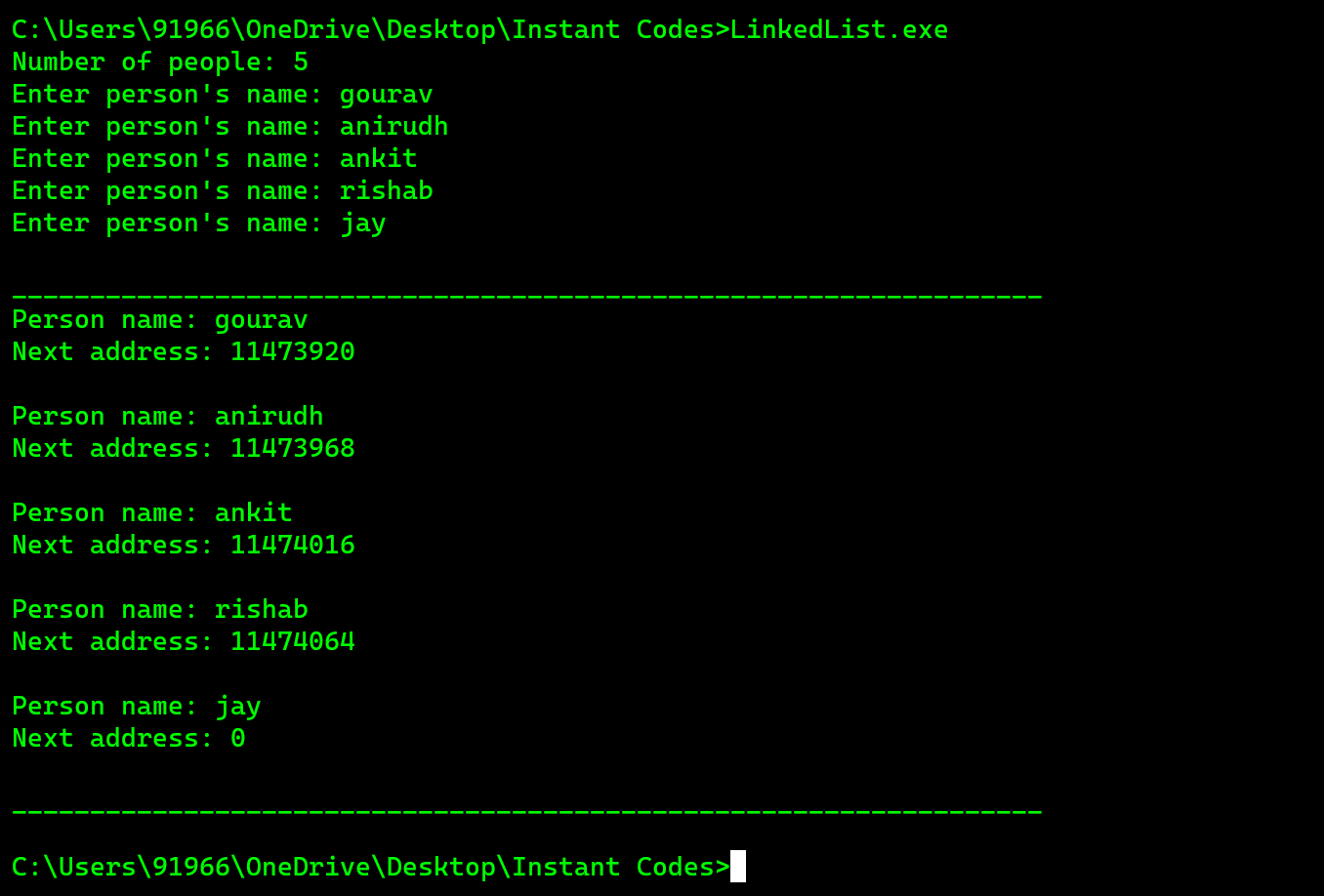
**Command Line Arguments in C/C++**



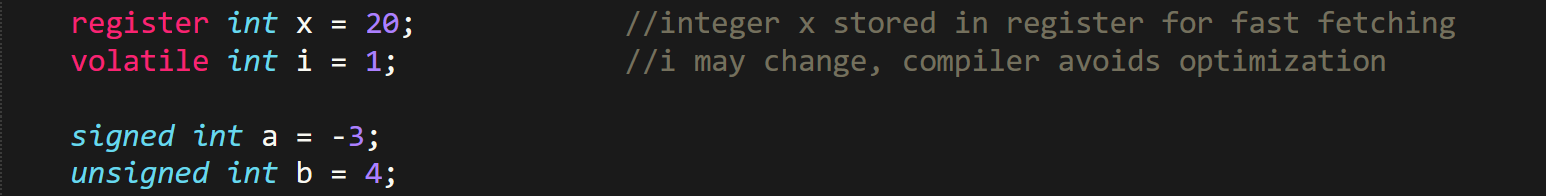
* ***argc*** = Argument count, ***argv*** = Argument values
* ***argc*** is by default ***1***, which is the **file name** itself.
* ***argv[0]*** is always the **file’s name**.
* **Code written in command prompt (example):**



* **Linked list program run using *cmd*:**



**Some Newly Observed Keywords**



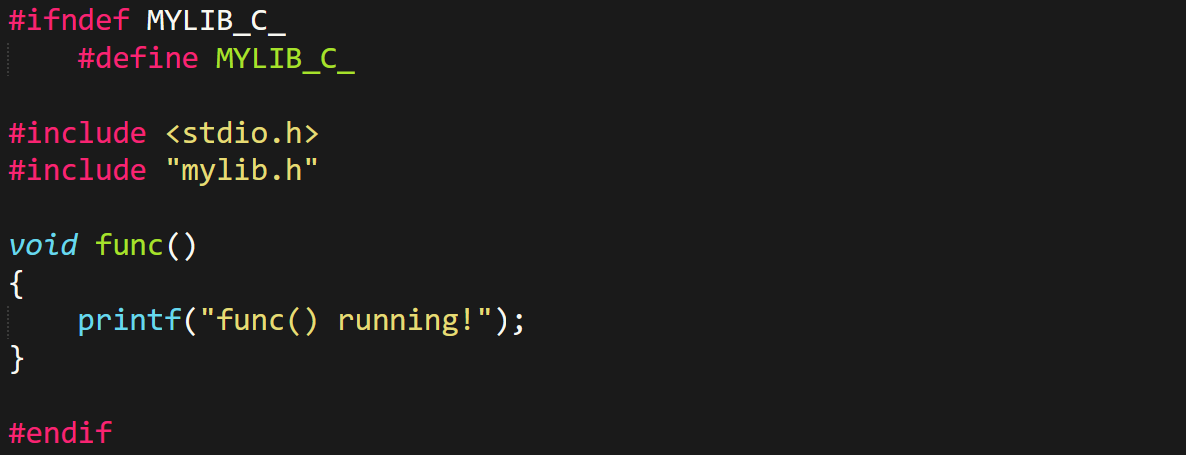
**C/C++ Libraries**

* The **object files** work as **intermediate** between a program and library.
* **Dependency walker:** A tool used for troubleshooting incompatibleness of ***.dll*** files.
* Generally, **static files** create **larger** binary files & thus occupies **more space**.
* In whole library process, ***gcc*** is used for **C** files & ***g++*** is used for **C++** files.
* Windows uses ***.obj*** extension for **object files**.
* Linking is done for running an **executable** which uses library(s).
* We can remove the including statement importing the library at **final stage** of project.

**Static Linking & Static Libraries**

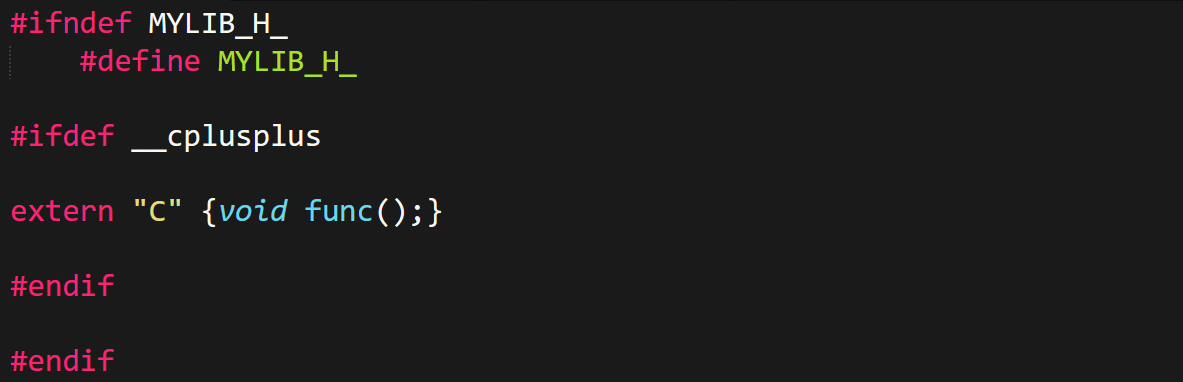
Creating (developer’s job):-

* **Step 1:** Create a **source file** containing **implementations** of the library:



***\*We can create such multiple files\****

* **Step 2:** Create a **header file** with only **function definitions**:



* **Step 3:** **Compile** library file(s) into **object files**:



***\*Gotta do this one by one for each .c file\****

* **Step 4:** Create **static libraries**:



***\*For multiple .obj files, add all file names in the command in line\****

Linking (user’s job):-

* **Step 1:** Create a **user program** importing our library:



* **Step 2:** **Compile** the user program into an **object file**:



* **Step 3:** **Linking** user program to **static library**:



***\*This creates an executable of user program which can be run\****

**Things to Take Under Consideration**

* It’s a good practice to add this to **ensure compatibility** of user’s **C/C++ version**.



* Use ***extern "C"*** for **functions** and **variables** (in **header files** only).
* But **don’t** use them for **class** or its **members**.

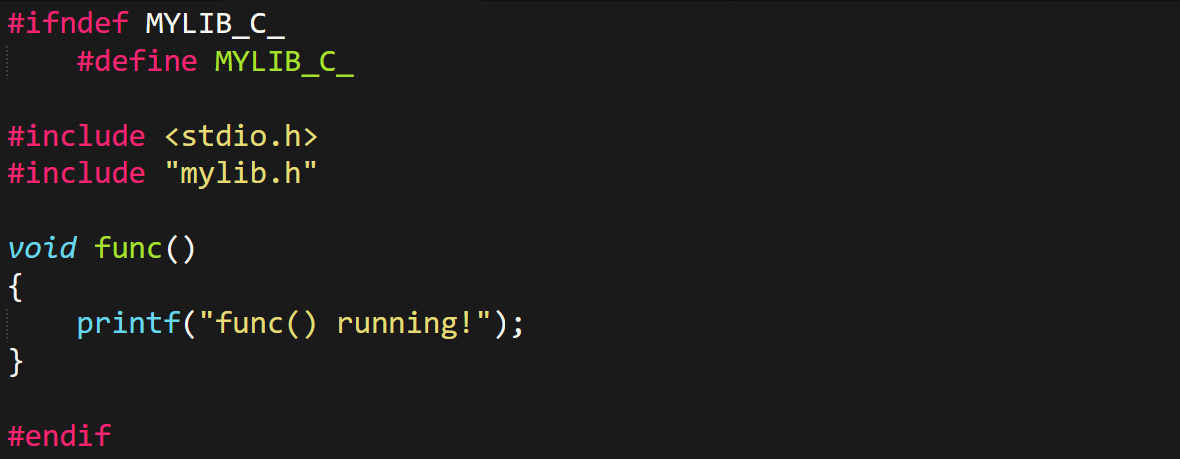
Project specific:-

* During testing phase, include with ***.c/.cpp*** file.
* Then after completing the project, **before linking** make them ***.h/.hpp***.

**Dynamic/Shared Files & Linking**

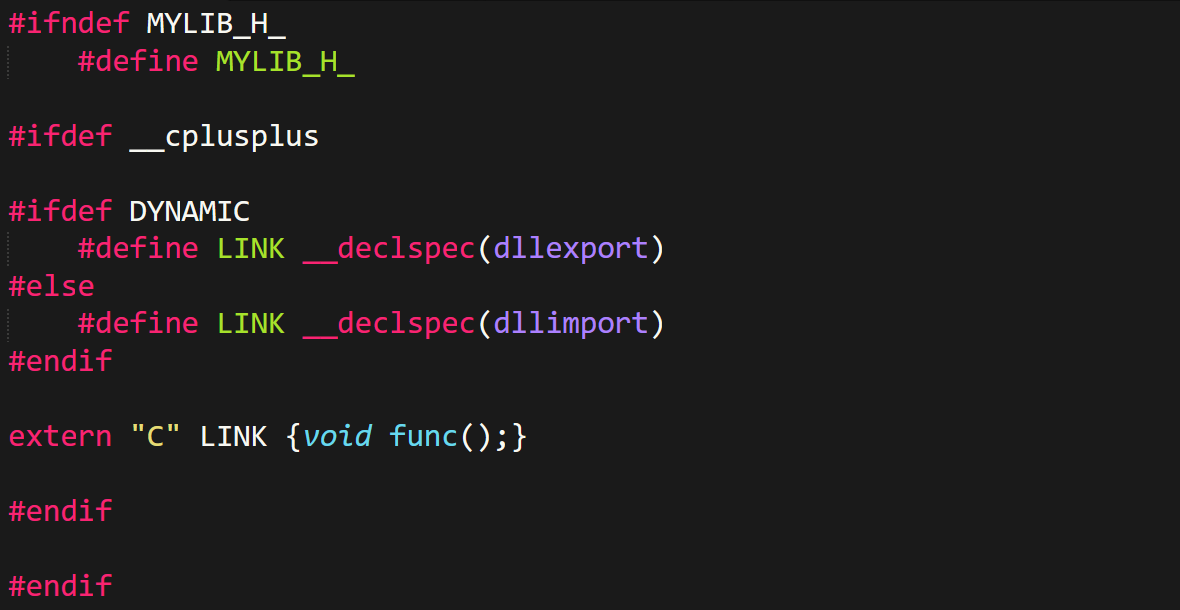
Creating:-

* **Step 1:** Create a **C/C++** file that contains **implementations** of the library:



***\*We can create such multiple files\****

* **Step 2:** Create a **header file** with only **function definitions**:



* **Step 3:** **Compile** library file(s) into **object files**:



***\*Gotta do this one by one for each .c file\****

* **Step 4:** Create **dynamic/shared** libraries:



***\*For multiple .obj files, add all file names in the command in line\****

Linking (user’s job):-

* **Step 1:** Create a **user program** importing our library:



* **Step 2:** Compile the **user program** into an **object file**:



* **Step 3:** **Linking** **user program** to **dynamic library**:



***\*This creates an executable of user program which can be run\****

**Library Structure**

Header file:-

* Guard **macro**.
* ***\_\_cplusplus***
* ***\_\_declspec(dllexport/dllimport)*** [if **dynamic**].
* Required standard **header files**.
* Namespaces.
* ***extern*** keyword.
* Declaration of **functions**, **structures** and **global variables** (if any).
* Declaration of class’s **public** members.
* Declaration & **definition** of class’s **private** members.

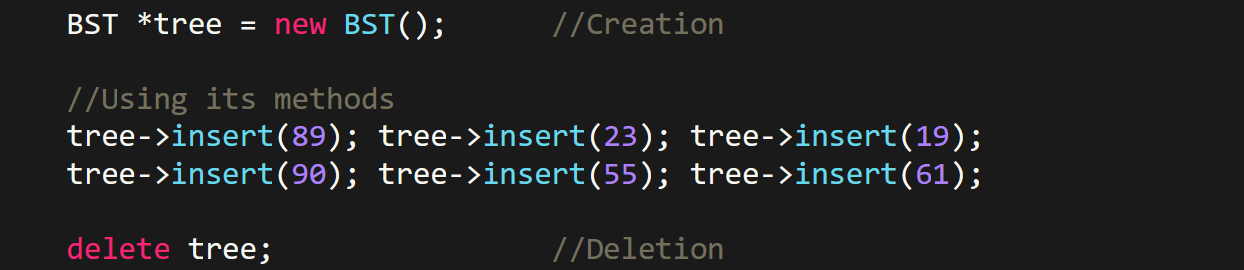
Source file:-

* Guard **macro**.
* Inclusion of **header file**.
* **Implementation** of declarations in header file.
* **Scope resolution operator** (***::***) for namespace references.

User program:-

* Inclusion of **source file**.

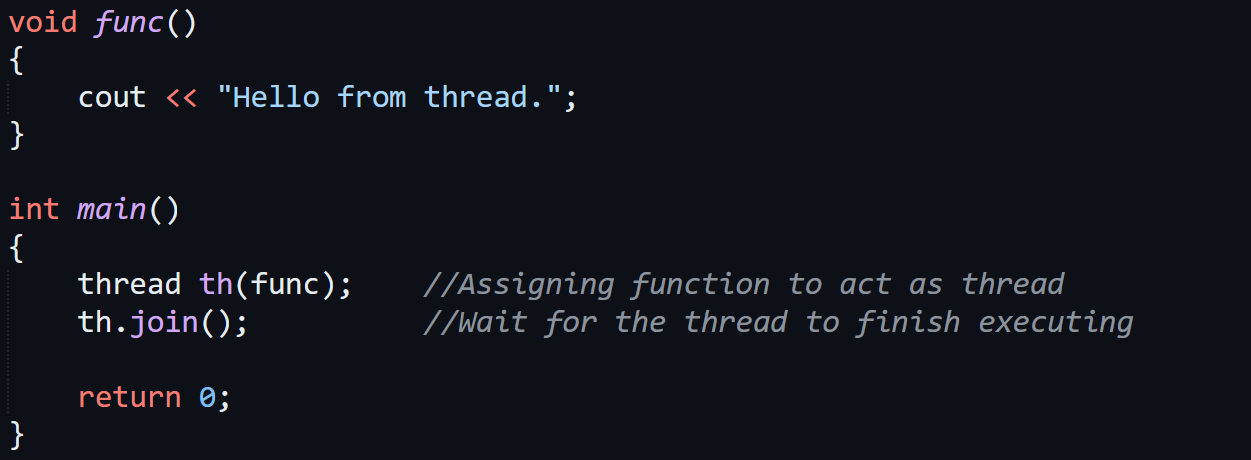
**Dynamic Class Objects**



* **Destructors** of ***dynamic class objects*** are **not** called automatically when the program ends.

**Multithreading**

Creation & join():-



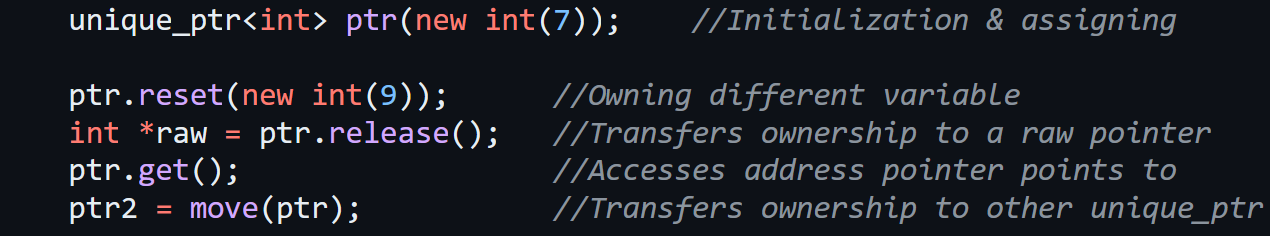
Thread ID:-



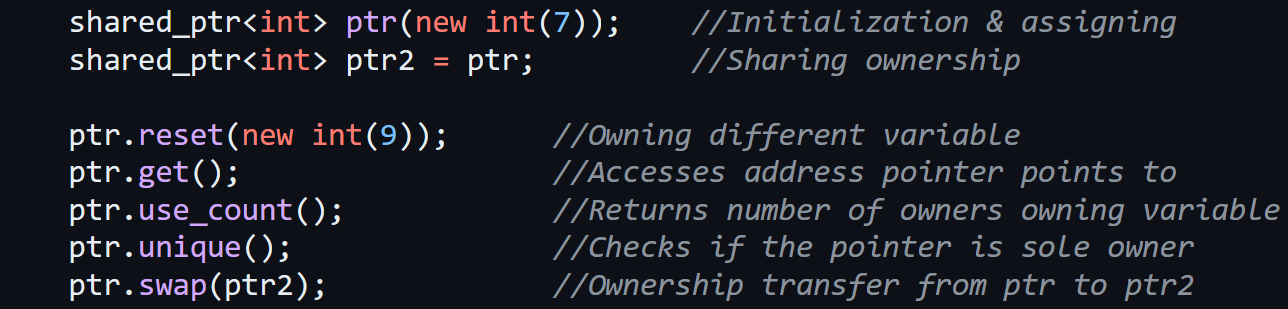
**Smart Pointers**



* ***Pointer*** is the **owner** & ***dynamic variable*** is **owned**.
* **Unique pointers:** Only **one** pointer can own a variable.



* **Shared pointers:** Ownership of a variable is shared among **multiple** pointers.



* **Weak pointers:** Share reference to a ***shared pointer*** but **doesn’t** participate in ownership of a variable.
* So, a ***weak pointer*** **can’t** manipulate value of the variable it is referring to through ***shared pointer***.
* Instead, it just **watches & tracks** the lifetime of an object pointed by ***shared pointer***.

