**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

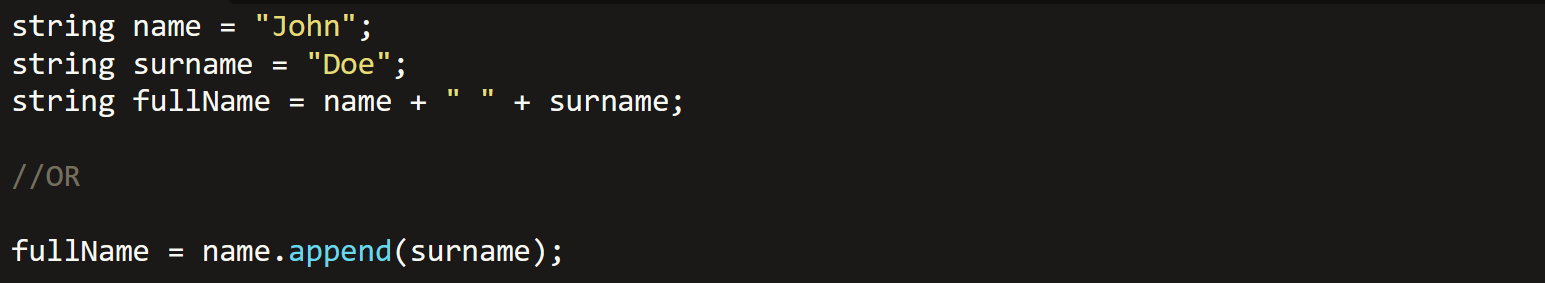
**Taking Input**



**The Switch Statement**

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

**String**



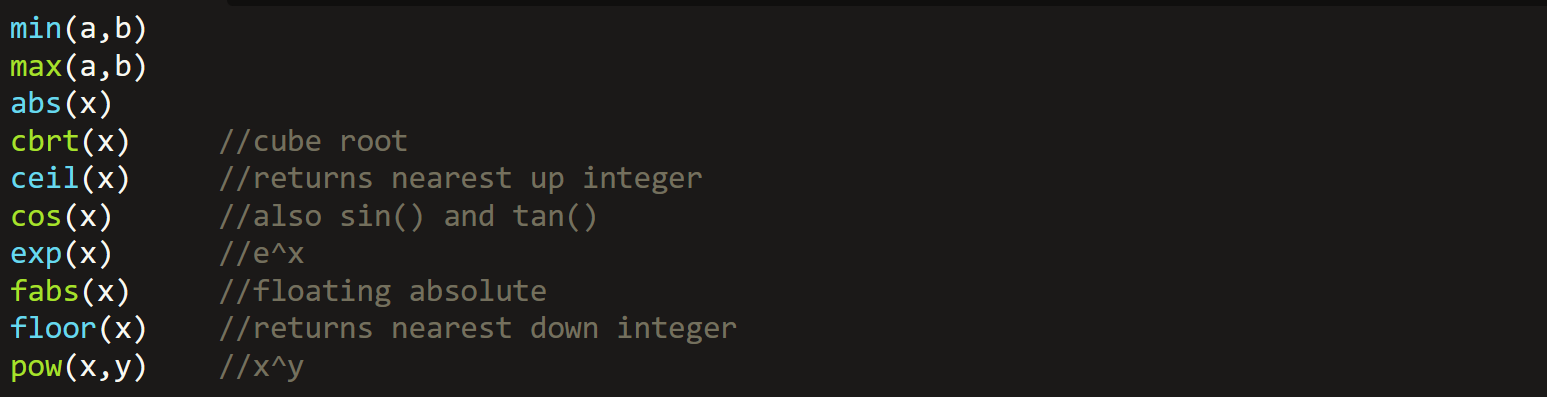
* The append function above appends **surname** to **string name** also.
* Using **<string>** header:-



* 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.





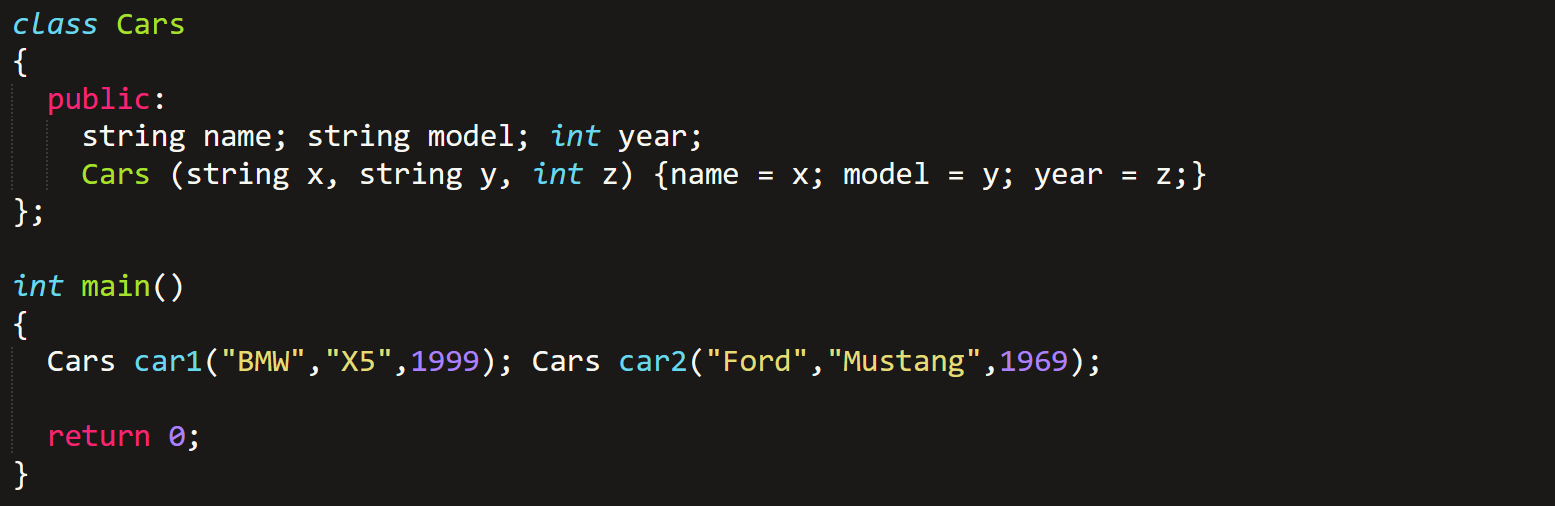
**Class Methods**



* 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**

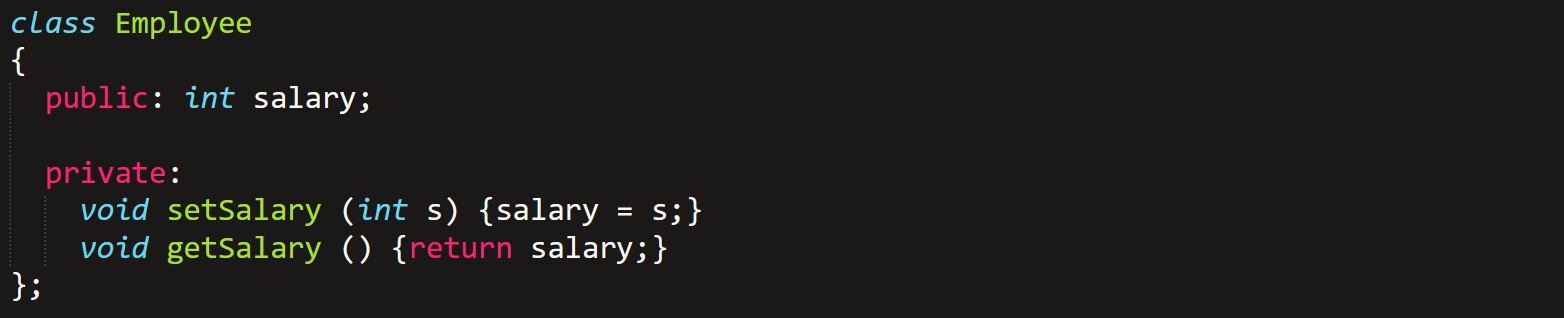
* Are always public.
* **Can’t** have a return value.
* It **is a** special **method**!
* Assigning value to object attribute (shortcut):-



* 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.

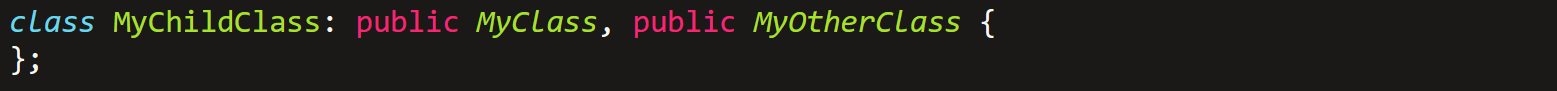
**Inheritance**

* 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**

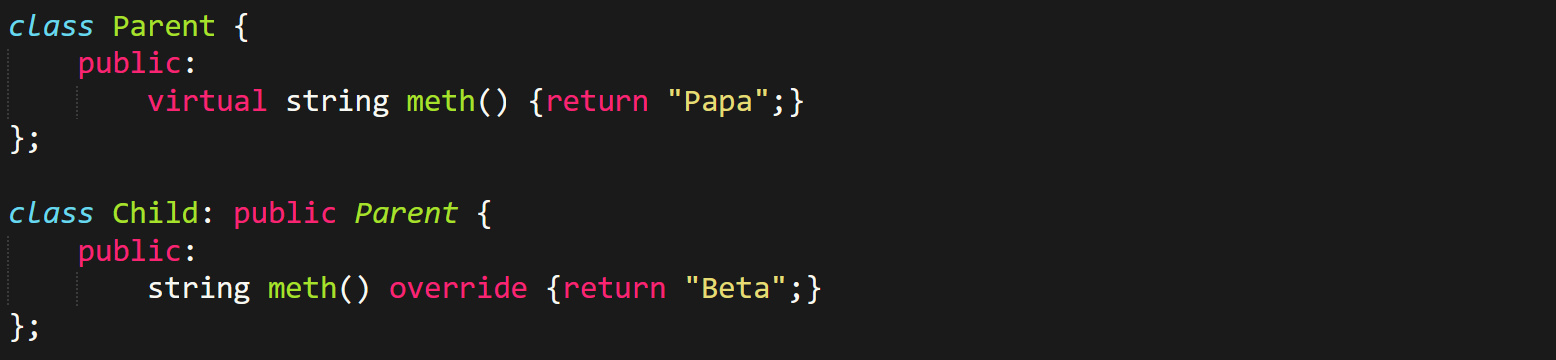
* Deriving a child class from multiple parents.



* 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**

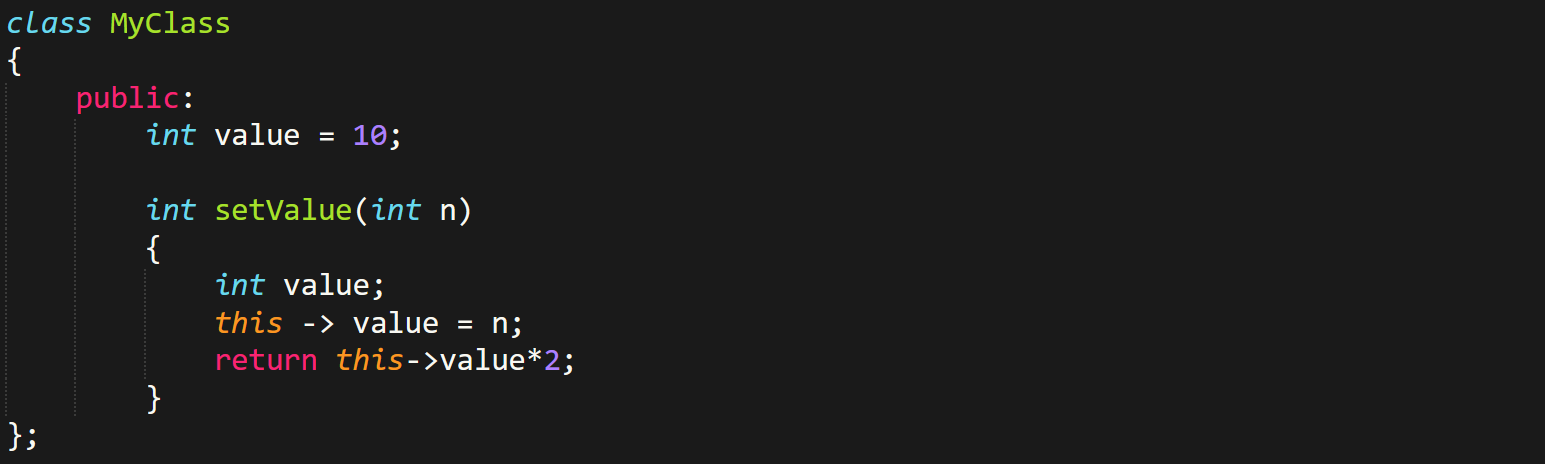
* Parameter’s type and quantity **must be same** in each class’s overridden methods.
* **virtual** & **override** keywords are used.



* 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.

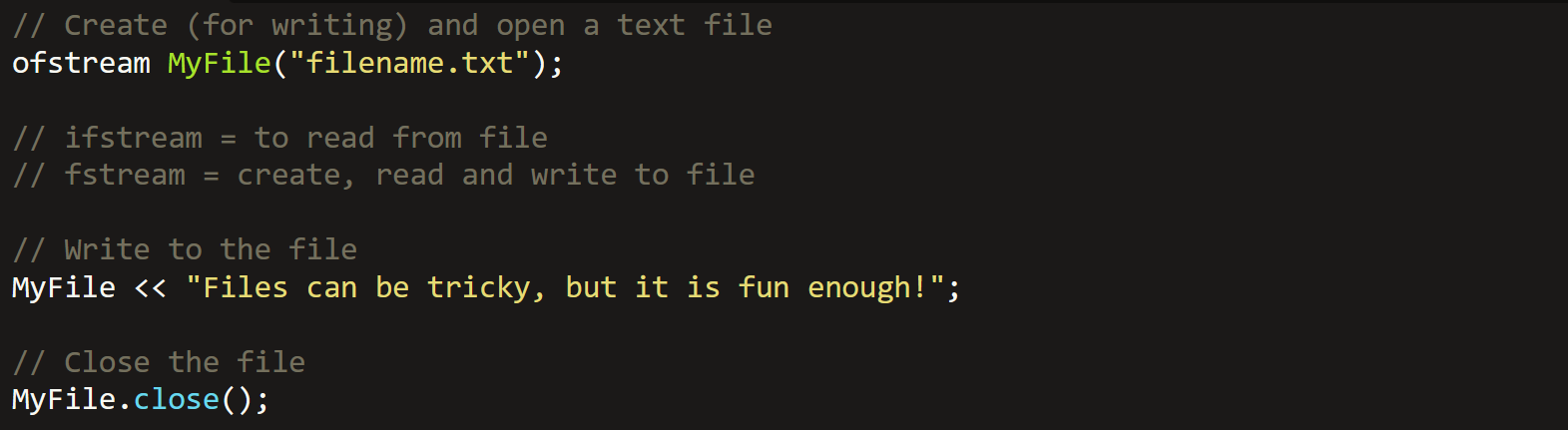
**This Keyword**

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

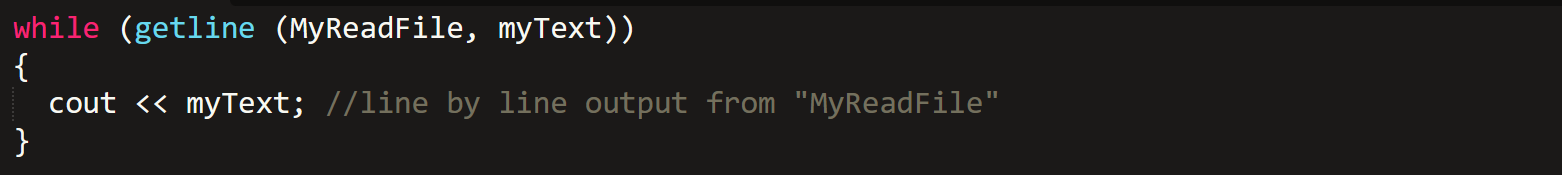


**File Handling**

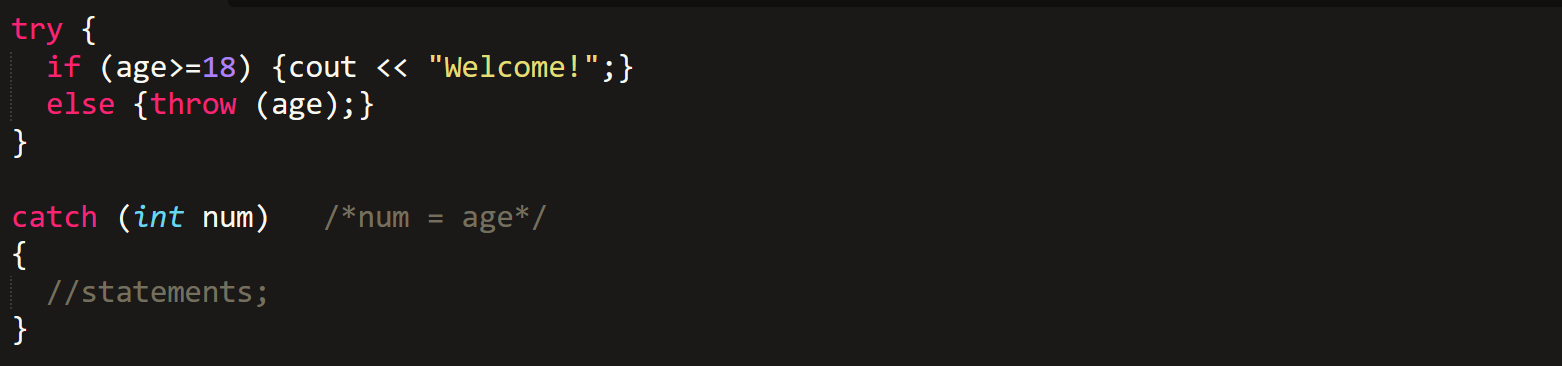
* We **include <fstream>** class for carrying out file related operations.



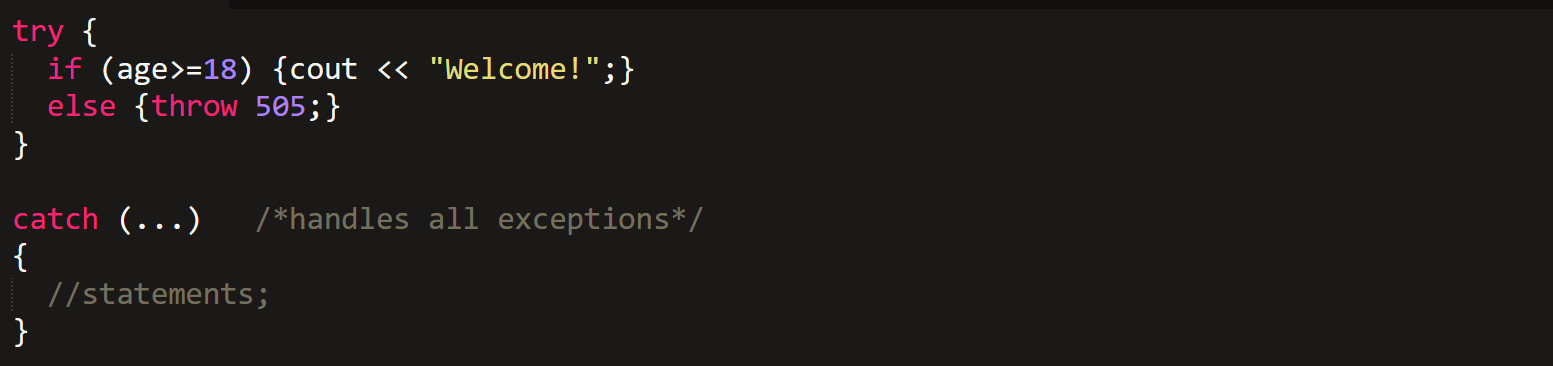
* The **“write to the file”** function writes actually when the file closes.
* To read from a text file line by line, we use **getline()** function:-



**Exceptions**

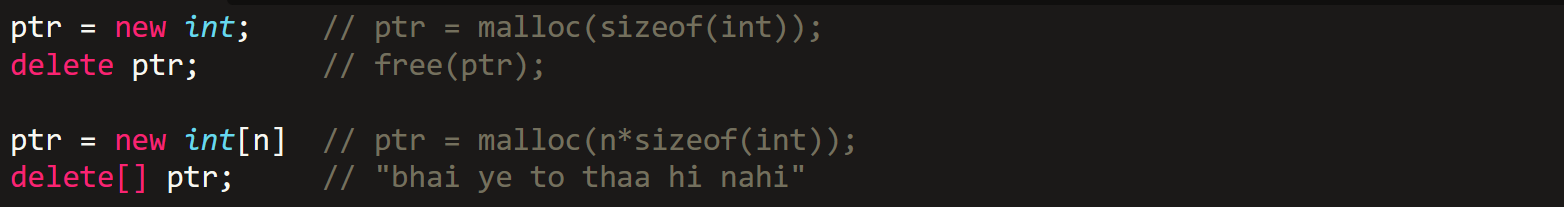


* If our throw is not an identifier, then we can use three dot (…) in catch:-



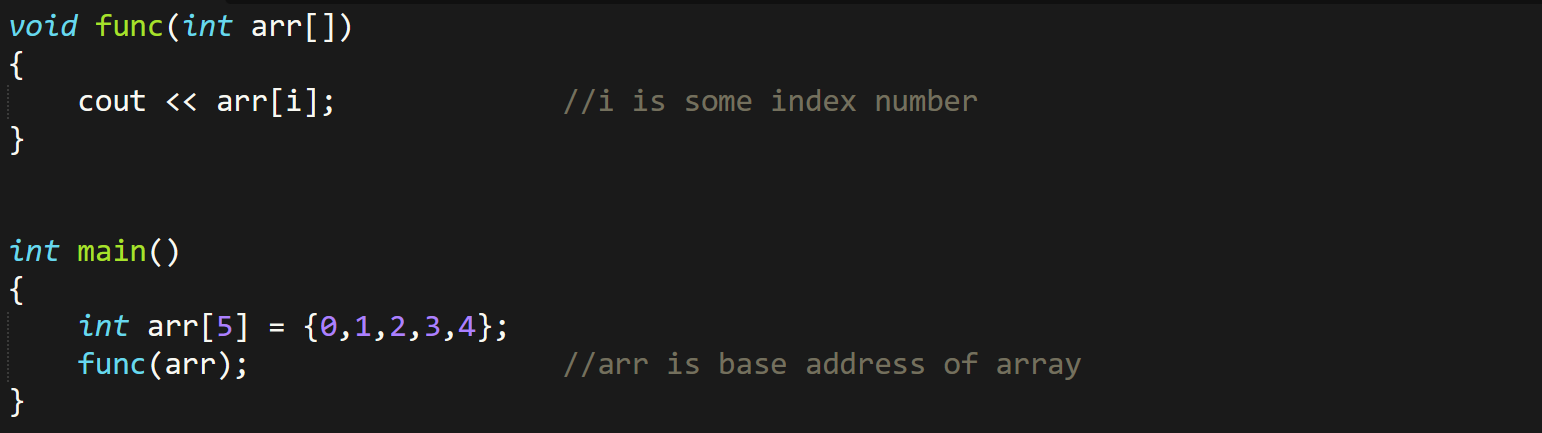
**Memory Management**

* 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!

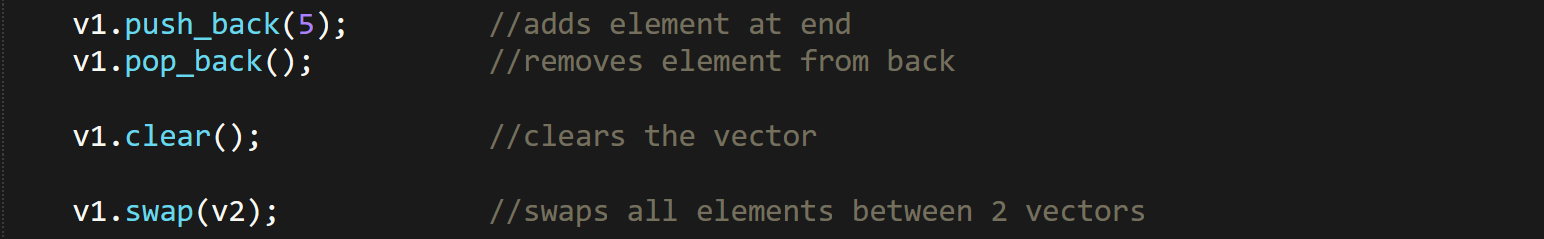
**Vectors**

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

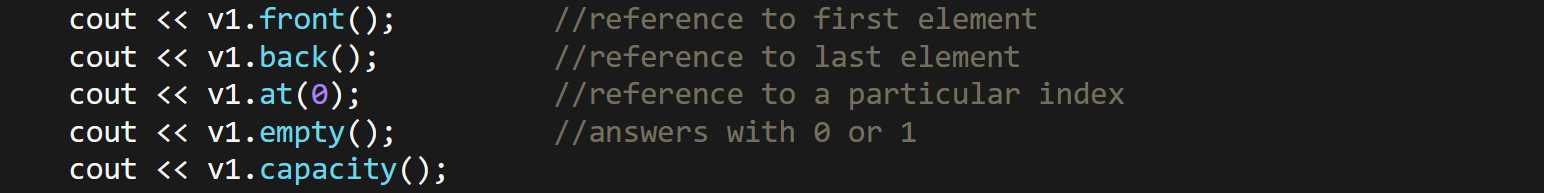
1. **Declaration:-**



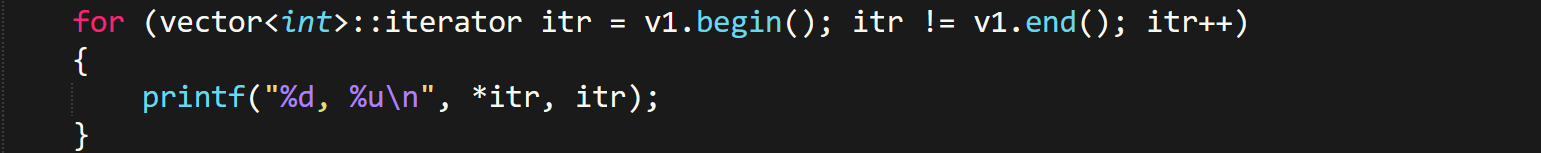
1. **Functions with void data types:-**



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



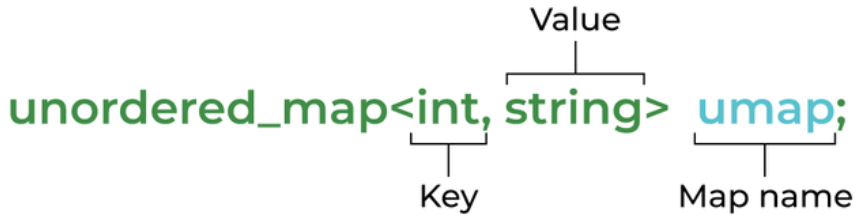
1. **Iterating/traversing through vectors:-**



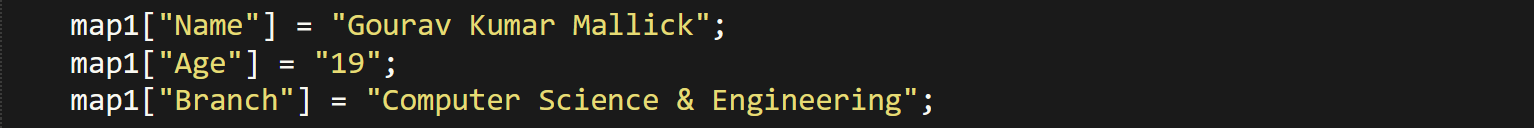
**Unordered Map**

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

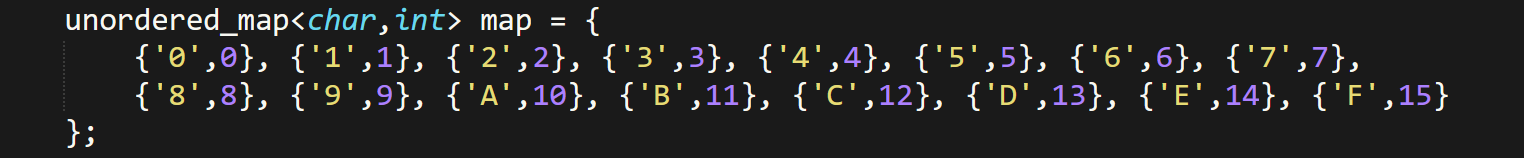
1. **Declaration:-**



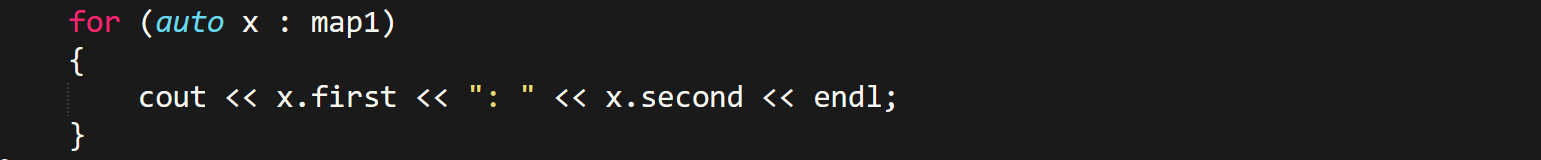
1. **Putting values:-**



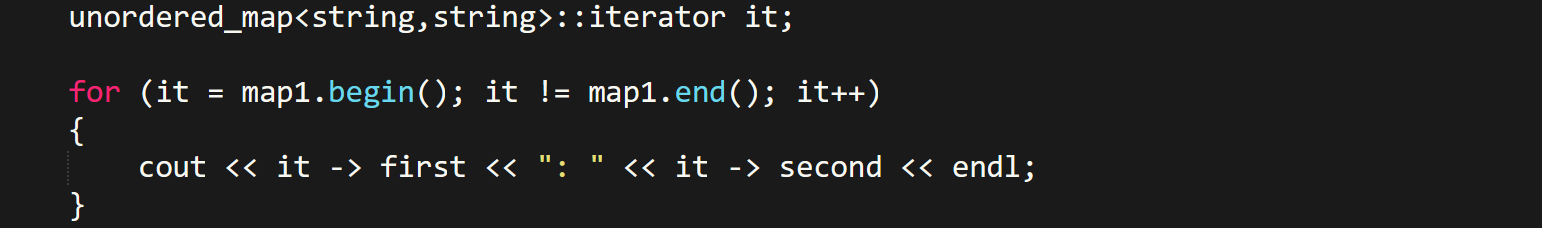
**OR**



1. **Traversal/iteration:-**



**OR**

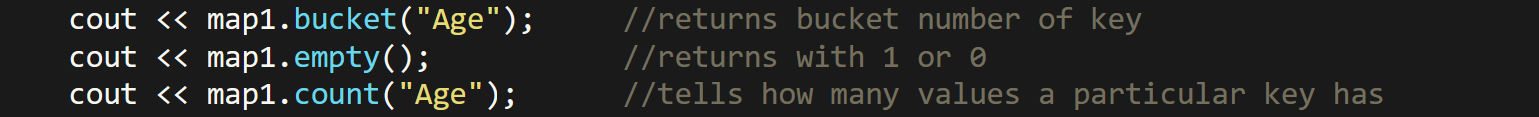


* **begin()** and **end()** are used with an **iterator only** as shown above.
* **end()** is place after crossing the last pair in memory.
* It traverses **in reverse**.
* Address operations **don’t** work on unordered maps.
* Time complexity of unordered map is **O(1)**, contrary to **map** with **O(log(n))**.

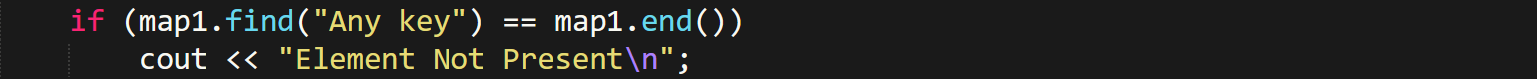
1. **Function with no return type:-**



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**

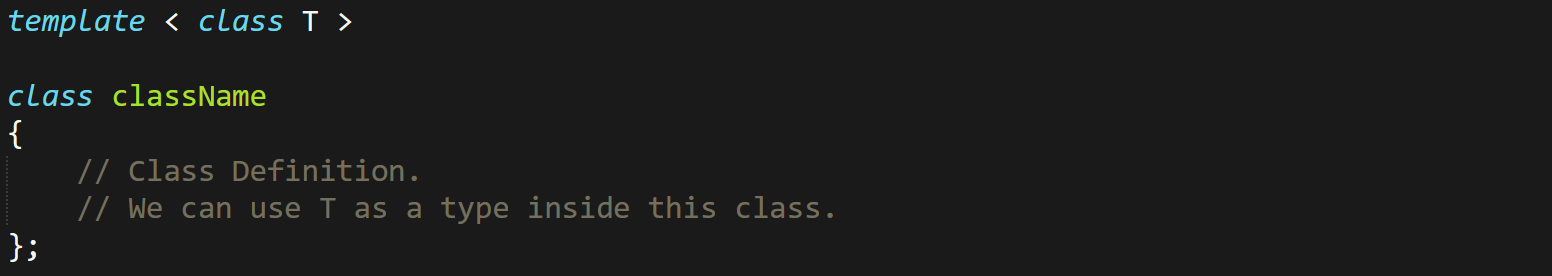


* In this one, compiler decides the data type by itself.
* ***auto*** keyword can’t be used.
* **Same template** data type (**T** here) can be used in multiple generics.
* Template **overloading** is possible.

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



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



**Multithreading**

* 2 types of multi-tasking: **Process based** & **thread based**.
* **<pthread.h>** for **C/C++**
* **<thread>** for **C++** only.

Steps for creating a thread:-

* **Step 1:** Create a function of **void pointer** type:



* + There must be **atleast one *void\** type** **argument** in this function.
  + The **NULL** in **pthread\_exit()** is its exit status.
* **Step 2:** Create thread ID storage (in **main()** function):



**OR (for array of threads)**



* **Step 3:** Creating thread:



* **Step 4:** Add join function:



* + This function **waits** for thread to finish before main function can end.

Arguments of pthread\_create():-

* **&thread** – Address of the thread ID
* **NULL** – Attributes for thread
* **func** – Function we want to call
* **NULL** – The void pointer argument discussed earlier

Storing thread data in structures:-



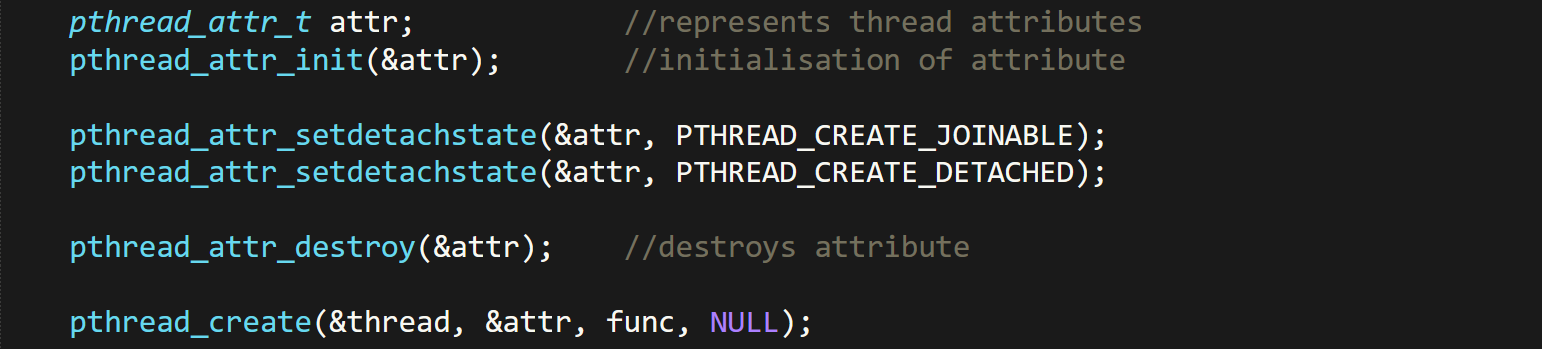
For delaying & sleep:-

* Use **<unistd.h>**.

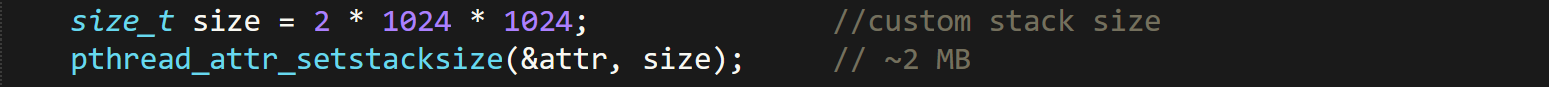


Setting attributes for threads:-

* **Threads can be made joinable or detachable:**

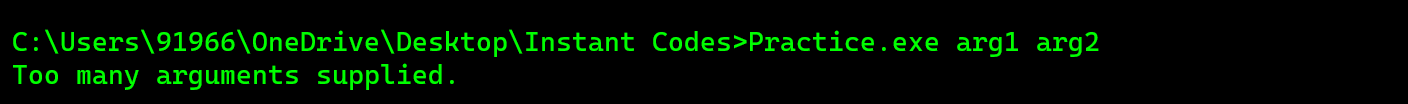


* **For allocating stack memory:**

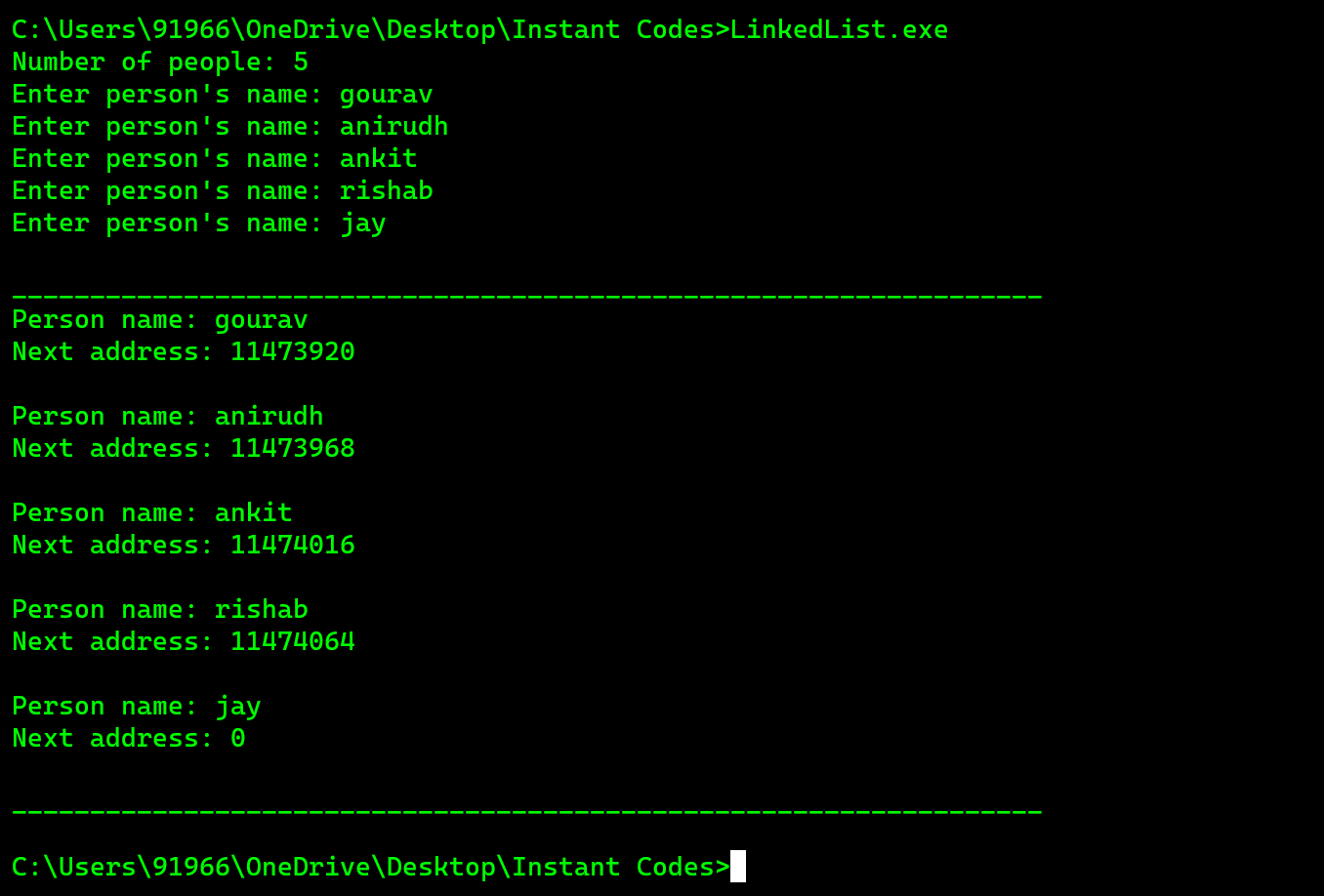


**Command Line Arguments in C/C++**

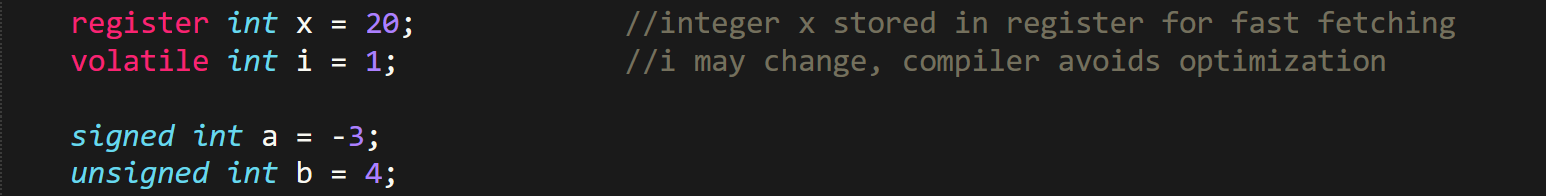
* ***argc*** is by default ***1***, which is the **file name** itself.
* ***argv[1]*** is the **filename**.
* **Code written in command prompt (example):**



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



**Some Newly Observed Keywords**



**C/C++ Libraries**

Components expected in a library:-

* **Header files**
* **Compiled binaries:** Dynamic link libraries (***.dll***) in Windows.
* **Library documentation**
* **Sample codes/ examples**
* **License information:** License about modifying & redistributing open source files.
* **Build files (kind of manuals)**

Some facts about .dll files:-

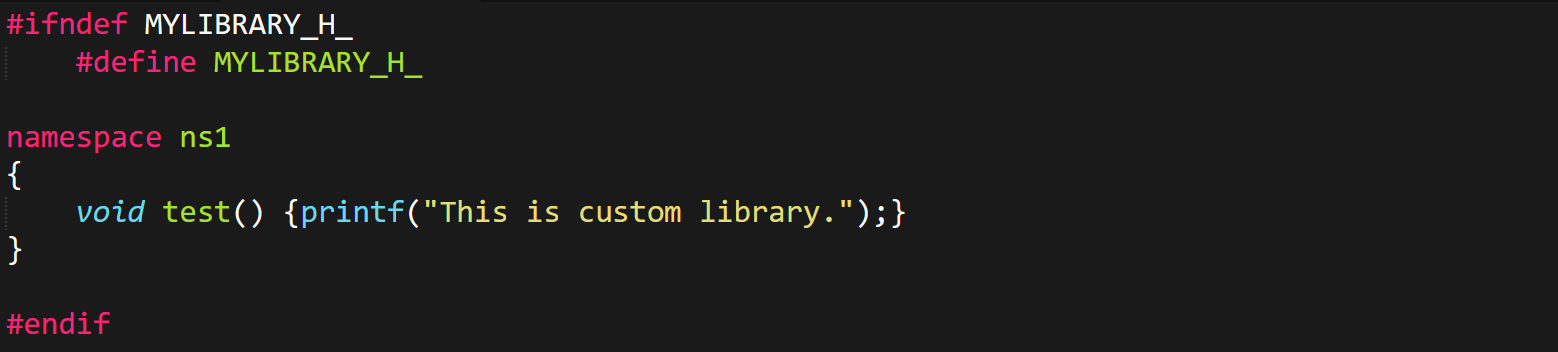
* Are also known as **shared libraries**.
* Contains precompiled **binary files**.
* These files are sharable among users.
* Multiple ***.dll*** files contains separate parts of code, so easy to troubleshoot.
* Library (***.lib***) files help in importing ***.dll*** codes to program.
* **Dependency walker:** A tool used for troubleshooting incompatibleness of ***.dll*** files.
* Extension in Mac & Linux are .daylib & .so respectively.
* Program importing it access its memory unlike **static** **libraries** which **reserve** separate memory spaces for its function to store.

Steps when integrating a library:-

* **Step 1:** **Get** the files of desired library.
* **Step 2:** **Identify** what kind of files it contains.
* **Step 3:** **Organize** your project structure by keeping headers, binaries, source files & third party files etc in different directories.
* **Step 4:** **Add** directory containing library’s header files to include path.
* **Step 5:** **Link** compiled library binaries, i.e. either static or dynamic.
* **Step 6:** **Compile and build** your program.
* **Step 7:** **Make sure** the library’s dynamic files are in the runtime environment.
* **Step 8:** **Include** the library in your code.
* **Step 9:** For larger projects, **integrate** library to your build system.
* **Step 10:** **Implement** proper error handlings for library initialization.
* **Step 11:** Thoroughly **read** libraries documentation for additional steps if required.

**Creating Static Library**

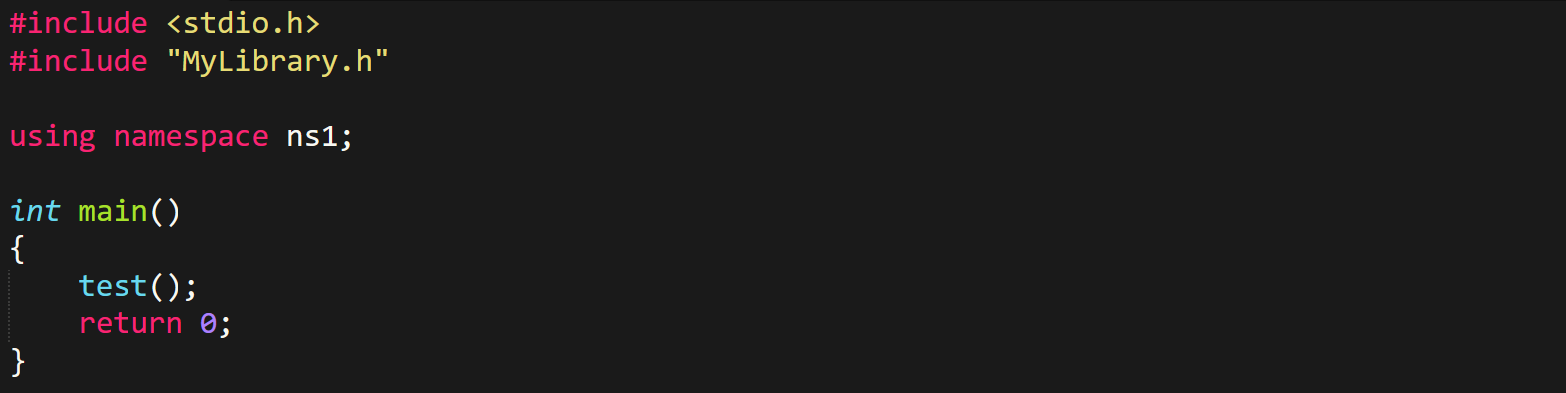
* **Step 1:** Make a file with extension of **.h**/**.hpp**:



* **Step 2:** **Compile** & **link** the header file to the **.lib** or **.a** (**archive**):

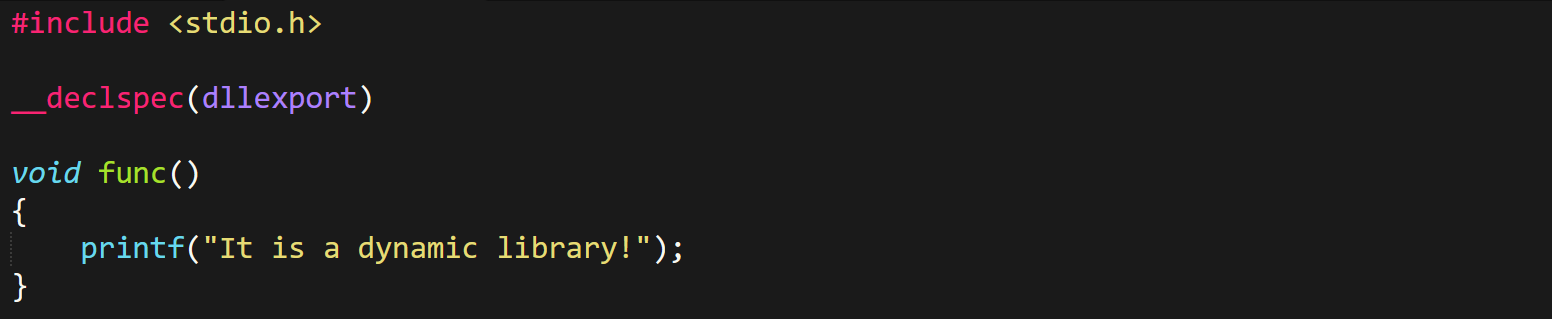


* **Step 3:** Importing & running the library:

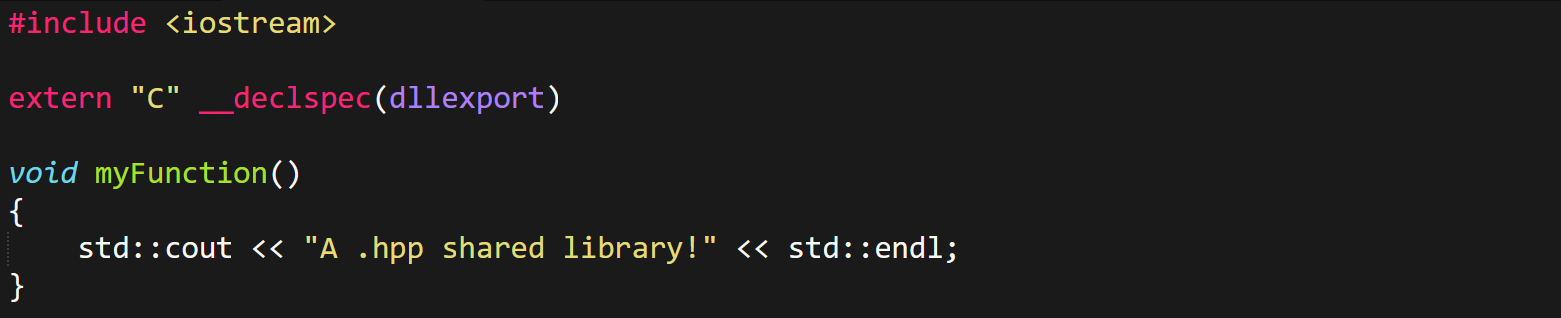


**Creating Dynamic/Shared Library**

* **Step 1:** Write library with **.h** or **.hpp** extension:
  + For **C** (**.h**):-



* + For **C++** (**.hpp**):-



* **Step 2:** **Compile** & **link** the header file to the **.dll** file:



* **Step 3:** ***\*Import and use the library just as we did in case of static libraries.\****