<https://youtu.be/uQB7jNWnZaI>

Basics::

**Line 1:** #include <iostream> is a **header file library** that lets us work with input and output objects, such as cout (used in line 5). Header files add functionality to C++ programs.

**Line 2:** using namespace std means that we can use names for objects and variables from the standard library.

**Header Files:**The files that tell the compiler how to call some functionality (without knowing how the functionality actually works) are called header files. They contain the function prototypes.

We use ***#include*** to use these header files in programs. These files end with **.h** extension.

---------------------------------------------------------------------------------------------------------

**Library:**[Library](https://www.geeksforgeeks.org/algorithms-library-c-stl/) is the place where the actual functionality is implemented i.e. they contain function body. Libraries have mainly two categories :

* Static
* Shared or Dynamic

Static:: . These libraries are specifically used at ***compile time*** which means the library should be present in correct location when user wants to compile his/her C or C++ program. In windows they end with **.lib** extension and with **.a** for MacOS.

**Shared or Dynamic:**These libraries are only required at ***run-time*** i.e, user can compile his/her code without using these libraries. In short these libraries are linked against at compile time to resolve undefined references and then its distributed to the application so that the application can load it at run time. For example, when we open our game folders we can find many **.dll**(dynamic link libraries) files. As these libraries can be shared by multiple programs, they are also called as shared libraries.These files end with **.dll** or **.lib** extensions. In windows they end with .dll extension.

<https://www.differencebetween.com/difference-between-header-file-and-vs-library-file/#:~:text=Summary%20%E2%80%93%20Header%20File%20vs%20Library%20File&text=The%20difference%20between%20a%20header%20file%20and%20library%20file%20is,functions%20in%20the%20header%20file>.

Diff bet header and libraray.

In C++, a **namespace** is a collection of related names or identifiers (functions, class, variables) which helps to separate these identifiers from similar identifiers in other namespaces or the global namespace.

In C++, a namespace is a way to group related variables, functions, and classes under a single name.

namespace mynamespace {

int x = 42;

void foo() {

// ...

}

class MyClass {

// ...

};

}

#include <iostream>

int main() {

std::cout << mynamespace::x << std::endl;

mynamespace::foo();

mynamespace::MyClass obj;

return 0;

}

The identifiers of the C++ standard library are defined in a namespace called std.

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

* auto
* register
* static
* extern
* mutable

In C++, the **mutable** storage class is used as a specifier for a class member variable within a class. When you declare a class member variable as **mutable**, it indicates that the variable can be modified even within a **const** member function.

This is particularly useful when you have a member variable that logically belongs to the object and should be modifiable for caching, synchronization, or any other reason, even when the object itself is considered constant.

<https://www.geeksforgeeks.org/c-mutable-keyword/>

// C++ program to illustrate the use of mutalbe storage

// class specifiers

#include <iostream>

using std::cout;

class Test {

public:

int x;

// defining mutable variable y

// now this can be modified

mutable int y;

Test()

{

x = 4;

y = 10;

}

};

int main()

{

// t1 is set to constant

const Test t1;

// trying to change the value

t1.y = 20;

cout << t1.y;

// Uncommenting below lines

// will throw error

// t1.x = 8;

// cout << t1.x;

return 0;

}

## 6. thread\_local Storage Class

The thread\_local Storage Class is the new storage class that was added in C++11. We can use the **thread\_local** storage class specifier to define the object as thread\_local. The thread\_local variable can be combined with other storage specifiers like static or extern and the properties of the thread\_local object changes accordingly.

### Properties of thread\_local Storage Class

* **Memory Location:** RAM
* **Lifetime:**Till the end of its thread

// C++ program to illustrate the use of thread\_local storage

// sprecifier

#include <iostream>

#include <thread>

using namespace std;

// defining thread local variable

thread\_local int var = 10;

// driver code

int main()

{

// thread 1

thread th1([]() {

cout << "Thread 1 var Value: " << (var += 18) << '\n';

});

// thread 2

thread th2([]() {

cout << "Thread 2 var Value: " << (var += 7) << '\n';

});

// thread 3

thread th3([]() {

cout << "Thread 3 var Value: " << (var += 13) << '\n';

});

th1.join();

th2.join();

th3.join();

return 0;

}

--------------------------------------------------------------------------------------------------------------------

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

References vs Pointers

References are often confused with pointers but three major differences between references and pointers are −

* You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
* Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
* A reference must be initialized when it is created. Pointers can be initialized at any time.
* #include <iostream>
* using namespace std;
* int main () {
* // declare simple variables
* int i;
* double d;
* // declare reference variables
* int& r = i;
* double& s = d;
* i = 5;
* cout << "Value of i : " << i << endl;
* cout << "Value of i reference : " << r << endl;
* d = 11.7;
* cout << "Value of d : " << d << endl;
* cout << "Value of d reference : " << s << endl;
* return 0;
* }
* When the above code is compiled together and executed, it produces the following result −
* Value of i : 5
* Value of i reference : 5
* Value of d : 11.7
* Value of d reference : 11.7

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

* I/O Library Header Files
* There are following header files important to C++ programs −

|  |  |
| --- | --- |
| **Sr.No** | **Header File & Function and Description** |
| 1 | **<iostream>**  This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively. |
| 2 | **<iomanip>**  This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**. |
| 3 | **<fstream>**  This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter. |

## The Standard Output Stream (cout)

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs

## The Standard Input Stream (cin)

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs

The Standard Error Stream (cerr)

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.

[Live Demo](http://tpcg.io/3xp5lM)

#include <iostream>

using namespace std;

int main() {

char str[] = "Unable to read....";

cerr << "Error message : " << str << endl;

}

When the above code is compiled and executed, it produces the following result −

Error message : Unable to read....

The Standard Log Stream (clog)

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

[Live Demo](http://tpcg.io/Dewejb)

#include <iostream>

using namespace std;

int main() {

char str[] = "Unable to read....";

clog << "Error message : " << str << endl;

}

When the above code is compiled and executed, it produces the following result −

Error message : Unable to read....

Difff between pointer and refrence

A reference variable is another name for an already existing variable. It is mainly used in '**pass by reference**' where the reference variable is passed as a parameter to the function and the function to which this variable is passed works on the original copy of the variable.

**Let's understand through a simple example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **void** func(**int** &);
4. **int** main()
5. {
6. **int** a=10;
7. std::cout <<"Value of 'a' is :" <<a<< std::endl;
8. func(a);
9. std::cout << "Now value of 'a' is :" <<a<< std::endl;
10. **return** 0;
11. }
12. **void** func(**int** &m)
13. {
14. m=8;
15. }

o/p = value of a=10

value of a =8

# **malloc() vs new in C++**

Both the **malloc()** and new in C++ are used for the same purpose. They are used for allocating memory at the runtime. But, malloc() and new have different syntax. The main difference between the malloc() and new is that the new is an operator while malloc() is a standard library function that is predefined in a **stdlib** header file.

### **What is new?**

The new is a memory allocation operator, which is used to allocate the memory at the runtime. The memory initialized by the new operator is allocated in a heap. It returns the starting address of the memory, which gets assigned to the variable. The functionality of the new [operator in C++](https://www.javatpoint.com/cpp-operators) is similar to the malloc() function, which was used in the [C programming language](https://www.javatpoint.com/c-programming-language-tutorial). [C++](https://www.javatpoint.com/cpp-tutorial) is compatible with the malloc() function also, but the new operator is mostly used because of its advantages.

**Syntax of new operator**

1. type variable = **new** type(parameter\_list);
2. #include <iostream>
3. **using** **namespace** std;
4. **int** main()
5. {
6. **int** \*ptr;  // integer pointer variable declaration
7. ptr=**new** **int**; // allocating memory to the pointer variable ptr.
8. std::cout << "Enter the number : " << std::endl;
9. std::cin >>\*ptr;
10. std::cout << "Entered number is " <<\*ptr<< std::endl;
11. **return** 0;

o/p enter a number:12

the entered number is:12

diff between malloc and new

* The new operator constructs an object, i.e., it calls the constructor to initialize an object while **malloc()** function does not call the constructor. The new operator invokes the constructor, and the delete operator invokes the destructor to destroy the object. This is the biggest difference between the malloc() and new.
* The new is an operator, while malloc() is a predefined function in the stdlib header file.
* The operator new can be overloaded while the malloc() function cannot be overloaded.
* If the sufficient memory is not available in a heap, then the new operator will throw an exception while the malloc() function returns a NULL pointer.
* In the new operator, we need to specify the number of objects to be allocated while in malloc() function, we need to specify the number of bytes to be allocated.
* In the case of a new operator, we have to use the delete operator to deallocate the memory. But in the case of malloc() function, we have to use the free() function to deallocate the memory.

**Syntax of new operator**

1. type reference\_variable = **new** type name;

**where,**

**type:** It defines the data type of the reference variable.

**reference\_variable:** It is the name of the pointer variable.

**new:** It is an operator used for allocating the memory.

### **free() function**

The free() function is used in C++ to de-allocate the memory dynamically. It is basically a library function used in C++, and it is defined in **stdlib.h** header file. This library function is used when the pointers either pointing to the memory allocated using malloc() function or Null pointer.

### **Syntax of free() function**

Suppose we have declared a pointer 'ptr', and now, we want to de-allocate its memory:

1. free(ptr);

### **Delete operator**

It is an operator used in [C++ programming language](https://www.javatpoint.com/cpp-tutorial), and it is used to de-allocate the memory dynamically. This operator is mainly used either for those pointers which are allocated using a new operator or NULL pointer.

### **Syntax**

1. **delete** pointer\_name

For example, if we allocate the memory to the pointer using the new operator, and now we want to delete it. To delete the pointer, we use the following statement:

1. **delete** p;

To delete the array, we use the statement as given below:

1. **delete** [] p;

**Some important points related to delete operator are:**

* It is either used to delete the array or non-array objects which are allocated by using the new keyword.
* To delete the array or non-array object, we use delete[] and delete operator, respectively.
* The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap
* It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.

### **Differences between delete and free()**

The following are the differences between delete and free() in C++ are:

* The delete is an operator that de-allocates the memory dynamically while the free() is a function that destroys the memory at the runtime.
* The delete operator is used to delete the pointer, which is either allocated using new operator or a NULL pointer, whereas the free() function is used to delete the pointer that is either allocated using malloc(), calloc() or realloc() function or NULL pointer.
* When the delete operator destroys the allocated memory, then it calls the destructor of the class in C++, whereas the free() function does not call the destructor; it only frees the memory from the heap.
* The delete() operator is faster than the free() function.

In C++, the `new` operator is used to allocate memory for an object on the heap and then construct an object in that memory. It is typically used with a corresponding `delete` operator to deallocate the memory and destroy the object when it's no longer needed.

Int \*ptr = new int;

\*ptr = 32;

After this line executes, **ptr** contains the memory address where a single integer is stored on the heap. You can use this pointer to access and manipulate the integer value stored at that memory location.

Int \*p;

\*p = 43

The pointer **p** is declared but not initialized before you try to dereference it and assign a value. Dereferencing an uninitialized pointer leads to undefined behavior, and your program may crash or exhibit unexpected beha

Here's how the `new` operator works internally in C++:

1. Memory Allocation:

When you use `new` to create an object, it first requests memory from the heap using functions like `malloc` or `operator new`. The size of the memory block allocated is determined by the object's size.

2. Object Construction:

After the memory is allocated, the constructor for the object is called to initialize the object's data members. If you're creating a built-in type like an integer, this step isn't relevant, but for user-defined classes, the constructor is responsible for initializing the object.

3. Returning a Pointer:

The `new` operator returns a pointer to the memory allocated, pointing to the newly created object. This pointer can be stored in a variable or used directly.

Here's an example of how you might use the `new` operator to create an object of a user-defined class:

class MyClass {

public:

MyClass(int value) : data(value) {

cout << "Constructor called with value: " << value << endl;

}

void print() {

cout << "Data: " << data << endl;

}

private:

int data;

};

int main() {

MyClass\* obj = new MyClass(42); // Allocates memory and calls the constructor

obj->print(); // Calls a member function on the object

delete obj; // Deallocates memory and calls the destructor (not explicitly shown here)

return 0;

}

```It's important to note that when you allocate memory using `new`, you are responsible for releasing that memory when you're done with it. You should use the `delete` operator to free the memory and call the object's destructor. Failing to do so can lead to memory leaks. In modern C++, using smart pointers like `std::unique\_ptr` or `std::shared\_ptr` is recommended to manage dynamic memory more safely and automatically.

In C++, the `delete` operator knows how much memory to deallocate because it uses information stored internally by the memory allocation system. When you allocate memory using the `new` operator, additional information about the allocated block's size is typically stored along with the memory. This information allows `delete` to determine the correct amount of memory to release.

Here's how it works in more detail:

1. Allocation Information: When you use `new` to allocate memory, the memory allocation system (e.g., `malloc` or `operator new`) not only allocates the requested memory but also reserves additional memory to store information about the size of the allocated block. This information typically includes the size of the allocated memory block in bytes.

2. Deallocation with `delete`: When you use the `delete` operator to deallocate memory, it looks at the additional information stored with the memory block to determine the size of the block. This information allows `delete` to release the exact amount of memory that was allocated.

Here's an example:

```cpp

int\* ptr = new int[5]; // Allocates memory for an array of 5 integers

delete[] ptr; // Deallocates memory based on the information stored

```

In this example, when you allocate memory for an array of 5 integers using `new`, information about the size of the allocated memory block is stored. When you later use `delete[]` to deallocate the memory, the system looks at this size information to correctly release the memory.

It's essential to use `delete` or `delete[]` for dynamic memory allocated with `new` or `new[]`, respectively. Failing to do so, or using the wrong form (e.g., `delete` for an array allocated with `new[]` or vice versa), can lead to undefined behavior, memory leaks, or other issues. In modern C++, using smart pointers like `std::unique\_ptr` or `std::shared\_ptr` is recommended to manage dynamic memory allocation and deallocation more safely and automatically.