C++

Code Compilation Syntax: g++ *filename.cpp* -o *any-name*

1. using namespace std:

- This is used to import the entirety of the std namespace into the current namespace of the program.
- The statement using namespace std is generally considered a bad practice. When we import a namespace, we are essentially pulling all type definitions into the current scope.
- The alternative to this statement is to specify the namespace to which the identifier belongs using the scope operator(::) each time we declare a type.
- The std namespace is huge. It has hundreds of predefined identifiers.
- Let us say we wish to use the cout from the std namespace. Now at a later stage of development, we wish to use another version of cout that is custom implemented in some library called "foo" then it will give conflict or undesired output.

2. "void main()" or "main()" ?

- A conforming implementation may provide more versions of main(), but they must all have return type int.
- The int returned by main() is a way for a program to return a value to "the system" that invokes it.
- On systems that don't provide such a facility the return value is ignored, but that doesn't make "void main()" legal C++ or legal C.
- Even if your compiler accepts "void main()" avoid it, or risk being considered ignorant by C and C++ programmers.
- In C++, main() need not contain an explicit return statement. In that case, the value returned is 0, meaning successful execution.

3. IO Stream:

i. Un-buffered standard error stream (cerr):

- The C++ cerr is the standard error stream which is used to output the errors.
- This is also an instance of the ostream class.
- As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately.
- It does not have any buffer to store the error message and display later.
- The main difference between cerr and cout comes when we would like to redirect output using "cout" that gets redirected to file but if we use "cerr" the error

doesn't get stored in file. (This is what un-buffered means. It can't store the message.)

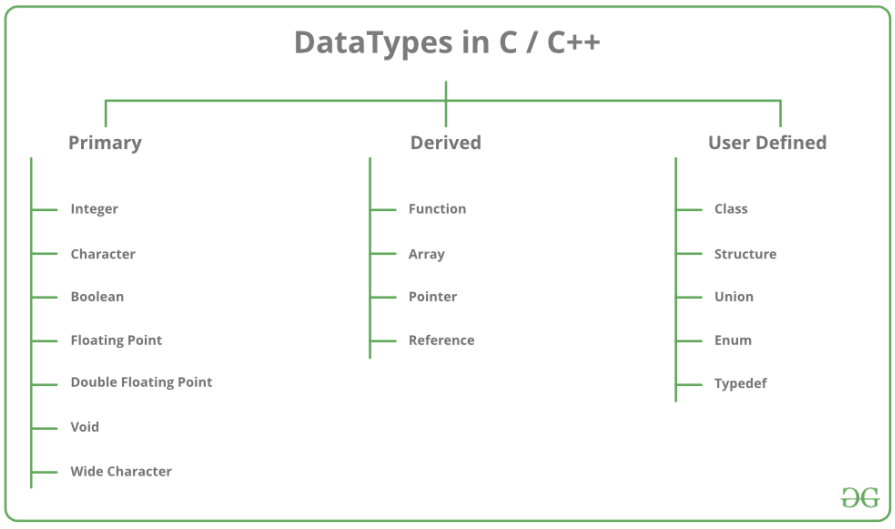
ii. Buffered standard error stream (clog):

- This is also an instance of ostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled, or the buffer is not explicitly flushed (using flush()).
- The error message will be displayed on the screen too.

4. endl vs \n:

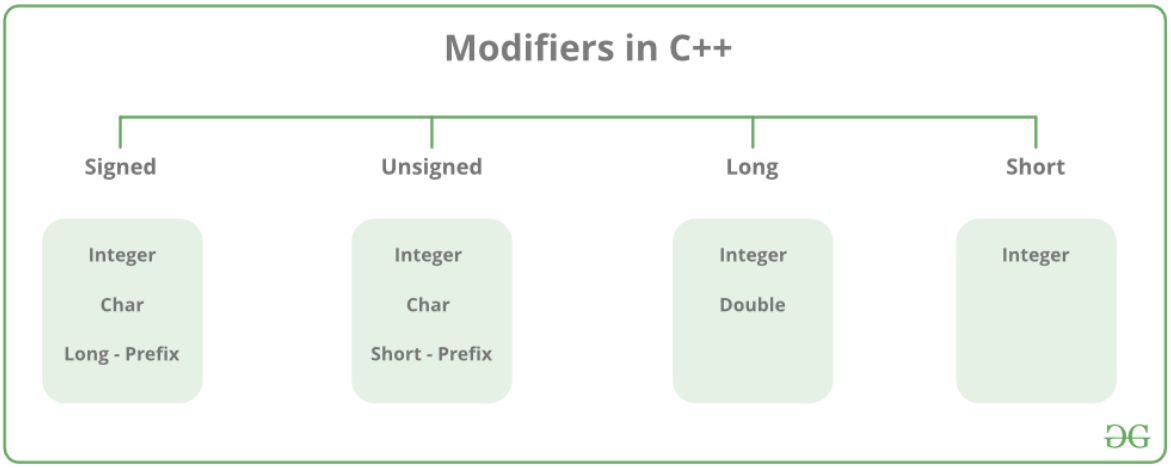
- **cout << endl :** Inserts a new line and flushes the stream.
- **cout << "\n" :** Only inserts a new line.
- Therefore, cout << endl; can be said equivalent to cout << '\n' << flush;
- So cout << "\n" seems performance wise better than cout << endl; unless flushing of stream is required.
- Some other differences between endl and \n are:
  - endl is manipulator while \n is character.
  - endl doesn't occupy any memory whereas \n is character, so it occupies 1 byte memory.
  - \n being a character can be stored in a string (will still convey its specific meaning of line break) while endl is a keyword and would not specify any meaning when stored in a string.
  - We can use \n both in C and C++ but, endl is only supported by C++ and not the C language

5. Data Types:



6. Datatype Modifiers:

- As the name implies, datatype modifiers are used with the built-in data types to modify the length of data that a particular data type can hold.



Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 4,294,967,295
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

- **Wide Character (wchar_t):** Wide character data type is also a character data type, but this data type has size greater than the normal 8-bit datatype. It is generally 2 or 4 bytes long.

7. size_t data type:

- size_t is an unsigned integral data type which is defined in various header files such as: <stddef.h>, <stdio.h>, <stdlib.h>, <string.h>, <time.h>, <wchar.h>.
- It's a type which is used to represent the size of objects in bytes and is therefore used as the return type by the sizeof operator.
- It is guaranteed to be big enough to contain the size of the biggest object the host system can handle.
- Basically, the maximum permissible size is dependent on the compiler, if the compiler is 32 bit then it is simply a typedef for unsigned int but if the compiler is 64 bit then it would be a typedef for unsigned long long.
- The size_t data type is never negative. Therefore, many C library functions like *malloc, memcpy and strlen* declare their arguments and return type as size_t.

```
// Declaration of various standard library functions.

// 'n' refers to max blocks that can be allocated which is to be non-negative.
void *malloc(size_t n);

// While copying 'n' bytes from 's2' to 's1' n must be non-negative integer.
void *memcpy(void *s1, void const *s2, size_t n);

// strlen() uses size_t because the length of any string will always be at least 0.
size_t strlen(char const *s);
```

- size_t or any unsigned type might be seen used as loop variable as loop variables are typically greater than or equal to 0.
- Note: When we use a size_t object, we must make sure that in all the contexts it is used, including arithmetic, we want only non-negative values.

8. Functions that are executed before and after main():

- With GCC family of C compilers, we can mark some functions to execute before and after main().
- So, some startup code can be executed before main() starts, and some cleanup code can be executed after main() ends.
- For example, in the following program, myStartupFun() is called before main() and myCleanupFun() is called after main().

```
#include<stdio.h>

/* Apply constructor attribute to myStartupFun() so that it is executed before main() */
void myStartupFun (void) __attribute__ ((constructor));

/* Apply destructor attribute to myCleanupFun() so that it is executed after main() */
```

```
void myCleanupFun (void) __attribute__ ((destructor));

/* Implementation of myStartupFun */
void myStartupFun (void)
{
   printf ("startup code before main()\n");
}

/* Implementation of myCleanupFun */
void myCleanupFun (void)
{
   printf ("cleanup code after main()\n");
}

int main (void)
{
   printf ("hello\n");
   return 0;
}
```

```
startup code before main()
hello
cleanup code after main()
```

9. Problem with scanf() when there is fgets()/gets()/scanf() after it:

- Consider below simple program in C. The program reads an integer using scanf(), then reads a string using fgets().
- ***Code:***

```
#include<stdio.h>

int main()
{
  int x;
  char str[100];
  scanf("%d", &x);
  fgets(str, 100, stdin);
  printf("x = %d, str = %s", x, str);
  return 0;
}
```

- ***Input:***

```
10
test
```

- ***Output:***

```
x = 10, str =
```

- The problem with above code is scanf() reads an integer and leaves a newline character in buffer. So fgets() only reads newline and the string "test" is ignored by the program.
- The similar problem occurs when scanf() is used in a loop. This happens because every scanf() leaves a newline character in buffer that is read by next scanf.
- **How to solve above problem ?**
  - We can make scanf() to read a new line by using an extra "\n", i.e., **scanf("%d\n", &x)**. In fact, **scanf("%d ", &x)** also works (Note extra space).
  - We can add a getchar() after scanf() to read an extra newline.

10. How to return multiple values from a function in C or C++ ?

- Unfortunately, C and C++ do not allow this directly. But fortunately, with a little bit of clever programming, we can easily achieve this.
- **Below are the methods to return multiple values from a function in C:**
  - By using Pointers.
  - By using Structures.
  - By using Arrays
  - By using Reference (only C++)
  - By using Class (only C++)

11. exit():

- *Syntax:*

```
void exit ( int status );
```

- exit() terminates the process normally.
- status: Status value returned to the parent process. Generally, a status value of 0 or EXIT_SUCCESS indicates success, and any other value or the constant EXIT_FAILURE is used to indicate an error.
- exit() performs following operations.
  - Flushes unwritten buffered data.
  - Closes all open files.
  - Removes temporary files.
  - Returns an integer exit status to the operating system.
- The C standard atexit() function can be used to customize exit() to perform additional actions at program termination.
- When exit() is called, any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, init, and the process parent is sent a SIGCHLD signal.
- The mystery behind exit() is that it takes only integer args in the range 0 - 255. Out of range exit values can result in unexpected exit codes. An exit value greater than 255 returns an exit code modulo 256.
- For example, exit 9999 gives an exit code of 15 i.e. (9999 % 256 = 15).

- Explanation: It is effect of 8-bit integer overflow. After 255 (all 8 bits set) comes 0. So the output is “exit code modulo 256”.

12. abort():

- *Syntax:*

```
void abort ( void ):
```

- Unlike exit() function, abort() may not close files that are open.
- It may also not delete temporary files and may not flush stream buffer.
- Also, it does not call functions registered with atexit().
- This function actually terminates the process by raising a SIGABRT signal.
- If we want to make sure that data is written to files and/or buffers are flushed, then we should either use exit() or include a signal handler for SIGABRT.

13. assert():

- *Syntax:*

```
void assert( int expression );
```

- If expression evaluates to 0 (false), then the expression, source code filename, and line number are sent to the standard error, and then abort() function is called.
- *Common error outputting is in the form:*
  Assertion failed: expression, file filename, line line-number
- *Example:*

```
Assertion failed: x==7, file test.cpp, line 13
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

- Assertions are statements used to test assumptions made by programmers. For example, we may use assertion to check if the pointer returned by malloc() is NULL or not.
- If the identifier NDEBUG (“no debug”) is defined with **#define NDEBUG** then the macro assert does nothing.

14. exit() vs _Exit():

- In C, exit() terminates the calling process without executing the rest code which is after the exit() function.
- Now the question is that if we have exit() function then why C11 standard introduced _Exit()? Actually exit() function performs some cleaning before termination of the program like connection termination, buffer flushes etc. The _Exit() function in C/C++ gives normal termination of a program without performing any cleanup tasks.
- For example, it does not execute functions registered with atexit.

15. return() vs exit() in main:

- When *exit(0)* is used to exit from program, destructors for locally scoped non-static objects are not called. But destructors are called if *return 0* is used.
- Note that static objects will be cleaned up even if we call exit().

16. Assertion vs Normal Error Handling:

- Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state a code expects before it starts running or state after it finishes running.
- Unlike normal error handling, assertions are generally disabled at run-time. Therefore, it is not a good idea to write statements in assert() that can cause side effects. For example, writing something like assert(x = 5) is not a good idea as x is changed and this change won't happen when assertions are disabled.

17. Array Decay

- **What is Array Decay?**
  - The loss of type and dimensions of an array is known as decay of an array.
  - This generally occurs when we pass the array into function by value or pointer. Because it sends first address to the array which is a pointer, hence the size of array is not the original one, but the one occupied by the pointer in the memory.

Input:

```
#include<iostream>
using namespace std;

// Function to show Array decay by passing array by value
void aDecay(int *p)
{
   // Printing size of pointer
   cout << "Size of array by passing by value: " << sizeof(p) << endl;
}

// Function to show that array decay happens even if we use pointer
void pDecay(int (*p)[7])
{
   // Printing size of array
   cout << "Size of array by passing by pointer: " << sizeof(p) << endl;
}

int main()
{
   int a[7] = {1, 2, 3, 4, 5, 6, 7,};
```

```
    // Printing original size of array
    cout << "Actual size of array is: " << sizeof(a) <<endl;

    // Passing a pointer to array as an array is always passed by reference
    aDecay(a);

    // Calling function by pointer
    pDecay(&a);

    return 0;
}
```

Output:

```
Actual size of array is: 28
Size of array by passing by value: 8
Size of array by passing by pointer: 8
```

Explanation: In the above code, the actual array has 7 int elements and hence has 28 size. But by calling by value and pointer, array decays into pointer and prints the size of 1 pointer i.e., 8 (4 in 32 bit).

- **How to prevent Array Decay?**
  - A typical solution to handle decay is to pass size of array also as a parameter and not use sizeof() on array parameters.
  - Another way to prevent array decay is to send the array into functions by reference. This prevents conversion of array into a pointer, hence prevents the decay.

Input:

```
#include<iostream>
using namespace std;

// Function that prevents Array decay by passing array by reference
void fun(int (&p)[7])
{
    // Printing size of array
    cout << "Size of array by passing by reference: " << sizeof(p) << endl;
}

int main()
{
    int a[7] = {1, 2, 3, 4, 5, 6, 7,};

    // Printing original size of array
    cout << "Actual size of array is: " << sizeof(a) << endl;

    // Calling function by reference
    fun(a);
```

```
    return 0;
}
```

Output:

```
Actual size of array is: 28
Size of array by passing by reference: 28
```

Explanation:
In the above code, passing array by reference solves the problem of decay of array. Sizes in both cases is 28.

18. nullptr:

- NULL is typically defined as (void *)0 and conversion of NULL to integral types is allowed. So, some time calling function with NULL argument becomes ambiguous.
- nullptr is a keyword that can be used at all places where NULL is expected. Like NULL, nullptr is implicitly convertible and comparable to any pointer type. Unlike NULL, it is not implicitly convertible or comparable to integral types.

19. NaN

- NaN, acronym for "Not a Number" is an exception which usually occurs in the cases when an expression results in a number that can't be represented.
- **Example:** Square root of negative numbers.

Input:

```
#include<iostream>
#include<cmath>      // for sqrt()
using namespace std;

int main()
{
    float a = 2, b = -2;

    // Prints the number (1.41421)
    cout << sqrt(a) << endl;

    // Prints "nan" exception as sqrt(-2) is complex number
    cout << sqrt(b) << endl;
}
```

Output:

```
1.41421
-nan
```

- We can check for NaN is by using “isnan()” function, this function returns true if a number is complex else it returns false.

20. CHAR_BIT:

- It is the number of bits in char. These days, almost all architectures use 8 bits per byte (But it is not the case always, some older machines used to have 7-bit byte).
- It is basically pre-defined macro.