General features:

## 1. constexpr

->Feature added in c++ 11.

-> Improves performance by doing computations at compiler time rather than run time
Spend time at compile time and save at runtime

-> Specifies that value of an object or a function can be evaluated at compile time and expression can be used in other constant expression

-> Example:

```
constexpr int product(int x, int y)
{
    return (x * y);
}

int main()
{
    const int x = product(10, 20);
    cout << x;
    return 0;
}
```

In above code, product() is evaluated at compile time.

## 2. auto type

-> Comes under Type inference which refers to automatic deduction of data type of of expression in programming language.

-> auto keyword specifies that type of variable that is being declared automatically deducted from initializer.

-> In functions, if return type is auto then it will be evaluated by return type expression at runtime.

-> Example 1:

```
using namespace std;

int main()
{
    auto x = 4;
    auto y = 3.37;
    auto ptr = &x;
    cout << typeid(x).name() << endl
         << typeid(y).name() << endl
         << typeid(ptr).name() << endl;

    return 0;
}
```

-> Here we can see x is int, y is double and ptr is pointer variable. C++ automatically determines data type of variable based on value assigned to variable.

-> Auto can be used to avoid long initializations when creating iterators for containers.

-> Example 2:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // Create a set of strings
    set<string> st;
    st.insert({ "geeks", "for", "geeks", "org" });

    // 'it' evaluates to iterator to set of string
    // type automatically
    for (auto it = st.begin(); it != st.end(); it++)
        cout << *it << " ";

    return 0;
}
```

## 3. decltype

-> Inspects the declared type of an entity or type of an expression.

-> Allows us to extract the type from variable so decltype is sort of an operator that evaluates the type of passed expression.

-> Example:

```
using namespace std;

int fun1() { return 10; }
char fun2() { return 'g'; }

int main()
{
    // Data type of x is same as return type of fun1()
    // and type of y is same as return type of fun2()
    decltype(fun1()) x;
    decltype(fun2()) y;

    cout << typeid(x).name() << endl;
    cout << typeid(y).name() << endl;

    return 0;
}
```

o/p: i c

We can see output is integer and character which are datatypes of fun1 and fun2 respectively

## 4. range based for loops

-> Executes for loop over a range. More readable than normal for loop.

-> Syntax: *for (range_declaration : range_expression)*
*loop_statement*

-> Example:for (auto i : v)
std::cout << i << ' ';

-> **Parameters:**

*range_declaration:* Declaration of named variable whose type is type of element of the sequence represented by range_expression, or a reference to that type.
Often uses auto specifier for automatic type deduction.

*range_expression:* any expression that represents a suitable sequence or a braced init list

*loop_statement :* any statement, typically a compound statement, which is the body of the loop.

-> Example:

```
int main()
{
    // Iterating over whole array
    std::vector<int> v = {0, 1, 2, 3, 4, 5};
    for (auto i : v)
        std::cout << i << ' ';

    std::cout << '\n';

    // the initializer may be a braced-init-list
    for (int n : {0, 1, 2, 3, 4, 5})
        std::cout << n << ' ';

    std::cout << '\n';

    // Iterating over array
    int a[] = {0, 1, 2, 3, 4, 5};
    for (int n : a)
        std::cout << n << ' ';

    std::cout << '\n';

    // Just running a loop for every array
    // element
    for (int n : a)
        std::cout << "In loop" << ' ';

    std::cout << '\n';
```

```
    // Printing string characters
    std::string str = "Geeks";
    for (char c : str)
        std::cout << c << ' ';

    std::cout << '\n';

    // Printing keys and values of a map
    std::map <int, int> MAP({{1, 1}, {2, 2}, {3, 3}});
    for (auto i : MAP)
        std::cout << '{' << i.first << ", "
                  << i.second << "}\n";
}
```

## 5. Nullptr

-> NULL is a macro whereas nullptr is a keyword

-> NULL takes numerical value as 0 whereas nullptr takes address to be null.

-> nullptr is prvalue of type null_ptr which is an integer literal that evaulates to zero.

**Issues with NULL**

*Implicit Conversion:*

```
char *str = NULL; // Implicit conversion from void * to char *

int i = NULL;     // OK, but `i` is not pointer type
int i=nullptr //error
int *p = NULL //ok
inpt * p = nullptr
```

*Function calling ambiguity*

```
void func(int) {}

void func(int*){}

void func(bool){}

func(NULL);     // Which one to call?
```

Compilers produces errors.

We can understand nullptr with following example:

```
void func(int x); //1
void func(int *x);  //2
 main()
 {
 func(NULL)// calls func 1 because NULL is integer, which is wrong.
```

}

*Solution to above problem:*

-> To avoid this problem we can use nullptr to avoid ambiguity between 2 functions.

-> func(nullptr) // calls func 2 because nullptr is not an integer and converts implicity to any pointer type

-> Avoids ambiguity between function overloads
more secure, initutive and expressive code, eg: if(ptr==nullptr) instead of if(ptr==0)

## 6. Scoped/Strongly typed enums

-> Some of the drawbacks of enums are:
enums implicitly convert to int
type of enums can not be specified
it is not possible to declare variable name which was defined already in enum

example: enum Color{red=0, green=2, blue}

int red = 5 //Error

-> We can see if red is declared as int variable it throws an error because red is already used in enum. But type of red is enum in one case and int in other case.

*To avoid this problem we use strongly typed enums*

-> Some rules of strongly typed enums are:
Enums can be accessed only in the scope of enumeration
Enums dont implicitly convert to int
Type of enums is by default int. Therefore you can forward enumeration

-> **Syntax of classical enums:**
enum Color;

-> S**yntax of strongly typed enums:**
enum class|struct Color;

```
enum struct NewEnum{
  one= 1,
  ten=10,
  hundred=100,
  thousand= 1000
};
```

Example: To use enum as int, we need to explicitly convert it with static_cast

```
#include <iostream>

enum OldEnum{
```

```
  one= 1,
  ten=10,
  hundred=100,
  thousand= 1000
};

enum struct NewEnum{
  one= 1,
  ten=10,
  hundred=100,
  thousand= 1000
};

int main(){

  std::cout << std::endl;

  std::cout << "C++11= " << 2*thousand + 0*hundred + 1*ten + 1*one << std::endl;
  std::cout << "C++11= " << 2*static_cast<int>(NewEnum::thousand) +
                        0*static_cast<int>(NewEnum::hundred) +
                        1*static_cast<int>(NewEnum::ten) +
                           1*static_cast<int>(NewEnum::one) << std::endl;

}
```

**Output: c++11 = 2011**
**c++11 = 2011**
**In order to calculate or output the enumerators, you have to convert them into integral types.**

## 7. static_assert

-> Way to check if a condition is true when the code is compiled
If it isnt compiled, then compiler should issue error message and then stop compiling process.

-> Prior to c++11 static assert was done using #error directive.
It worked well with simple tasks but fails for complex tasks such as checking the size of data type using the sizeof operator.

***Syntax of static assert:***
*static_assert( constant_expression, string_literal );*
Parameters:
*constant_expression:* An integral constant expression
that can be converted to a Boolean.

*string_literal*: The error message that is displayed
when the constant_expression parameter is false.

Example:

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

template <class T, int Size>
class Vector {
   // Compile time assertion to check if
   // the size of the vector is greater than
   // 3 or not. If any vector is declared whose
   // size is less than 4, the assertion will fail
   static_assert(Size > 3, "Vector size is too small!");

   T m_values[Size];
};

int main()
{
   Vector<int, 4> four; // This will work
   Vector<short, 2> two; // This will fail

   return 0;
}
```

**-> Here if vector size goes below three, compiler will throw error saying static assertion has failed.**

-> static_assert can be used in namespace scope, class scope as well as block scopes. Some examples are given below:

*Namespace Scope:*

```
// CPP program to illustrate
// declaring static_assert in namespace scope
#include <iostream>
static_assert(sizeof(void*) == 8,
"DTAMDL(*LLP64) is not allowed for this module.");
int main()
{
   cout << "Assertion passed.
   The program didn't produce an error";
   return 0;
}
```

Output: Assertion passed. Program didnt produce error

*Class Scope:*

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

template <class T, int Size>
class Vector {
```

```
    // Compile time assertion to check if
    // the size of the vector is greater than
    // 3 or not. If any vector is declared whose
    // size is less than 4, the assertion will fail
    static_assert(Size > 3, "Vector size is too small!");

    T m_values[Size];
};

int main()
{
    Vector<int, 4> four; // This will work
    Vector<short, 2> two; // This will fail

    return 0;
}
```

Output:
error: static assertion failed: Vector size is too small!

*Block Scope:*

```
// CPP program to illustrate
// declaring static_assert in block scope
template <typename T, int N>
void f()
{
    static_assert(N >= 0, "length of array a is negative.");
    T a[N];
}
int main()
{
    // assertion fails here
    // because the length of the array passed
    // is below 0
    f<int, -1>();
    return 0;
}
```

**Output: error: static assertion failed: length of array a is negative.**

*Erroneous static_assert*
Here constant expression is passed in static_assertion needs to be valid expression.

Example:

```
int main()
{
    static_assert(1 / 0, "never shows up!");
    return 0;
}
```

Output: Error non-constant condition for static assertion
  static_assert(1 / 0, "never shows up!");

  Here we pass 1/0, which is not valid constanr expression.

## 8. strict initializers with {}

```
int x(10);//ok
float y(2.3f);//ok
int z(2.3f); //conversion
int w{2.3f};//error
int b{20};//ok

Sample s1(10,20);
//Sample::Sample(int,int);
Sample s2(2.3,5.6};  //conversion
Sample s3{2.3,5.6};  //error
Sample s4{12,18};    //ok
```

## 9.  using keyword for alisaing

->Alias declaration can be used to declare a name to use as a synonym for a previously declared type.

-> This mechanism can also be used to create an alias template which can be used particularly useful for custom allocators

Syntax:
*using identifier = type;*

Remarks: identifier
The name of the alias.

type
The type identifier you are creating an alias for.

-> Example 1:

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

Example 2:

```
// C++11
using fmtfl = std::ios_base::fmtflags;
```

```
// C++03 equivalent:
// typedef std::ios_base::fmtflags fmtfl;

fmtfl fl_orig = std::cout.flags();
fmtfl fl_hex = (fl_orig & ~std::cout.basefield) | std::cout.showbase | std::cout.hex;
// ...
std::cout.flags(fl_hex);
```

-> Here instead of doing std::ios_base::fmtflags everytime, we can use fmtfl.

-> Works with function pointers also.

-> Limitation to typedef is it doesnt work with templates.
So use using keyword
Example 3:

```
template<typename T> using ptr = T*;

// the name 'ptr<T>' is now an alias for pointer to T
ptr<int> ptr_int;
```

## 10. user defined literals (UDL)

-> Lets understand user defined iterals with a example given below:

```
  long double Weight = 2.3; //pounds? kilograms? grams?
```

-> By using user defined literals we can attach units to values which makes code more readable and conversions computations can be done at compile time

```
weight = 2.3kg;
ratio = 2.3kg/1.2lb;
```

-> UDLs are treated as call to literal operator. Only suffix form is supported. Name of the literal operator is operator "" followed by suffix.

-> Example:

```
// C++ code to demonstrate working of user defined
// literals (UDLs)
#include<iostream>
#include<iomanip>
using namespace std;

// user defined literals

// KiloGram
long double operator"" _kg( long double x )
{
    return x*1000;
}
```

```
// Gram
long double operator"" _g( long double x )
{
    return x;
}

// MiliGram
long double operator"" _mg( long double x )
{
    return x / 1000;
}

// Driver code
int main()
{
    long double weight = 3.6_kg;
    cout << weight << endl;
    cout << setprecision(8) << ( weight + 2.3_mg ) << endl;
    cout << ( 32.3_kg / 2.0_g ) << endl;
    cout << ( 32.3_mg *2.0_g ) << endl;
    return 0;
}
```

Output:
3600
3600.0023
16150
0.0646

-> By seeing above example we can observe that _kg,_g,_mg are defined as operators and when it is used in main function like 3.6_kg, 2.3_mg it gives required meaning and computation can be done at compile time.

## 11. Binary literals

-> Let us learn it with an example:
int i=0b0011;
int j = 0B01101001
Binary literal can be declared using b or B.

## 12. Digit seperators

-> C++14 define Simple Quotation Mark ' as a digit separator, in numbers and user-defined literals. This can make it easier for human readers to parse large numbers.

-> Example:

```
long long decn = 1'000'000'000ll;
long long hexn = 0xFFFF'FFFFll;
long long octn = 00'23'00ll;
long long binn = 0b1010'0011ll;
```

## 13. Raw string literals

-> In C++, to escape characters like “\n” we use an extra “\”.

-> From C++ 11, we can use raw strings in which escape characters (like \n \t or \” ) are not processed.

-> The syntax of raw string is that the literal starts with R”( and ends in )”.

-> Example:

```
int main()
 { // A Normal string
 string string1 = "Geeks.\nFor.\nGeeks.\n" ; // A Raw string string string2 =
R"(Geeks.\nFor.\nGeeks.\n)";
 cout << string1 <<endl;
 cout<<string2 <<endl;

 return 0;
 }
```

**Output:**
**Geeks.**
**For.**
**Geeks.**

**Geeks. \nFor. \nGeeks. \n**

-> We can observe that for string 1 it takes \n as escape characters and each word in printed in different line

-> In string 2, it is taken as raw string. So \n is not taken as escape character and here \n is considered as normal character.