1.**Static\_assert**

Description:

The static\_assert declaration tests a software assertion at compile time. This can be especially useful for template code.

Syntax looks like this:

static\_assert ( bool\_constexpr , string )

Static assertions are a way to check if a condition is true when the code is compiled. If it isn’t, the compiler is required to issue an error message and stop the compiling process. The condition that needs to be checked is a constant expression.

Difference from assert():

An **assert statement** is a preprocessor macro that evaluates a conditional expression at runtime. If the conditional expression is true, the assert statement does nothing. If the conditional expression evaluates to false, an error message is displayed and the program is terminated. This error message contains the conditional expression that failed, along with the name of the code file and the line number of the assert. This makes it very easy to tell not only what the problem was, but where in the code the problem occurred. This can help with debugging efforts immensely.

The assert functionality lives in the <cassert> header, and is often used both to check that the parameters passed to a function are valid, and to check that the return value of a function call is valid.

C assert() example:

#include <cassert> // for assert()

int getArrayValue(const std::array<int, 10> &array, int index)

{

    // we're asserting that index is between 0 and 9

    assert(index >= 0 && index <= 9); // this is line 6 in Test.cpp

    return array[index];

}

If the program calls getArrayValue(array, -3), the program prints the following message:

Assertion failed: index >= 0 && index <=9, file Test.cpp, line 6

Unlike **assert**, which operates at runtime, **static\_assert** is designed to operate at compile time, causing the compiler to error if the condition is not true. If the condition is false, the diagnostic message is printed.

**Example 1:**

#include <iostream>

using namespace std;

template <class T, int Size>

class Vector {

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;

}

**Example 2:**

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;

}

**Example 3:**

Here’s an example of using static\_assert to ensure types have a certain size:

static\_assert(sizeof(long) == 8, "long must be 8 bytes");

static\_assert(sizeof(int) == 4, "int must be 4 bytes");

int main()

{

return 0;

}

On the author’s machine, when compiled, the compiler errors:

1>c:\consoleapplication1\main.cpp(19): error C2338: long must be 8 bytes

**2.std::bind**

**std::bind** is a Standard Function Objects that acts as a Functional Adaptor i.e. it takes a function as input and returns a new function Object as an output with with one or more of the arguments of passed function bound or rearranged.

Suppose We have a function to add two numbers i.e.

int add(int first, int second){

return first + second;

}

**std::bind takes a function as its first parameter and then that function’s argument as its parameter.**

auto add\_func = std::bind(&add, \_1, \_2);

Here **add\_func** is a function object i.e. equivalent to add().  
**std::bind** took the first parameter a function i.e. &add and then its arguments as \_1 & \_2 as his own arguments.

So, whenever we call this new function object i.e.

add\_func(4,5);

It will internally call the add() function and pass the first parameter at the place of \_1 and second at the place of \_2.

auto new\_add\_func = std::bind(&add, 12, \_1);

So, when we call **new\_add\_func(5)** it will internally call the **add()** function with first parameter will be  12 always and second parameter will 5 i.e. passed as argument.

We can also rearrange arguments using std::bind() i.e. \_1 and \_2 etc decides the place of argument to be passed i.e.

auto mod\_add\_func = std::bind(&add, \_2, \_1);

When we call **mod\_add\_func(12,15)** it is equivalent to calling **add(15, 12)**.

\_1 represents the first passed argument and \_2 second passed argument. Now while constructing new function object through std::bind we changed the order of arguments by passing \_2 first and \_1 second in the underlined function.

**Use:**

As std::bind acts as a functional adaptor and gives the a new function objects, hence it is very usefull with many STL algorithms.

**Example 1:**

#include <functional>

#include <iostream>

#include <algorithm>

using namespace std::placeholders;

int add(int first, int second)

{

return first + second;

}

bool divisible(int num , int den)

{

if(num % den == 0)

return true;

return false;

}

int approach\_1()

{

int arr[10] = {1,20,13,4,5,6,10,28,19,15};

int count = 0;

for(int i = 0; i < sizeof(arr)/sizeof(int); i++)

{

if(divisible(arr[i], 5))

count++;

}

return count;

}

int approach\_2()

{

int arr[10] = {1,20,13,4,5,6,10,28,19,15};

return std::count\_if(arr, arr + sizeof(arr)/sizeof(int) , std::bind(&divisible, \_1, 5));

}

int main()

{

int x = add(4,5);

// Will return 9

// What if we want to fix the first argument

auto new\_add\_func = std::bind(&add, 12, \_1);

x = new\_add\_func(5);

// Will return 17

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

 auto mod\_add\_func = std::bind(&add, \_2, \_1);

x = mod\_add\_func(12, 15);

// Will return 27

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

std::function<int (int) > mod\_add\_funcObj = std::bind(&add, 20, \_1);

x = mod\_add\_funcObj(15);

// Will return 35

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

std::cout<<approach\_1()<<std::endl;

std::cout<<approach\_2()<<std::endl;

return 0;

}

**Example 2:**

#include <functional>

#include <string>

#include <iostream>

using namespace std;

using namespace std::placeholders;

void show(const string& a, const string& b, const string& c)

{

    cout << a << "; " << b << "; " << c << endl;

}

int main()

{

    auto x = bind(show, \_1, \_2, \_3);

    auto y = bind(show, \_3, \_1, \_2);

    auto z = bind(show, "hello", \_2, \_1);

    x("one", "two", "three");

    y("one", "two", "three");

    z("one", "two");

    return 0;

}

The output is:

one; two; three  
three; one; two  
hello; two; one

**Example 3:**

/\*

\* This example show some ways of using std::function to call

\* a) C-like function

\* b) class-member function

\* c) operator()

\* d) lambda function

\*

\* Function call can be made:

\* a) with right arguments

\* b) argumens with different order, types and count

\*/

#include <iostream>

#include <functional>

#include <iostream>

#include <vector>

using std::cout;

using std::endl;

using namespace std::placeholders;

// simple function to be called

double foo\_fn(int x, float y, double z)

{

double res = x + y + z;

std::cout << "foo\_fn called with arguments: "

<< x << ", " << y << ", " << z

<< " result is : " << res

<< std::endl;

return res;

}

// structure with member function to call

struct foo\_struct

{

// member function to call

double foo\_fn(int x, float y, double z)

{

double res = x + y + z;

std::cout << "foo\_struct::foo\_fn called with arguments: "

<< x << ", " << y << ", " << z

<< " result is : " << res

<< std::endl;

return res;

}

// this member function has different signature - but it can be used too

// please not that argument order is changed too

double foo\_fn\_4(int x, double z, float y, long xx)

{

double res = x + y + z + xx;

std::cout << "foo\_struct::foo\_fn\_4 called with arguments: "

<< x << ", " << z << ", " << y << ", " << xx

<< " result is : " << res

<< std::endl;

return res;

}

// overloaded operator() makes whole object to be callable

double operator()(int x, float y, double z)

{

double res = x + y + z;

std::cout << "foo\_struct::operator() called with arguments: "

<< x << ", " << y << ", " << z

<< " result is : " << res

<< std::endl;

return res;

}

};

int main(void)

{

// typedefs

using function\_type = std::function<double(int, float, double)>;

// foo\_struct instance

foo\_struct fs;

// here we will store all binded functions

std::vector<function\_type> bindings;

// var #1 - you can use simple function

function\_type var1 = foo\_fn;

bindings.push\_back(var1);

// var #2 - you can use member function

function\_type var2 = std::bind(&foo\_struct::foo\_fn, fs, \_1, \_2, \_3);

bindings.push\_back(var2);

// var #3 - you can use member function with different signature

// foo\_fn\_4 has different count of arguments and types

function\_type var3 = std::bind(&foo\_struct::foo\_fn\_4, fs, \_1, \_3, \_2, 0l);

bindings.push\_back(var3);

// var #4 - you can use object with overloaded operator()

function\_type var4 = fs;

bindings.push\_back(var4);

// var #5 - you can use lambda function

function\_type var5 = [](int x, float y, double z)

{

double res = x + y + z;

std::cout << "lambda called with arguments: "

<< x << ", " << y << ", " << z

<< " result is : " << res

<< std::endl;

return res;

};

bindings.push\_back(var5);

std::cout << "Test stored functions with arguments: x = 1, y = 2, z = 3"

<< std::endl;

for (auto f : bindings)

f(1, 2, 3);

}

Live

Output:

Test stored functions with arguments: x = 1, y = 2, z = 3

foo\_fn called with arguments: 1, 2, 3 result is : 6

foo\_struct::foo\_fn called with arguments: 1, 2, 3 result is : 6

foo\_struct::foo\_fn\_4 called with arguments: 1, 3, 2, 0 result is : 6

foo\_struct::operator() called with arguments: 1, 2, 3 result is : 6

lambda called with arguments: 1, 2, 3 result is : 6

**3.String**

**--------------------------------------------------------------------------------**

**1**.Raw string literal

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 )”.

The unusual form **R”(**<actual string>**)”**  is the raw string modifier. When encountering  a raw string, the compiler will treat all characters in the string literal as simple  plain characters including normally escaped characters  . For example,  It will not attempt to translate ‘**\t**‘  into a tab character,  instead, the string literal will include the actual sequence “**\t**”

std::string str = R"(The '\n' character can easily be shown)" "\n" "and used";

std::cout << str;

output:

The ‘\n’ character can easily be shown  
and used

Basically a raw string literal is a string in which the escape characters (like \n \t or \" ) of C++ are not processed. A raw string literal starts with R"( and ends in )", let's see an in an example the difference between a normal string and a raw string in C++:

Example:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string normal\_str="First line.\nSecond line.\nEnd of message.\n";

string raw\_str=R"(First line.\nSecond line.\nEnd of message.\n)";

cout<<normal\_str<<endl;

cout<<raw\_str<<endl;

return(0);

}

normal\_str will be processed at compilation time so you will see three lines of text and an empty line. In the case of the variable raw\_str which is a raw string literal, the compiler will not process the escape characters, so you will see a single line of text with a content identical with what you have in the C++ source code.

4.**std::array**

The array is a container for constant size arrays. This container wraps around fixed size arrays and also doesn’t loose the information of its length when decayed to a pointer.  
In order to utilize array, we need to include the array header:

Example:

#include <array>

#include <algorithm>

#include <array>

#include <iostream>

#include <iterator>

#include <string>

using namespace std;

int main() {

// construction uses aggregate initialization

// double-braces required

array<int, 5> ar1{{3, 4, 5, 1, 2}};

array<int, 5> ar2 = {1, 2, 3, 4, 5};

array<string, 2> ar3 = {{string("a"), "b"}};

cout << "Sizes of arrays are" << endl;

cout << ar1.size() << endl;

cout << ar2.size() << endl;

cout << ar3.size() << endl;

cout << "\nInitial ar1 : ";

for (auto i : ar1)

cout << i << ' ';

// container operations are supported

sort(ar1.begin(), ar1.end());

cout << "\nsorted ar1 : ";

for (auto i : ar1)

cout << i << ' ';

// Filling ar2 with 10

ar2.fill(10);

cout << "\nFilled ar2 : ";

for (auto i : ar2)

cout << i << ' ';

// ranged for loop is supported

  cout << "\nar3 : ";

  for (auto &s : ar3)

    cout << s << ' ';

  return 0;

}

std::array is a great replacement for built-in fixed arrays. It's efficient, in that it doesn’t use any more memory than built-in fixed arrays. The only real downside of a std::array over a built-in fixed array is a slightly more awkward syntax, that you have to explicitly specify the array length (the compiler won’t calculate it for you from the initializer), and the signed/unsigned issues with size and indexing. But those are comparatively minor quibbles — we recommend using std::array over built-in fixed arrays for any non-trivial array use.

5.**std::tuple**

A tuple is an object that can hold a number of elements. The elements can be of different data types. The elements of tuples are initialized as arguments in order in which they will be accessed.

**Operations on tuple** :-

**1. get()** :- get() is used to access the tuple values and modify them, it accepts the index and tuple name as arguments to access a particular tuple element.

**2. make\_tuple()** :- make\_tuple() is used to assign tuple with values. The values passed should be in order with the values declared in tuple.

// C++ code to demonstrate tuple, get() and make\_pair()

#include<iostream>

#include<tuple> // for tuple

using namespace std;

int main()

{

// Declaring tuple

tuple <char, int, float> geek;

// Assigning values to tuple using make\_tuple()

geek = make\_tuple('a', 10, 15.5);

// Printing initial tuple values using get()

cout << "The initial values of tuple are : ";

cout << get<0>(geek) << " " << get<1>(geek);

cout << " " << get<2>(geek) << endl;

// Use of get() to change values of tuple

get<0>(geek) = 'b';

get<2>(geek) = 20.5;

// Printing modified tuple values

cout << "The modified values of tuple are : ";

cout << get<0>(geek) << " " << get<1>(geek);

cout << " " << get<2>(geek) << endl;

return 0;

}

Output:

The initial values of tuple are : a 10 15.5

The modified values of tuple are : b 10 20.5

**tuple\_size** :- It returns the number of elements present in the tuple.

Example:

//C++ code to demonstrate tuple\_size

#include<iostream>

#include<tuple> // for tuple\_size and tuple

using namespace std;

int main()

{

// Initializing tuple

tuple <char,int,float> geek(20,'g',17.5);

// Use of size to find tuple\_size of tuple

cout << "The size of tuple is : ";

cout << tuple\_size<decltype(geek)>::value << endl;

return 0;

}

Swap():The swap(), swaps the elements of the two different tuples.

The contents of the [tuple](http://www.cplusplus.com/tuple) object x are exchanged with those of y. Both objects must be of the same type (i.e., contain the same types of elements).

Example:

// swap tuples

#include <iostream> // std::cout

#include <tuple> // std::tuple, std::get

int main ()

{

std::tuple<int,char> a (10,'x');

std::tuple<int,char> b (20,'y');

swap(a,b);

std::cout << "a contains: " << std::get<0>(a) << " and " << std::get<1>(a) << '\n';

return 0;

}

Output:

a contains: 20 and y