Contents

[New Language Features 2](#_Toc405364471)

[*1.* nullptr and std::nullptr\_t 2](#_Toc405364472)

[*2.* Automatic Type Deduction with auto 3](#_Toc405364473)

[*3.* Uniform Initialization and Initializer Lists 4](#_Toc405364474)

[*a.* Uniform Initialization 4](#_Toc405364475)

[*b.* Initializer Lists 5](#_Toc405364476)

[*4.* Range-Based for Loops 7](#_Toc405364477)

[*5.* Keyword decltype 8](#_Toc405364478)

[*6.* Suffix/Trailing Return Type 9](#_Toc405364479)

[*7.* Lambda Expressions 10](#_Toc405364480)

[*a.* What are Function Objects? 10](#_Toc405364481)

[*8.* Keyword constexpr 15](#_Toc405364482)

[*a.* Constant expressions 15](#_Toc405364483)

[*b.* Difference between `constexpr` and `const` 15](#_Toc405364484)

[*9.* Move Semantics and Rvalue References 19](#_Toc405364485)

[*a.* Lvalues and Rvalues 19](#_Toc405364486)

# New Language Features

## nullptr and std::nullptr\_t

* C++11 lets you use nullptr instead of 0 or NULL to specify that a pointer refers to no value.
* nullptr is a new keyword. Both true and false are keywords and as literals they have a type (bool).
* nullptr has type std::nullptr\_t, defined in <cstddef>
* nullptr is a prvalue (you cannot take the address of it using &).
* nullptr is Type-safe and Clear-Cut Null Pointer.
* Standard requires that sizeof(nullptr\_t) be sizeof(void\*).
* Integral null pointer constant can be converted to std::nullptr\_t. The opposite direction is not allowed.
* A cast of nullptr\_t to an integral type needs a reinterpret\_cast.

Example:

#include <iostream>

#include <vector>

using namespace std;

int main(void)

{

std::nullptr\_t ptr;

ptr = nullptr;

std::nullptr\_t \*ptr1 = &ptr;

//ptr1 = &nullptr; // nullptr is a prvalue.

// integral null pointer constant can be converted to std::nullptr\_t.

// The opposite direction is not allowed.

ptr = 0;

ptr = NULL;

//int b = nullptr; // Not allowed

// A cast of nullptr\_t to an integral type needs a reinterpret\_cast

int b = reinterpret\_cast<int>(nullptr);

int a = NULL; // This we can do.

// The Standard requires that sizeof(nullptr\_t) be sizeof(void\*).

cout<<"Size of void \* = "<<sizeof(void\*)<<endl;

cout<<"Size of nullptr= "<<sizeof(nullptr)<<endl;

return 0;

}

Example:

#include <iostream>

using namespace std;

auto function(int x)->int

{

cout<<"Value of X: "<<x<<endl;

return x;

}

auto function(char\* ptr)->int

{

if(ptr == nullptr)

{

cout<<"Pointer is null. Allocating memory"<<endl;

// Do some stuff here...

}

else

cout<<"Value contained in the pointer: "<<\*ptr<<endl;

return 0;

}

auto main(void)->int

{

function(NULL);

function(nullptr);

char ch='A';

char\* ptr1 = &ch;

function(ptr1);

return 0;

}

## Automatic Type Deduction with auto

* We can declare a variable or an object without specifying its specific type by using auto.

Example:

auto i = 42; // i has type int

double f();

auto d = f(); // d has type double

* The type of a variable declared with auto is deduced from its initializer. Thus, an initialization is required.

auto i; // ERROR: can’t deduce the type of i

* Additional qualifiers are allowed. For example:

static auto vat = 0.19;

* Usefulness of auto…
  + Robustness: If the expression’s type is changed—this includes when a function return type is changed—it just works.
  + Usability: You don't have to worry about type name spelling difficulties and typos. (DRY – Don’t Repeat Yourself)
  + Easier refactoring.
  + Efficiency: Your coding can be more efficient.
* The auto keyword is a simple way to declare a variable that has a complicated type.
* When auto sets the type of a declared variable from its initializing expression, it proceeds as follows:
  + If the initializing expression is a reference, the reference is ignored.
  + If, after Step 1 has been performed, there is a top-level const and/or volatile qualifier, it is ignored.

## Uniform Initialization and Initializer Lists

### Uniform Initialization

* Before C++11, novice programmers, could easily become confused by the question of how to initialize a variable or an object. Initialization could happen with parentheses, braces, and/or assignment operators.

int a(1); // Variable definition

int b(); // Function declaration

int b(foo); // Variable definition or function declaration

// Initialize array variable

string a[] = { "foo", " bar" };

// Error: initializer list for non-aggregate vector

vector<string> v = { "foo", " bar" }; // Error

void f(string a[]);

f( { "foo", " bar" } ); // Syntax error: block as argument

* For this reason, C++11 introduced the concept of uniform initialization, which means that for any initialization, you can use one common syntax. This syntax uses braces.

Example 1:

int x1(5.3); // OK, but x1 becomes 5

int x2 = 5.3; // OK, but x2 becomes 5

// Uniform Initialization

int x3{5.3}; // ERROR: narrowing

int x4 = {5.3}; // ERROR: narrowing

char c1{7}; // OK: even though 7 is an int, this is not narrowing

char c2{999}; // ERROR: narrowing (if 999 doesn’t fit into a char)

Example 2:

#include <iostream>

using namespace std;

auto main(void)->int

{

int x1 = int{1};

int x2 = {1}; // The = is optional

int x3{1};

int\* p = new int{1};

cout<<"Value of X1: "<< x1 <<endl;

cout<<"Value of X2: "<< x2 <<endl;

cout<<"Value of X3: "<< x3 <<endl;

cout<<"Value of \*p: "<< \*p <<endl;

return 0;

}

### Initializer Lists

* To support the concept of initializer lists for user-defined types, C++11 provides the class template std::initializer\_list<>. It can be used to support initializations by a list of values or in any other place where you want to process just a list of values.

Example 1:

#include <iostream>

#include <initializer\_list>

using namespace std;

void print(std::initializer\_list<int> vals)

{

// Process a list of values

for(auto a=vals.begin(); a!=vals.end(); ++a)

{

cout << \*a << "\n";

}

}

int main(void)

{

print({12,3,5,7,11,13,17}); // Pass a list of values to print()

return 0;

}

#include <iostream>

#include <initializer\_list>

using namespace std;

class TempClass

{

public:

TempClass(int,int)

{

cout << "TempClass(int,int) constructor called here" << endl;

}

TempClass(std::initializer\_list<int>)

{

cout << "Constructor with initializer\_list is called here"<<endl;

}

};

int main(void)

{

TempClass p(77,5); // Calls TempClass::TempClass(int,int)

TempClass q{77,5}; // Calls TempClass::TempClass(initializer\_list)

TempClass r{77,5,42};// Calls TempClass::TempClass(initializer\_list)

TempClass s = {77,5};// Calls TempClass::TempClass(initializer\_list)

return 0;

}

Example 2:

## Range-Based for Loops

* C++11 introduces a new form of for loop, which iterates over all elements of a given range, array, or collection.
* The general syntax is as follows:

for ( decl : coll )

{

statement;

}

Example 1:

#include <iostream>

#include <initializer\_list>

using namespace std;

int main(void)

{

for ( int i : { 2, 3, 5, 7, 9, 13, 17, 19 } )

{

cout << i << endl;

}

int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for( auto y : x )

{

cout << y << " ";

}

cout<<endl;

for( auto &y : x )

{

y = y \* 2;

cout << y << " ";

}

return 0;

}

Example 2:

#include <iostream>

#include <vector>

using namespace std;

int main(void)

{

vector<int> vec;

vec.push\_back( 10 );

vec.push\_back( 20 );

for (int i : vec ) { cout << i <<endl; }

return 0;

}

## Keyword decltype

* The decltype type specifier yields the type of a specified expression.

Syntax:

decltype(expression)

Example:

int main(void)

{

int i = 4;

const int j = 6;

const int& k = i;

int a[5];

int \*p;

decltype(i) var1; // int

decltype(1) var2; // int

decltype(2+3) var3; // int(+ operator returns an rvalue)

decltype(i=1) var4 = i; // int&, assignment to int returns an lvalue

decltype((i)) var5 = i; // int&

decltype(j) var6 = 1; // const int

decltype(k) var7 = j; // const int&

decltype(a) var8; // int[5]

decltype(\*p) var11 = i; // int&(\*operator returns an lvalue)

return 0;

}

## Suffix/Trailing Return Type

* Sometimes, the return type of a function depends on an expression processed with the arguments.

template <typename T1, typename T2>

decltype(x+y) add(T1 x, T2 y);

* Unfortunately, the above code doesn't work. The compiler will parse codes from left to right, so it will issue an error message to indicate that variables ‘x’ and ‘y’ are used before their declarations.
* To solve this problem, C++11 introduced a feature called trailing return types. Using this feature, the previous program can be rewritten as follows:

template <typename T1, typename T2>

auto add(T1 x, T2 y) -> decltype(x+y);

Example:

#include <iostream>

using namespace std;

template<typename T1, typename T2>

auto sum(T1 & t1, T2 & t2) -> decltype(t1 + t2)

{

return t1 + t2;

}

int main(void)

{

auto i = 1;

auto j = 1.3;

auto k = sum(i,j);

cout << k << endl;

return 0;

}

* We specify the function return type after the declaration of parameter declarations. Composite symbol “-> decltype(x+y)” is called a ***Trailing Return Type***. The auto keyword is placed before the function identifier, which is the placeholder of the return type specifier. When a trailing return type is used, the placeholder return type must be auto. Meanwhile, the auto type specifier cannot be used in a function declaration without a trailing return type.

## Lambda Expressions

* A lambda expression, sometimes also referred to as a lambda function or as a lambda, is a simplified notation for defining and using an anonymous function object.

### What are Function Objects?

* A function object is a computer programming construct allowing an object to be invoked or called as if it were an ordinary function, usually with the same syntax.
* A typical use of a function object is in writing callback functions. A callback in procedural languages, such as C, may be performed by using function pointers.

#include <stdlib.h>

#include <iostream>

using namespace std;

// Callback function.

int compareInts(const void\* a, const void\* b)

{

return \*(const int \*)a - \*(const int \*)b;

}

int main(void)

{

int items[] = { 4, 3, 1, 2 };

qsort(items, sizeof(items)/sizeof(items[0]),sizeof(items[0]), compareInts);

for(int i=0; i< 4; i++)

cout<< items[i] << endl;

return 0;

}

* In C++ a function object may be used instead of an ordinary function by defining a class that overloads the function call operator by defining an operator() member function. In C++ this is called a class type functor, and may appear as follows:

#include <iostream>

using namespace std;

// Comparator predicate: returns true if a < b, false otherwise

struct IntComparator

{

bool operator()(const int &a, const int &b) const

{

return a < b;

}

};

int main()

{

IntComparator cpm;

int a = 10;

int b = 55;

if(cpm(a, b))

{

cout<< "a is Less Than b" << endl;

}

else

cout<< "a is greater than or equal to b" << endl;

return 0;

}

* Instead of defining a named class with an operator(), later making an object of that class, and finally invoking it, we can use a shorthand.
* So what is a ***Lambda***? It's an expression that represents the idea of doing something, performing some operation or calculation.
* A ***lambda expression*** represents a callable unit of code. It can be thought of as an unnamed, inline function.
* Like any function, a lambda has a ***return type***, a ***parameter list***, and a ***function body***. Unlike a function, lambdas may be defined inside a function.

Syntax:

[capture\_list] (parameter\_list) -> return type { function\_body }

* A lambda expression consists of a sequence of parts:
  + A possibly empty capture list, specifying what names from the definition environment can be used in the lambda expression’s body, and whether those are copied or accessed by reference. The capture list is delimited by [].
  + An optional parameter list, specifying what arguments the lambda expression requires. The parameter list is delimited by ().
  + An optional mutable specifier, indicating that the lambda expression’s body may modify the state of the lambda.
  + An optional noexcept specifier.
  + An optional return type declaration of the form -> type
  + A body, specifying the code to be executed. The body is delimited by {}

Some Simple Lambdas:

#include <iostream>

using namespace std;

int main(void)

{

// Lambda That Prints Hello World.

[] {cout << "Hello World" << endl; } ();

// Lambda That Does Nothing. It’s a Valid Lambda.

[]{}();

// Lambda That Returns A Double

auto l1 = [] () -> double { return 50;};

cout<< l1() << endl;

return 0;

}

* Above code can be rewritten as shown below…

#include <iostream>

using namespace std;

int main(void)

{

auto returnValue = [=]()->bool {return a < b; };

if(returnValue())

{

cout<< "a is Less Than b" <<endl;

}

else

cout<< "a is greater than or equal to b" << endl;

return 0;

}

* Especially when we are using Standard Library, there will be so many tiny functions.

Example:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void print(int i){ cout << i << " "; }

int main(void)

{

vector<int> v;

v.push\_back(3);

v.push\_back(6);

v.push\_back(9);

v.push\_back(2);

for\_each(v.begin(),v.end(),print);

cout << endl;

return 0;

}

* Lambdas help us to get rid of these tiny functions…

Example:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void print(int i) { cout << i << " "; }

bool odd(int i) { return i%2; }

int main(void)

{

vector<int> v;

v.push\_back(3);

v.push\_back(6);

v.push\_back(9);

v.push\_back(2);

for\_each(v.begin(),v.end(),print);

cout << endl;

for\_each(v.begin(), v.end(), [](int n) { cout << n << " "; } );

cout << endl;

auto oDD = find\_if(v.begin(),v.end(),odd);

while(oDD!=v.end())

{

cout<<\*oDD << " ";

oDD = find\_if(++oDD,v.end(),odd);

}

cout << endl;

for\_each(

v.begin(),

v.end(),

[](int n) {

cout << n;

if (n % 2 == 0) {

cout << " even " << endl;

}

else {

cout << " odd " << endl;

}

}

);

return 0;

}

## Keyword constexpr

* The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time. Such variables and functions can then be used where only compile time ***constant expressions*** are allowed.

### Constant expressions

* Defines an expression that can be evaluated at compile time. Such expressions can be used as non-type template arguments, array sizes, and in other contexts that require constant expressions.

Example:

#include <iostream>

#include <algorithm>

using namespace std;

int main(void)

{

const int cn = 2;

array<int, cn> a2; // OK, cn is a constant expression

//int n = 1;

//array<int, n> a1; // error, n is not a constant expression

return 0;

}

### Difference between `constexpr` and `const`

* A constant expression is an expression whose value cannot change and that can be evaluated at compile time.
* A literal is a constant expression. A const object that is initialized from a constant expression is also a constant expression.
* const guarantees that a program doesn’t change a variable’s value. However, const doesn’t guarantee which type of initialization the variable undergoes. For instance, a const variable initialized by a function call requires dynamic initialization (the function is called at runtime; consequently, the constant is initialized at runtime).
* With certain functions, there’s no justification for this restriction because the compiler can evaluate the function call statically, effectively replacing the call with a constant value.

Example:

#include <iostream>

#include <algorithm>

using namespace std;

int main(void)

{

const int mx = numeric\_limits<int>::max();// dynamic initialization

cout<< mx <<endl;

return 0;

}

* The function max()merely returns a literal value. However, because the initializer is a function call, mx undergoes dynamic initialization. Therefore, you can’t use it as a constant expression:

Example:

int arr[mx]; //compilation error: “constant expression required”

* constexpr functions guarantee compile-time evaluation so long as their arguments are constant expressions, too. However, if any of the arguments isn’t a constant expression, the constexpr function may be evaluated dynamically.

Example:

constexpr int square(int x)

{

return x \* x; //OK, compile time evaluation only if x is a constant //expression

}

const int res=square(5); //compile-time evaluation of square(5)

int y=getval();

int n=square(y); //dynamic evaluation of square(y)

* A constant-expression value is a variable or data member declared constexpr. It must be initialized with a constant expression or an rvalue constructed by a constant expression constructor with constant expression arguments.
* A constexpr value behaves as if it was declared const, except that it requires initialization before use and its initializer must be a constant expression.

Example 2:

#include <iostream>

#define NUM 42

using namespace std;

template <unsigned int N>

struct Fibonacci

{

enum { value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value };

};

template <>

struct Fibonacci<1>

{

enum { value = 1 };

};

template <>

struct Fibonacci<0>

{

enum { value = 1 };

};

constexpr unsigned int fib(unsigned int n)

{

return (n > 1 ? fib(n-1) + fib(n-2) : 1 );

}

int main()

{

const long i = fib(NUM); // here i should be initialized at the time of

// declaration

cout << "Meta\_fib(NUM): " << Fibonacci<NUM>::value << endl; // compile time cout << "Constexpr\_fib(NUM) : " << i << endl;

cout << "Constexpr\_fib(NUM) : " << fib(NUM) << endl; // run time :-?

return 0;

}

Example 3:

const int max\_files = 20; // max\_files is a constant expression

const int limit = max\_files + 1;  // limit is a constant expression

int \_size = 27;  // \_size is not a constant expression

const int sz = get\_size();  // sz is not a constant expression

* Although \_size is initialized from a literal, it is not a constant expression because it is a plain int, not a const int.
* Even though sz is a const, the value of its initializer is not known until run time. Hence, sz is not a constant expression.

## Move Semantics and Rvalue References

We need to understand some concepts before we move forward…

### Lvalues and Rvalues

Q: What is a ***value***?

A: ***Value*** is an expression which cannot be evaluated any further.

Q: What are ***Lvalues***?

A: ***Lvalues*** (locator value) have storage addresses, means they are variables. When we use an object as an ***lvalue***, we use the object’s identity (its location in memory).

Q: What are ***Rvalues***?

A: ***Rvalues*** are the value of an expression. ***Rvalue*** is just a term only used to distinguish from ***Lvalue***. When we use an object as an ***rvalue***, we use the object’s value (its contents).

Example:

* Assume that we have written expression 5 + 6. This expression evaluates to 11, but in the program we have not explicitly designated where in the computer this 11 is stored, the expression is an Rvalue.
* Assume that we have declared a variable ***‘y’***. And we are assigning ***y = 5 + 6***. Now ***‘y’*** has the value 11. So variable ***‘y’*** is an Lvalue (Modifiable Lvalue).

***Note:*** The term “***Modifiable Lvalue***” is used because; ***const*** variables can’t be modified even though it has storage.

*Note: Example in the next page*

* In C++11, an expression can be an:
  1. ***rvalue***
  2. ***lvalue***
  3. ***xvalue***
  4. ***glvalue***
  5. ***prvalue***
* Two categories have become five categories.
  + What are these new categories of expressions?
  + How do these new categories relate to the existing ***rvalue*** and ***lvalue*** categories?
  + Why are these new categories needed?

Example:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*This C program will not compile\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <conio.h>

int function1(void)

{

return 5 + 6;

}

int main()

{

int x;

const int y = 7;

function1() = 10;// We can’t do this as function returns rvalue

x = function1(); // This is Right

y = 100; // Variable ‘y’ is declared as const

5 + 6 = x; // We cannot store x into 5 + 6

getch();

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*OUTPUT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Code will not compile.

Error: expression must be a modifiable lvalue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/