C++ 17 Language Features

Contents

[1. Structured bindings 2](#_Toc95995392)

[2. Selection statements with initializer 3](#_Toc95995393)

[a. Structured Binding and ‘if’ with initializer 4](#_Toc95995394)

[3. Inline variables 4](#_Toc95995395)

[4. Aggregates with Base classes 4](#_Toc95995396)

[5. constexpr lambda 7](#_Toc95995397)

[6. New Attributes 8](#_Toc95995398)

[7. Adding attributes to namespaces and enumerators 10](#_Toc95995399)

[8. Nested namespaces 11](#_Toc95995400)

[9. Heap allocation with alignment 12](#_Toc95995401)

[10. \_\_has\_include Preprocessor Expression 13](#_Toc95995402)

[11. constexpr if 14](#_Toc95995403)

[12. Fold expressions 15](#_Toc95995404)

[Variadic Arguments 15](#_Toc95995405)

[Variadic Templates 15](#_Toc95995406)

[Parameter Packs 15](#_Toc95995407)

[Fold Expression 16](#_Toc95995408)

[13. Template argument deduction for class templates 17](#_Toc95995409)

[14. Lambda capture this by value 18](#_Toc95995410)

[15. UTF-8 character literals 18](#_Toc95995411)

[16. Direct list initialization of enums 20](#_Toc95995412)

[17. New rules for auto deduction from braced-init-list 20](#_Toc95995413)

[18. Declaring non-type template parameters with auto 21](#_Toc95995414)

[Non-type parameters 21](#_Toc95995415)

# Structured bindings

* Structured Bindings give us the ability to declare multiple variables initialized from a tuple or struct.
* The main purpose of Structured Bindings in C++ 17 is to make the code clean and easy to understand.
* Like a reference, a structured binding is an alias to an existing object.

Example 1:

|  |
| --- |
| struct MyStruct {    int x;    double y;  };  int main() {    MyStruct obj{5, 0.5};    auto [i, f] = obj; // Structured Binding    cout << "Integer Value: " << i << "\nDouble Value: " << f << endl;    return 0;  } |

Example 2:

|  |
| --- |
| tuple<string, int, char> fun(void){      tuple<string, int, char> p = {"Nishith", 36, 'M'};      return p;  }  int main() {    auto [name, age, sex] = fun();    cout << "Name: " << name << "\nAge: " << age << "\nSex: " << sex << "\n";    return 0;  } |

Example 3:

|  |
| --- |
| MyStruct fun(void) {    MyStruct ms = {5, 0.5};    return ms;  }  int main() {    auto [i, d] = fun();    cout << "Integer Value: " << i << "\nDouble Value: " << d << endl;    return 0;  } |

Example 4:

|  |
| --- |
| int arr[2] = {1, 2};  auto fun(void) -> int(&)[2] { // Returns reference to int array    return arr;  }  int main() {    auto [i1, i2] = fun();    cout << "a[0]: " << i1 << "\na[1]: " << i2 << endl;    return 0;  } |

Example 5:

|  |
| --- |
| int main() {      map<std::string, int> mapping { {"a", 1},                                      {"b", 2},                                      {"c", 3} };          // const auto reference!!!      for (const auto& [key, value] : mapping) {  // Structured Binding          cout << "Key: " << key << " Value: " << value << endl;      }  } |

# Selection statements with initializer

* New additional syntax for ‘if’ and ‘switch’.

if(initialization; condition)

switch(initialization; condition)

Old New

|  |
| --- |
| *std*::*lock\_guard*<*std*::*mutex*> lk(mx);  if (v.*empty*()) {  v.*push\_back*(*val*);  } |

|  |
| --- |
| if (*std*::*lock\_guard*<*std*::*mutex*> lk(mx); v.*empty*()) {  v.*push\_back*(*val*);  } |

* The variable that we initialize here is available till the end of optional ‘*else’* statement.

Old

|  |
| --- |
| Foo gadget(argc);  switch (auto s = gadget.status()) {      case OK: gadget.zip(); break;      case Bad: throw BadFoo(s.message());  } |

|  |
| --- |
| switch (Foo gadget(args); auto s = gadget.status()) {      case OK: gadget.zip(); break;      case Bad: throw BadFoo(s.message());  } |

New

# Structured Binding and ‘if’ with initializer

Old

|  |
| --- |
| map<string, int> coll;  auto ret = coll.insert({"new", 42});  if(!ret.second){      const auto& ele = \*(ret.first);      cout << "Already there: " << ele.first << endl;  } |

New:

|  |
| --- |
| map<string, int> coll;  if(auto [pos,done] = coll.insert({"new",42}; !done)){      const auto& [key,value] = \*pos;      cout << "Already there: " << key << endl;  } |

# Inline variables

* As we know, we must define a static/global object in a CPP file.
* The rule is, if we have a global or static object, it must be initialized in one translational unit.
* With 'Inline', we don’t need CPP files to define static/global objects.
* So, static variables marked with inline count as definitions.

|  |  |
| --- | --- |
| #ifndef MONITOR\_H  #define MONITOR\_H  #include <iostream>  #include <string>  class Monitor {  public:  Monitor() {}  void log(const *std*::*string*& msg) {  *std*::*cout* << msg << *std*::*endl*;  }  };  // Declaring the global monitor in Header File.  inline Monitor progMonitor;  #endif | // Source.cpp  #include "Monitor.h"  int main(void) {  progMonitor.log("Inside main()");  return 0;  } |
| // init.cpp  #include "Monitor.h"  void init() {  progMonitor.log("Inside init()");  ...  } |

# Aggregates with Base classes

* What are aggregates and why they are special?
  + An aggregate is an array or a class with no user-declared constructors, no private or protected non-static data members, no base classes, and no virtual functions.
  + Any array is an aggregate.
  + A class (or struct, or union) is an aggregate if and only if it satisfies the below criteria...
    - This does not mean an aggregate class cannot have constructors, in fact it can have a default constructor and/or a copy constructor if they are implicitly declared by the compiler, and not explicitly by the user.
    - No private or protected non-static data members. You can have as many private and protected member functions (but not constructors) as well as as many private or protected static data members and member functions as you like and not violate the rules for aggregate classes.
    - An aggregate class can have a user-declared/user-defined copy-assignment operator and/or destructor.
  + An array is an aggregate even if it is an array of non-aggregate class type.

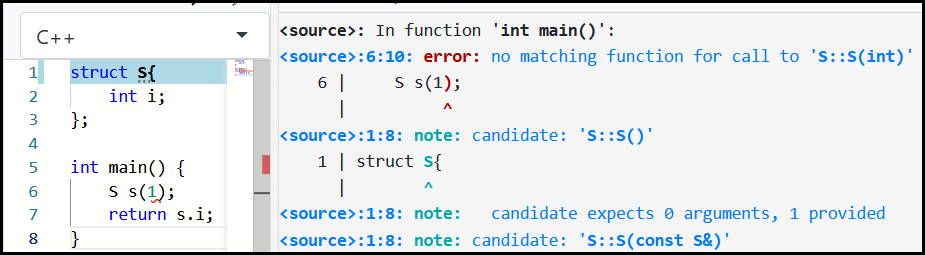
Example:

|  |
| --- |
| class NotAggregate1 {  virtual void f() {} // Should not contain virtual functions.  };  class NotAggregate2 {  int x; // Should not contain private/protected data members.  };  class NotAggregate3 {  public:  NotAggregate3(int) {} // Should not contain user-defined constructor.  };  class Aggregate1 {  public:  NotAggregate1 member1; // Public member  Aggregate1& operator=(Aggregate1 const & rhs) {/\* \*/ } // Copy-assignment  private:  void f() {} // Private function is okay, but not data member.  }; |

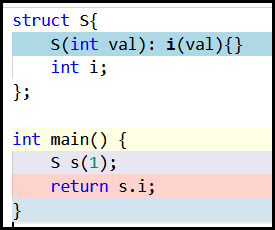
* How aggregates are special?
  + They, unlike non-aggregate classes, can be initialized with curly braces {}.

**What is the feature introduced in C++17?**

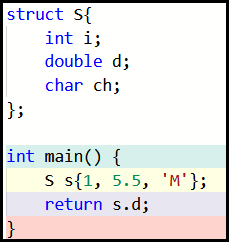
* In C++03, below code will give error.



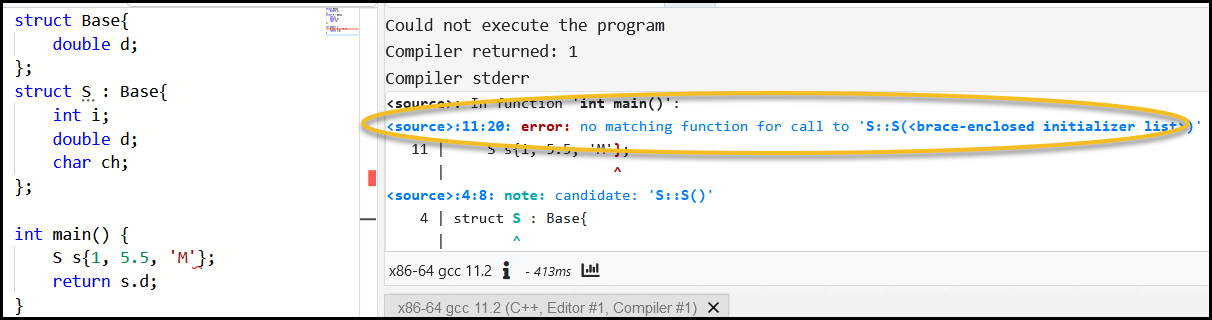
* To fix, we need to add constructor…



* In C++11, we were given the ability of uniform initialization syntax… Hence the below code would work…

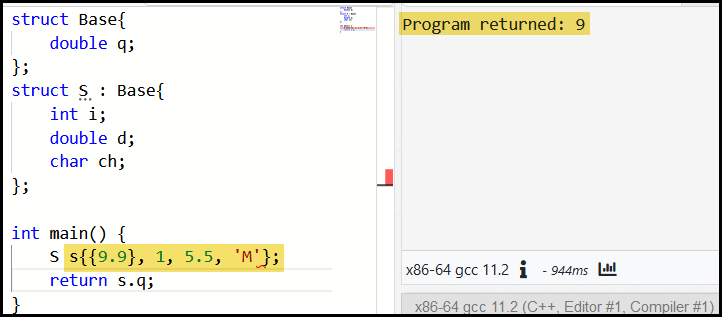


* If we introduce a class hierarchy in the above code… Let’s say…



* What C++17 is giving us is the ability to initialize the base class also…

Example 1:



Example 2:

|  |
| --- |
| #include<iostream>  #include<complex>  using namespace std;  template<typename T>  struct D: std::string, std::complex<T> {      std::string data;  };  int main(int argc, char\*\* argv) {      D<float> d{{"Hello"}, {9.9, 8.8}, "World"};      cout << d.data << endl;      cout << static\_cast<string>(d) << endl;      cout << static\_cast<complex<double>>(d) << endl;      cout << is\_aggregate<decltype(d)>::value << endl;      return 0;  } |

# constexpr lambda

* Lambdas are by default constexpr now (if possible)
  + Can be forced with constexpr

|  |
| --- |
| #include <iostream>  using namespace std;  int main(void) {      auto squared = [](auto val) { // Implicitly constexpr          return val \* val;      };      // Can be used to as the size of array!      int myArray[squared(4)] = { 9 };      for (auto i : myArray) {          cout << i << endl;      }      return 0;  } |

* We can also signal that we want the lambda to be available at compile time by…

|  |
| --- |
| auto squared = [](auto val) constexpr {      return val \* val;  }; |

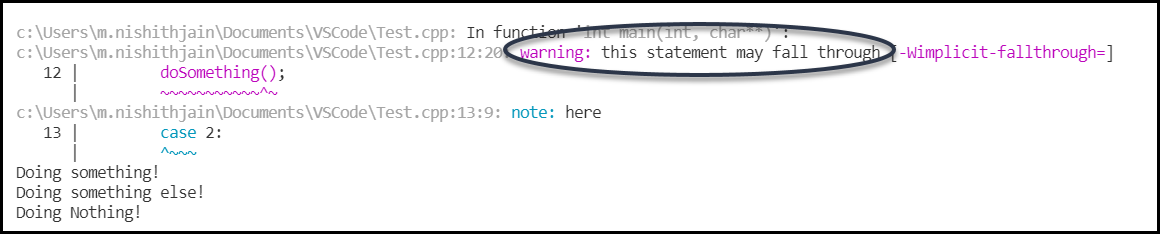
# New Attributes

* C++17 introduces three new attributes: [[fallthrough]], [[nodiscard]] and [[maybe\_unused]].

[[fallthrough]];

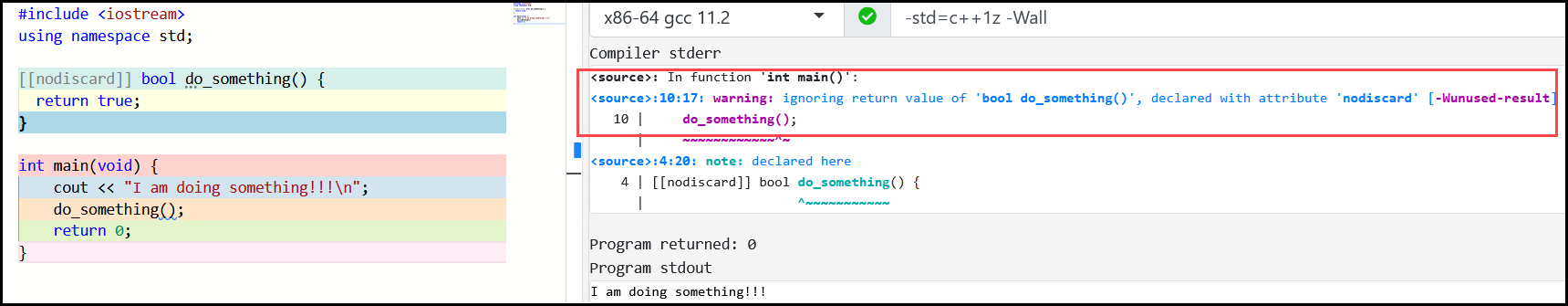
* This attribute indicates to the compiler that falling through in a switch statement is intended behaviour.

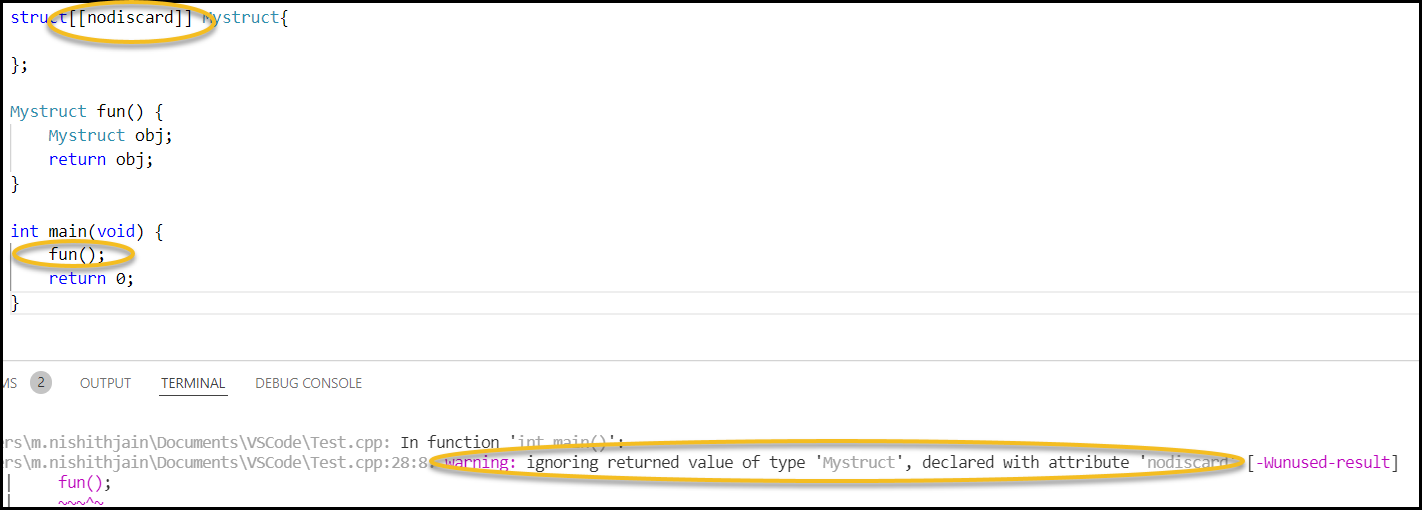
|  |
| --- |
| #include <iostream>  using namespace std;  void doSomething(void) {cout << "Doing something!\n";}  void doSomethingElse(void) {cout << "Doing something else!\n";}  void doNothing(void) {cout << "Doing Nothing!\n";}  int main(int argc, [[maybe\_unused]]char \*\*argv){      argv = nullptr;      switch(argc){          case 1:          doSomething();  // Compiler will give warning here!          case 2:          doSomethingElse();          [[fallthrough]]; // Intended fallthrough behavior.          case 3:          doNothing();          break;      }      return 0;  } |



[[nodiscard]];

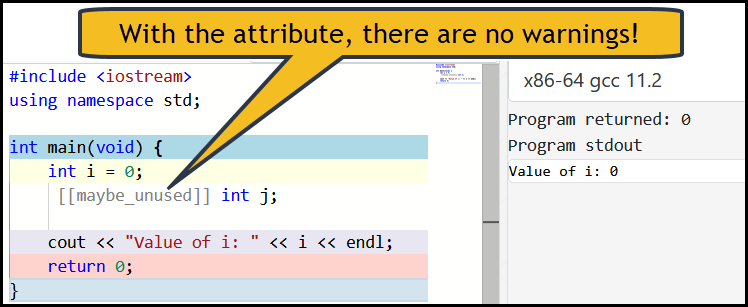
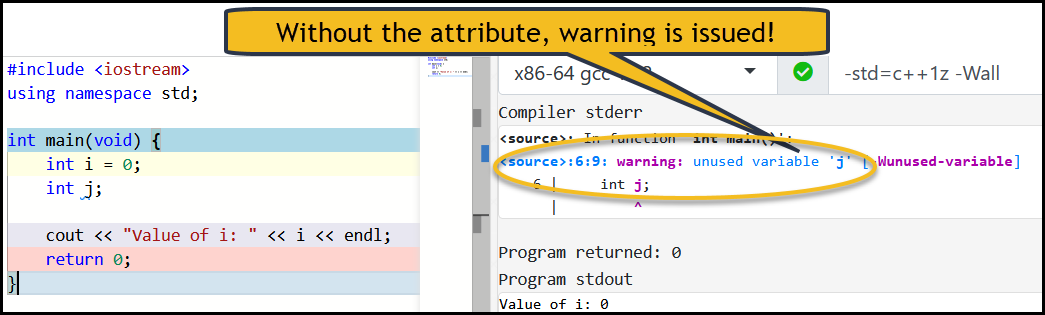
* This attribute issues a warning when either a function or class has this attribute, and its return value is discarded.





[[maybe\_unused]]

* This attribute indicates to the compiler that a variable or parameter might be unused and is intended.
* Even we can mark the functions [[maybe\_unused]].



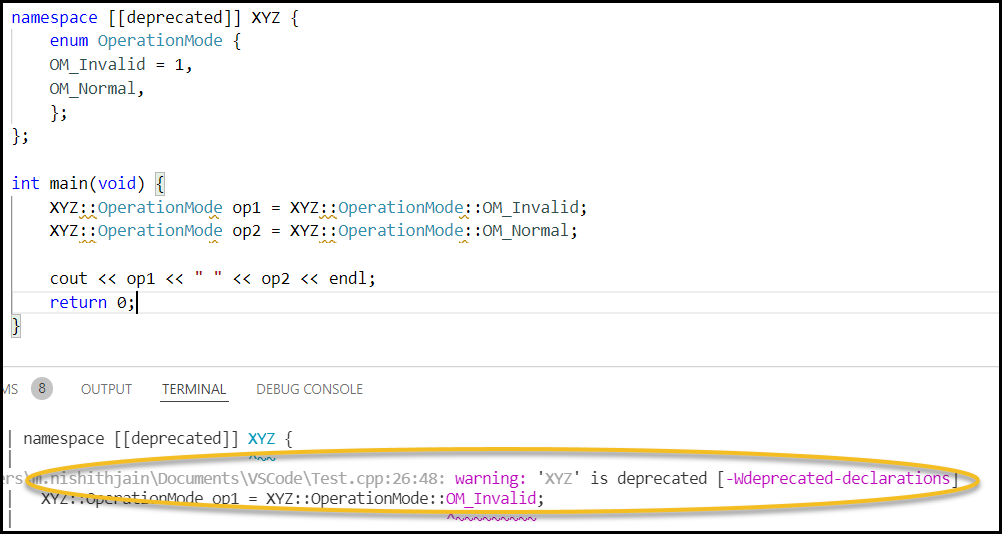
# Adding attributes to namespaces and enumerators

* In C++17, we have new places where we can add attributes. i.e. ‘namespace’ and ‘enumerators’.

|  |
| --- |
| enum OperationMode {    OM\_Invalid,    OM\_Normal,    OM\_Terrified [[deprecated("Re-named to OM\_Invalid")]],    OM\_AbortOnError  [[deprecated("Exceptions are used instead")]] = 4  };  int main(void) {      OperationMode op1 = OperationMode::OM\_Terrified;      OperationMode op2 = OperationMode::OM\_AbortOnError;      cout << op1 << " " << op2 << endl;      return 0;  } |



|  |
| --- |
| namespace [[deprecated]] XYZ {      enum OperationMode {      OM\_Invalid = 1,      OM\_Normal,      };  };  int main(void) {      XYZ::OperationMode op1 = XYZ::OperationMode::OM\_Invalid;      XYZ::OperationMode op2 = XYZ::OperationMode::OM\_Normal;      cout << op1 << " " << op2 << endl;      return 0;  } |



# Nested namespaces

* In older version of C++, if we want to organize classes logically, we would have done the below code…

|  |
| --- |
| namespace MyCompany  {  namespace MyModule  {  namespace MyModulePart  {  namespace MySubModulePart  {  class MyClass {  int x;  int y;  public:  MyClass(){ }  };  }  }  }  } |

* The above syntax looks nasty and horrible within the header file…
* C++17 has introduced ‘nested namespaces’… The same code can be written in C++17 as…

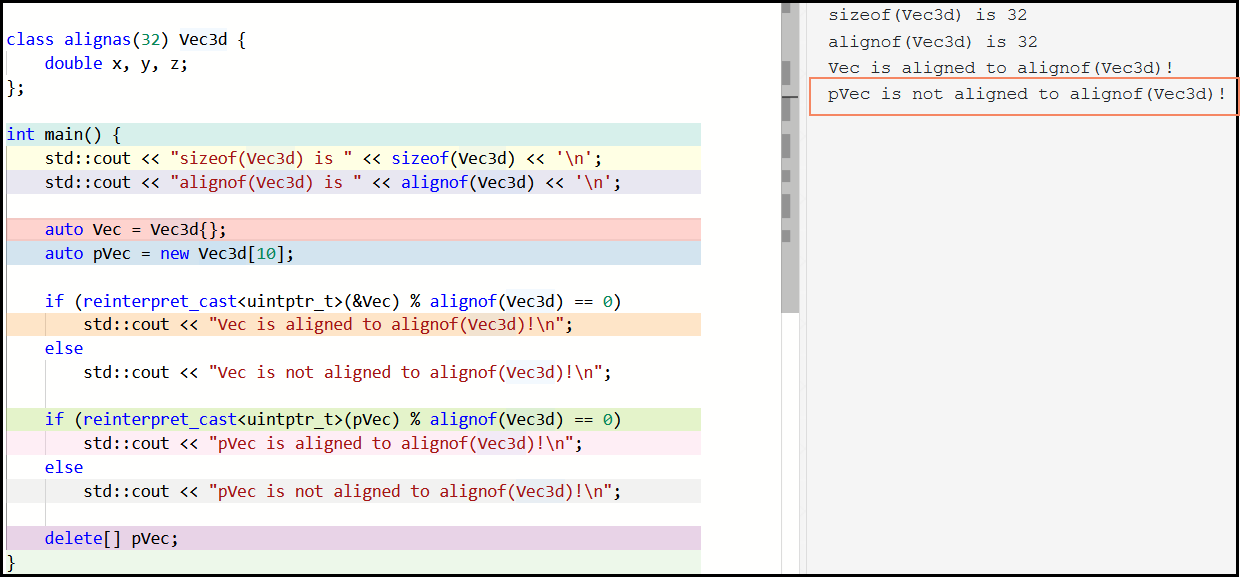
|  |
| --- |
| namespace MyCompany::MyModule::MyModulePart::MySubModulePart {  class MyClass {  ...  };  } // namespace MyCompany::MyModule::MyModulePart::MySubModulePart |

# Heap allocation with alignment

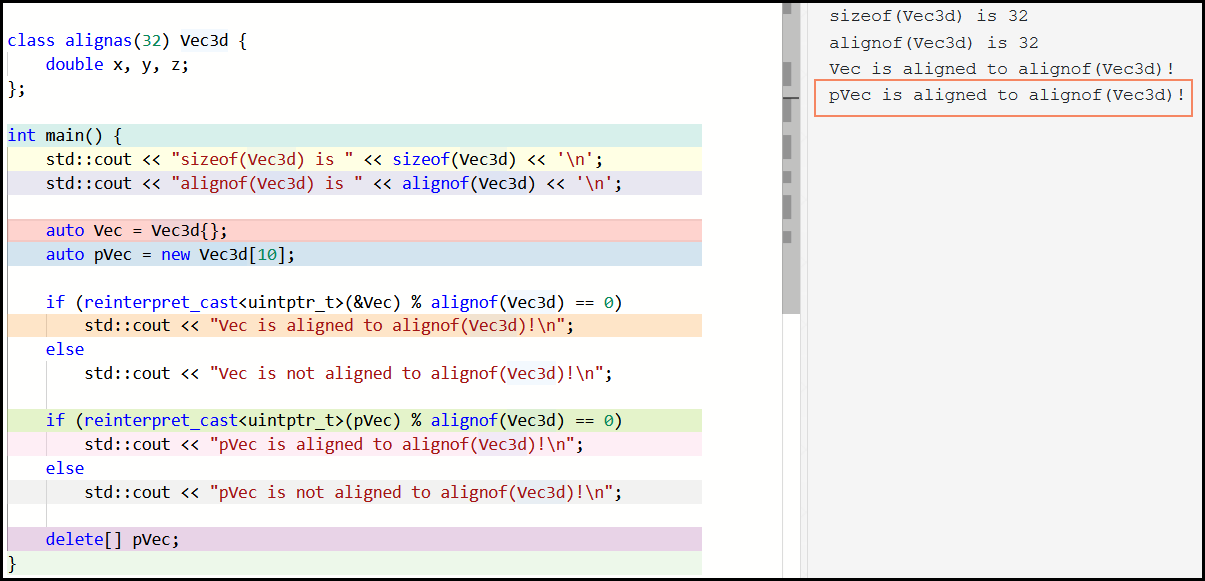
* Let’s see the below code…

|  |
| --- |
| #include <cassert>  #include <cstdint>  #include <iostream>  #include <malloc.h>  #include <new>  class alignas(32) Vec3d {  double x, y, z;  };  int main() {  *std*::*cout* << "sizeof(Vec3d) is " << sizeof(Vec3d) << '\n';  *std*::*cout* << "alignof(Vec3d) is " << alignof(Vec3d) << '\n';  auto Vec = Vec3d{};  auto pVec = new Vec3d[10];  if (reinterpret\_cast<*uintptr\_t*>(&Vec) % alignof(Vec3d) == 0)  *std*::*cout* << "Vec is aligned to alignof(Vec3d)!\n";  else  *std*::*cout* << "Vec is not aligned to alignof(Vec3d)!\n";  if (reinterpret\_cast<*uintptr\_t*>(pVec) % alignof(Vec3d) == 0)  *std*::*cout* << "pVec is aligned to alignof(Vec3d)!\n";  else  *std*::*cout* << "pVec is not aligned to alignof(Vec3d)!\n";  delete[] pVec;  } |

* The code shows a structure - Vec3d that uses three double fields; it also marks the type with alignas that makes the objects aligned to 32 bytes.
* Then the example creates two objects: one on the stack and one on the free store.
* Do they both have the same alignment (32 bytes)?

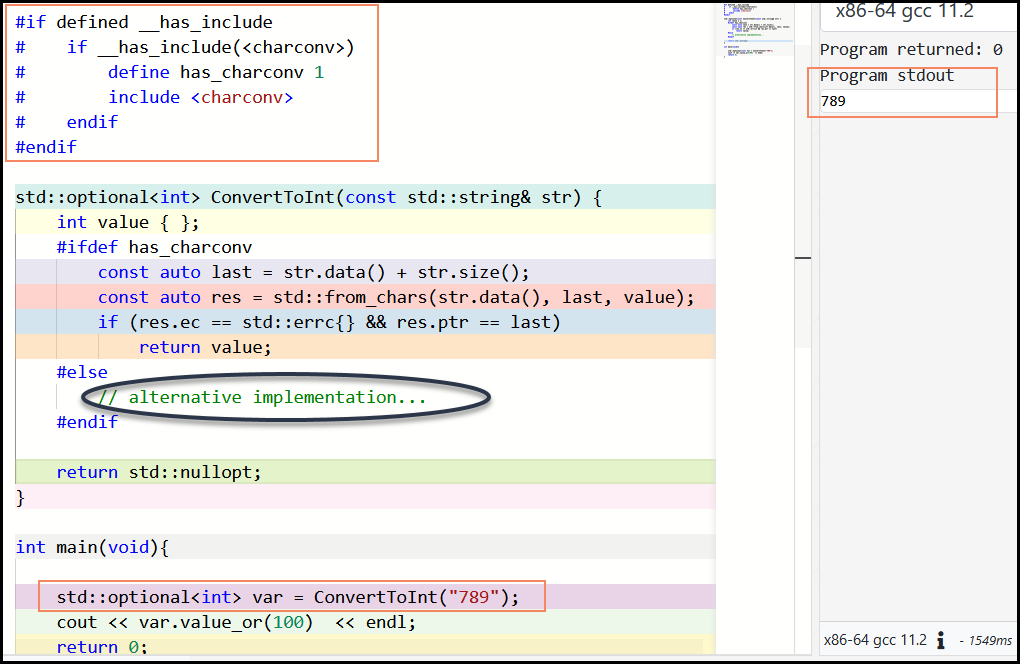


* Now, in the newest standard, we have updated dynamic memory allocations, and now we have a guarantee that the memory will be aligned as requested.
* In C++17, We have now 14 global new() function overloads and 8 class-specific methods! Plus, corresponding delete functions.



# \_\_has\_include Preprocessor Expression

* C++17 offers a preprocessor directive that allows you to check if the header is present or not.
* The special operator \_\_has\_include (operand) may be used in #if and #elif expressions to test whether the header referenced by its operand can be included using the #include directive. Using the operator in other contexts is not valid.



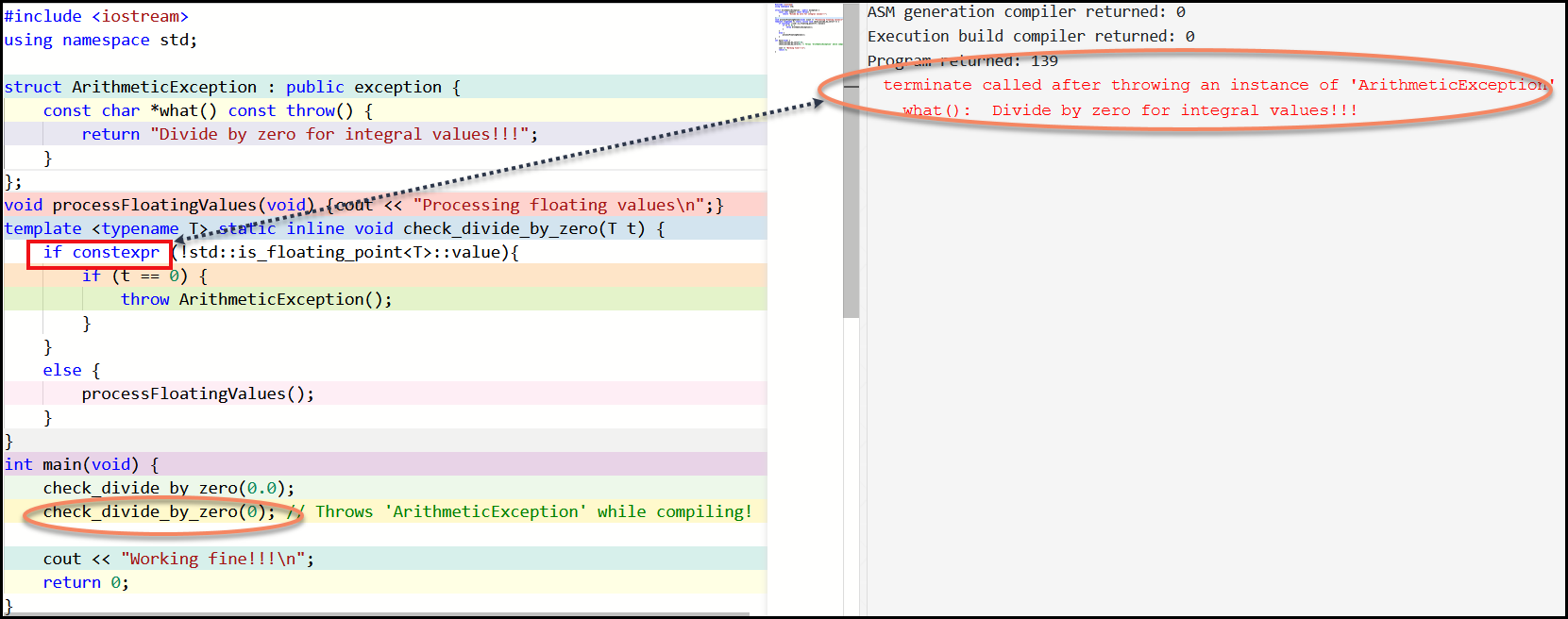
* In the above code, we declare has\_charconv based on the \_\_has\_include condition. If the header is not there, we need to provide an alternative implementation for *ConvertToInt*.

Example: Read the below directives to understand \_\_has\_include.

|  |
| --- |
| #if \_\_has\_include (<optional>)  # include <optiona1>  # define HAS\_OPTIONAL 1  #elif \_\_has\_include(<*experimental*/optional>)  # include <experimental/optional>  # define HAS\_OPTIONAL 1  # define OPTIONAL\_IS\_EXPERIMENTAL 1  #else  # define HAS\_OPTIONAL 0  #endif |

# constexpr if

* C++17 provides a compile-time ‘if’ which is ‘constexpr if’.
* The feature allows you to discard branches of an if statement at compile-time based on a constant expression condition.



|  |
| --- |
| #include <iostream>  using namespace std;  struct ArithmeticException : public exception {      const char \*what() const throw() {          return "Divide by zero for integral values!!!";      }  };  void processFloatingValues(double d) {      cout << 1.0/d <<  endl;  }  template <typename T> static inline void check\_divide\_by\_zero(T t) {      if constexpr (!std::is\_floating\_point<T>::value){          if (t == 0) {              throw ArithmeticException();          }      }      else {          processFloatingValues(t);      }  }  int main(void) {      check\_divide\_by\_zero(0.0);      //check\_divide\_by\_zero(0); // Throws 'ArithmeticException' while compiling!      cout << "Working fine!!!\n";      return 0;  } |

# Fold expressions

Let’s understand the varargs first…

### Variadic Arguments

* A function which takes an arbitrary amount of parameters.
* Inherited from the C varargs construct.
* Runtime feature.
* Denoted by three dots as the last parameter (...).
* Not typesafe!
* Length of parameter list is unknown
* Example: int printf(const char \* format, ...);

### Variadic Templates

* A template which takes an arbitrary amount of arguments.
* New in C++ 11.
* Compile-time feature
  + Typesafe!
  + Length of parameter list is deduced by the compiler.
* Uses a parameter pack.

### Parameter Packs

* Essentially a list of parameters.
* Denoted by three dots after the type of the last parameter (T... args)
* sizeof...(args) may be used to get the pack length at compile time.
* Expands when necessary, but still at compile time.
* May be empty also.

### Fold Expression

* A fold expression is an instruction for the compiler to repeat the application of an operator over a variadic template pack.
* Suppose if we want to write a ‘sum’ function that computes the sum of all its parameters and returns it, How would we implement this function?

|  |
| --- |
| template<typename... Values>  auto sum(Values const&... values){      // code here  } |

* To implement sum with generic code, we can use a fold expression:

|  |
| --- |
| template<typename... Values>  auto sum(Values... values){      return (0 + ... + values); // sum() would fail without 0  }  int main(void) {      cout << "sum(1,2,3,4,5): " << sum(1,2,3,4,5) << endl;      cout << "sum(1,2,3,4,5,6,7,8,9): " << sum(1,2,3,4,5,6,7,8,9) << endl;      cout << "sum(): " << sum(); // This would fail without 0 in the expression!      return 0;  } |

* operator+ is associativity over ints. Where as operator- doesn't have the associative property. Example:

|  |
| --- |
| #include <iostream>  using namespace std;  template<typename... Values>  auto diff1(Values const&... values) {      return (values - ...);  }  template<typename... Values>  auto diff2(Values const&... values) {      return(... - values);  }  int main(void) // Program is in C++  {      cout << "diff1(1,2,3,4,5): " << diff1(1,2,3,4,5) << endl;      cout << "diff2(1,2,3,4,5): " << diff2(1,2,3,4,5) << endl;      return 0;  }  /\*--------------OUTPUT--------------  diff1(1,2,3,4,5): 3  diff2(1,2,3,4,5): -13  -----------------------------------\*/ |

Example 2:

|  |
| --- |
| struct Node {      int value;      Node\* left;      Node\* right;      Node(int i=0) : value(i), left(nullptr), right(nullptr) {}  };  auto left1 = &Node::left;  auto right1 = &Node::right;  template <typename T, typename ... TD >  Node \* traverse(T start, TD ... paths) {      return (start ->\* ... ->\*paths);  }  int main() {      Node\* root = new Node{0};      root->left = new Node{1};      root->left->right = new Node{2};      Node\* node = traverse(root, left1, right1);      cout << node->value << endl; // Prints 2      return 0;  } |

# Template argument deduction for class templates

* Class template parameter types can now be deduced according to arguments passed to the constructor.

Example:

|  |
| --- |
| #include <complex>  #include <iostream>  using namespace std;  int main() {    std::cout << std::complex<int>{5, 3}; // OK, all C++ versions    std::cout << std::complex{5, 3}; // OK since C++17, deduces std::complex<int>    std::cout << std::complex(5, 3); // OK since C++17, deduces std::complex<int>    //std::cout << std::complex(5, 3.3);// Error:args do not have the same type T    return 0;  } |

# Lambda capture this by value

* Capturing this in a lambda's environment was previously reference-only.
* \*this (C++17) will now make a copy of the current object, while this (C++11) continues to capture by reference.

|  |
| --- |
| #include <iostream>  using namespace *std*;  struct MyObj {    int value{ 123 };  // Now we can capture 'this' by value!  auto getValueCopy() {  return[\*this]{ return value; };  }  // C++11 continues to capture by reference.  auto getValueRef() {  return [this] { return value; };  }  };  int main(void) {  MyObj mo;  auto valueCopy = mo.getValueCopy();  auto valueRef = mo.getValueRef();  mo.value = 799;    *cout* << "Captured By Value: " << valueCopy() << *endl*;  *cout* << "Captured By Reference: " << valueRef() << *endl*;  return 0;  }  /\*  Captured By Value: 123  Captured By Reference: 799  \*/ |

# UTF-8 character literals

* We have five encoding-prefixes for string-literals (none, L, u8, u, U)

Example:

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  int main(void) {    //  We have five encoding-prefixes for string-literals (none, L, u8, u, U)    auto s0 = "hello";    // None    auto s1 = L"hello";   // const wchar\_t\*    auto s2 = u8"hello";  // const char\* encoded as UTF-8    auto s3 = u"hello";   // const char16\_t\*, encoded as UTF-16    auto s4 = U"hello";   // const char32\_t\*, encoded as UTF-32    cout << "s0: " << typeid(s0).name() << endl;    cout << "s1: " << typeid(s1).name() << endl;    cout << "s2: " << typeid(s2).name() << endl;    cout << "s3: " << typeid(s3).name() << endl;    cout << "s4: " << typeid(s4).name() << endl;    return 0;  }  /\*  s0: char const \*  s1: wchar\_t const \*  s2: char const \*  s3: char16\_t const \*  s4: char32\_t const \*  \*/ |

* We had only four for character literals. The missing one is u8 for character literals. This has been introduced in C++17.

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  int main(void) {    auto s0 = 'h';    // c - Character    auto s1 = L'h';   // w - Wide character    auto s2 = u8'h';  // This is newly introduced!!!    auto s3 = u'h';   // Ds -    auto s4 = U'h';   // Di    cout << "s0: " << typeid(s0).name() << endl;    cout << "s1: " << typeid(s1).name() << endl;    cout << "s2: " << typeid(s2).name() << endl;    cout << "s3: " << typeid(s3).name() << endl;    cout << "s4: " << typeid(s4).name() << endl;    return 0;  }  /\*  s0: c  s1: w  s2: c  s3: Ds  s4: Di  \*/ |

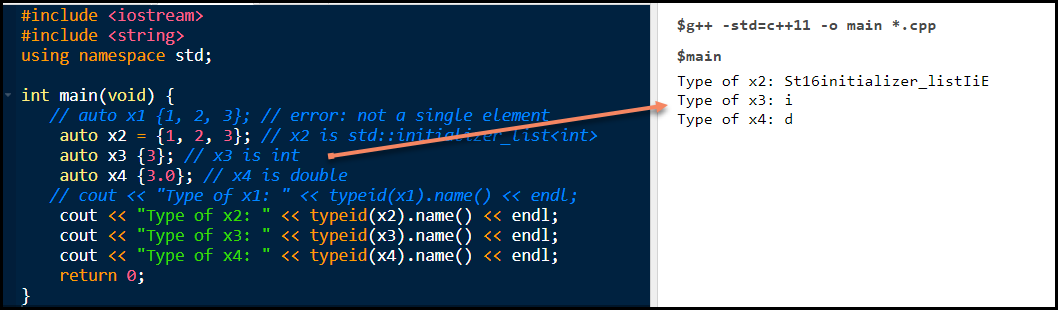
# Direct list initialization of enums

* Enums can now be initialized using braced syntax.

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  enum BYTE : unsigned char {      RED,      YELLOW,      GREEN,  };  int main(void) {    BYTE b {0};           // OK    //BYTE c {-1};        // ERROR: constant expression evaluates to -1                          // which cannot be narrowed to type 'BYTE'    BYTE d = BYTE{1};     // OK    //BYTE e = BYTE{256}; // ERROR: constant expression evaluates to 256                          // which cannot be narrowed to type 'BYTE'    cout << "Value of BYTE b: " << b << endl;    cout << "Value of BYTE d: " << d << endl;    return 0;  } |

# New rules for auto deduction from braced-init-list

* Changes to auto deduction when used with the uniform initialization syntax.
* Previously, auto x {3}; deduces a std::initializer\_list<int>, which now deduces to int.



# Declaring non-type template parameters with auto

### Non-type parameters

1. A template non-type parameter is a template parameter where the type of the parameter is predefined and is substituted for a constexpr value passed in as an argument.
2. A non-type parameter can be any of the following types:
   * An integral type
   * An enumeration type
   * A pointer or reference to a class object
   * A pointer or reference to a function
   * A pointer or reference to a class member function
   * std::nullptr\_t

Example:

|  |
| --- |
| template <typename T, int size> // size is a Non-type parameter |

* Before C++17, we have to specify the type for Non-type parameter. In the above example, we have specified it as int.
* From C++17, while respecting the non-type template parameter list of allowable types[\*], template arguments can be deduced from the types of its arguments:
  + \* - For example, you cannot use a double as a template parameter type, which also makes this an invalid deduction using auto.
* In the below example, we have now used ‘auto’ instead of ‘int’ for Non-type parameter.

|  |
| --- |
| #include <iostream>  using namespace std;  template <typename T, auto size> // size is a Non-type parameter  class StaticArray {  private:      // The non-type parameter controls the size of the array.      T m\_array[size] {};  public:      T\* getArray();      T& operator[](int index) {          return m\_array[index];      }  };  // This is how a function for a class with a non-type parameter  // is defined outside of the class.  template <typename T, auto size>  T\* StaticArray<T, size>::getArray() {      return m\_array;  }  int main() {      // Declare an integer array with room for 12 integers      StaticArray<int, 12> intArray;      // Fill it up in order, then print it backwards      for (int count { 0 }; count < 12; ++count)          intArray[count] = count;      for (int count { 11 }; count >= 0; --count)          std::cout << intArray[count] << ' ';      std::cout << '\n';      // Declare a double buffer with room for 4 doubles      StaticArray<double, 4> doubleArray;      for (int count { 0 }; count < 4; ++count)          doubleArray[count] = 4.4 + 0.1 \* count;      for (int count { 0 }; count < 4; ++count)          std::cout << doubleArray[count] << ' ';      //StaticArray<double, 4.5> doubleArray1; // Error!!!      // Double cannot be passed as a Non-type parameter!!!      // See Point Number 2 above.      return 0;  } |