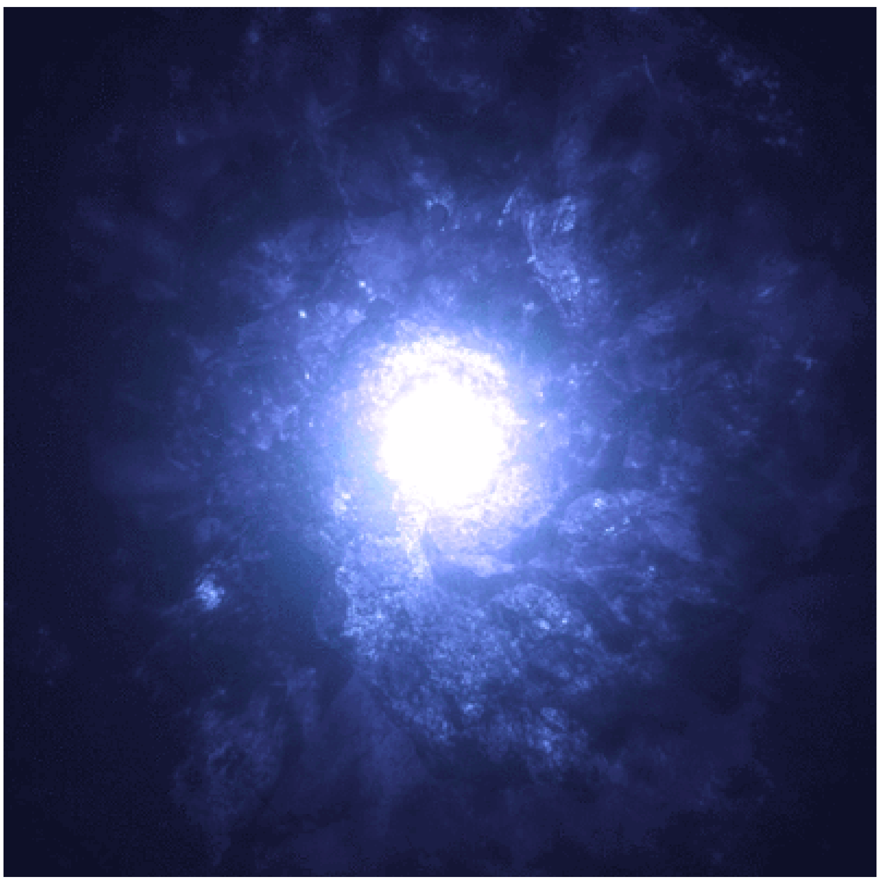
Adrian D. Finlay
@thewiprogrammer. Writer @hackernoon. Code, LOTS of it. Mangos, LOVE THEM! Barbering. Health. Travel. Business. & more! Network w/ me @ adriandavid.me/network
Nov 10, 2017 · 17 min read

Modern C++ Part III: New Language Features in C++17

The never ending journey into learning C++ features….



Giphy: FELIKS TOMASZ KONCZAKOWSKI GIF

C++ is a general purpose, multi-paradigm, compiled language that was invented by danish computer scientist, Bjarne Stroustroup, and released in 1983. C++ marries classes, such as those found in Simula, with the majority of the C language, to create a language that is like an Object Oriented version of the C language. However, C++ is dar divorced from the days of "C with classes". Almost all C++ compiler vendors provide an implementation of a the standard class library, libstdc++. C++ is among the most widely implemented and widely used languages in the history of modern computing.

In my opinion, it is best suited for **systems programming, embedded programming, high performance computing, resource constrained computing** (think tiny devices), & the **development of low level APIs, language compilers, interpreters, device drivers, & the design of software infrastructure.**

Typically, the choice to use C++ is predicated by the need for performance & efficiency (little bloat, efficient use of resources and implementation constructs, getting as close to the metal as possible). For better or worse, C++ is ideologically flexible—it does not constrain you to programming in one paradigm such as many other languages. It contains a bevy of features, which is a frequent source of criticism by certain members of the programming community. C++ is an outlier of sorts in that the philosophy behind C++ embraces including good ideas from many different ideological perspectives as opposed to the KISS (*Keep It Simple Stupid)* philosophy which is more oriented towards having one simple way to do one thing.

C++ is also used for common Desktop Application Software. C++ has found widespread use in truly massive systems. For example: Google Chrome, Mozilla Firefox, Telephony Infrastructure, Chrome V8, and much, much more. Read Stroustrup's (incomplete) list here. C++ might be a better decision than using C for many reasons, the most popular of which, in my opinion, are the various abstractions that C++ provides, most notably, the class. The class allows for highly structured programs that bind data and the functions that act on such data. This often makes for more organized programs than the C equivalent.

The upcoming revision to the ISO for standard is C++17. It will ship with **35** new language features and **4** deprecated language features (**do let me know if I am wrong on any of this**). I will group some of the related features together. The industry leading compilers (GCC, MSVC, Clang) have already implemented many of the new C++17 features ahead of it's general release [3]. The list of new features presented in this article has been generated from several sources [1][3][4][5][6][7]. C++ is expected to ship sometime this year (2017) [7]. You may track it’s current ISO approval status, here. C++17 is a major feature release, the largest such release since C++11.

Update: ISO Specification for C++17 is now published, work is being done on C++20.

New Language Features

1. Addition of __has_include macro
2. UTF 8 Character Literals
3. Hexadecimal Floating Point Literals
4. New rules for deduction of single member list using auto
5. Update to __cplusplus value
6. inline variables
7. New Syntax for Nested Namespace definitions
8. Initializers added to if/switch statements
9. constexpr if
10. New standard attributes [[fallthrough]], [[maybe_unused]] & [[nodiscard]] ^
11. Attributes for Enumerator & Namespaces
12. Error message for static_assert now optional
13. Structured binding declarations
14. Keyword typename now allowed in lieu of class in a template’s template paramater
15. Constant evaluation for non-type template arguments
16. Class template argument deduction
17. Extensions on over-aligned Memory Allocation
18. Fold expressions
19. List-style Initialization of Enumerations
20. Specifying non-type template parameters with auto

21. constexpr lambda expressions
22. Lambda this by value (*this)
23. Extending Aggregate Initialization to Base Types
24. Unknown Attributes Required to be Ignored
25. Pack Expansions legal in using declarations
26. Generalization of Range-based for loop
27. The byte data type ^^
28. Using attribute namespaces without repetition
29. Stricter Order of Evaluation Rules
30. Exception Specifications are part of type definitions
31. Template-Template Parameters match compatible arguments
32. Guaranteed Copy Elision
33. Changes to Specification on Inheriting Constructors

^These are three features grouped into one, which consequently when expanded would make the new feature list count 35.

^^This is implemented in std::byte (<cstddef>) and is not a part of the actual language such as the other primitive data types. It is considered a basic type inasmuch that std::string is considered a basic type.

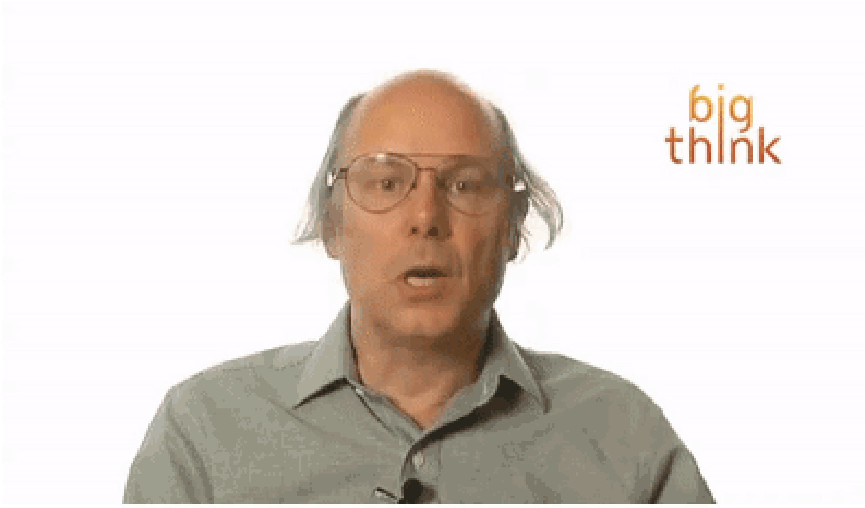
C++17 also introduced a revision to Elementary string conversions which you can read about here.

Deprecated Language Features

1. Removal of Trigraphs by default
2. Removal of deprecated Increment Operator (++) for bool type
3. Removal of deprecated register keyword
4. Removal of deprecated Dynamic Exception Specifications

Check with this list often to see compiler support for the various language changes! [3] You can find information about GCC C++17 support here, and LLVM/Clang C++17 Support here. Please note that I will be covering C++ language features only and **will NOT discuss the several changes to the standard library**.

Why so many features Bjarne? Maybe one day he'll tell me.



GfyCat: "Stroustrup" (Taken from BigThink YouTube Video)

Want the source? Grab it here.

afinlay5/Cplusplus17

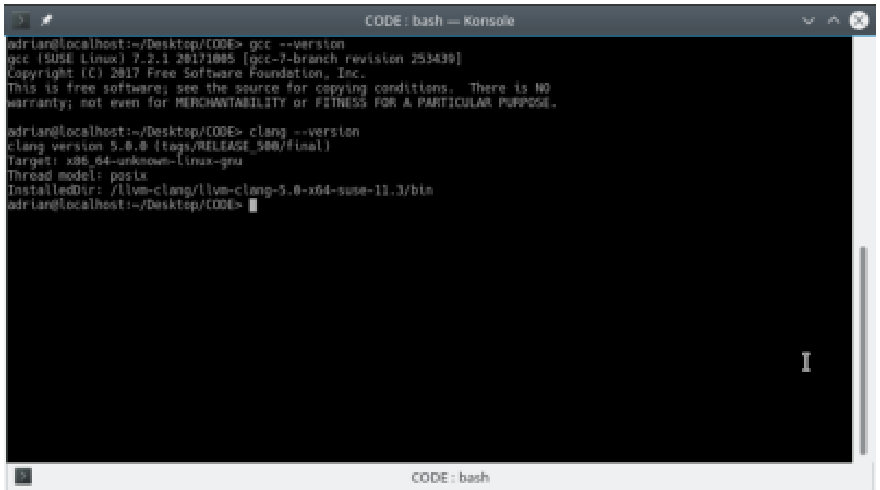
CMake source code repository for C++17 source code examples posted on personal blog…

github.com

The Compilers I'll be using

I will be using GCC version 7.2.1 and Clang (LLVM) version 5.0.0, both the latest versions of the respective compilers as of this publications posting, to test and run my examples. **A testament to C++'s breadth, both of these compilers are themselves written in C++!** Both compilers are part of suites of tools providing **compiler support for**

several different languages on several different architectures. I will be compiling and running the code on bash on my SUSE Linux box.

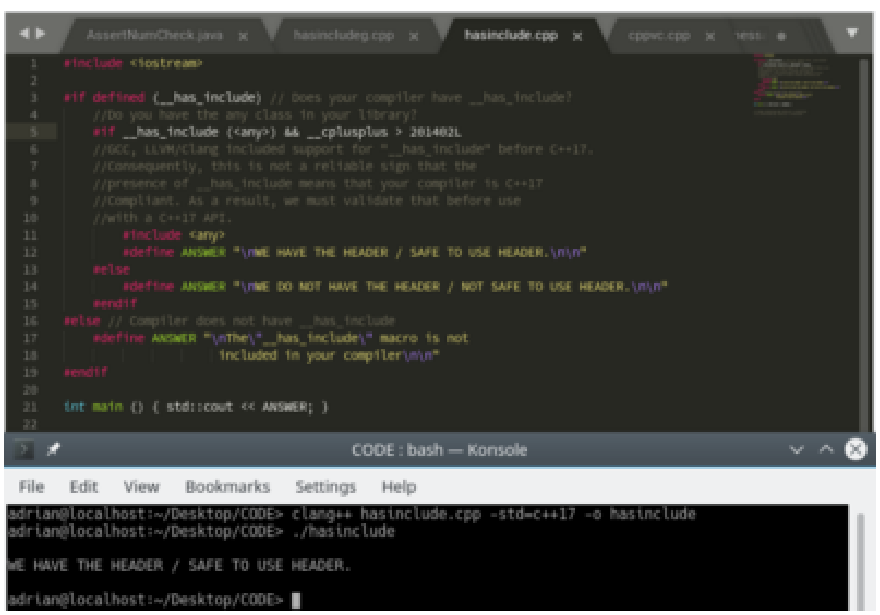


I'll post the code first and the output in bash second. Let's start with the New Language Features.

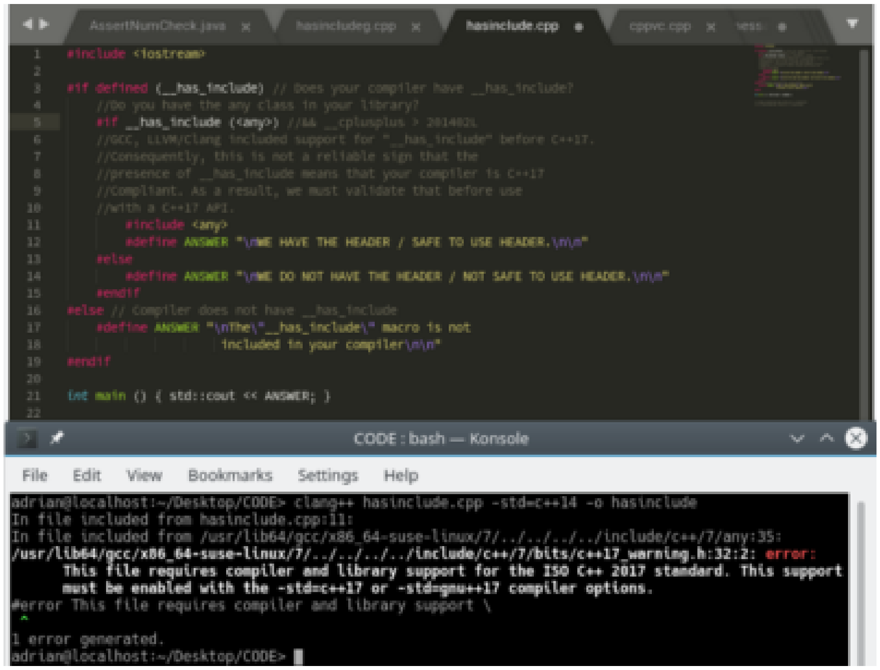
New Language Features

1) Addition of __has_include macro

The macro **__has_include** (added in C++17) allows the programmer to check the availability of a header to be checked by a preprocessor directive.



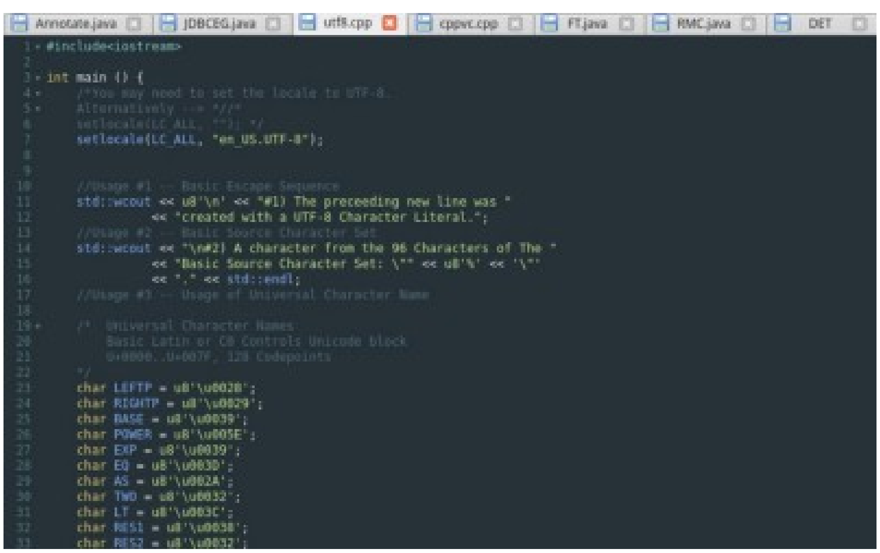
Notice the comments from Ln. 5–10. We must check for C++17 support. For example, std::any is only available in C++17. Otherwise this will happen:

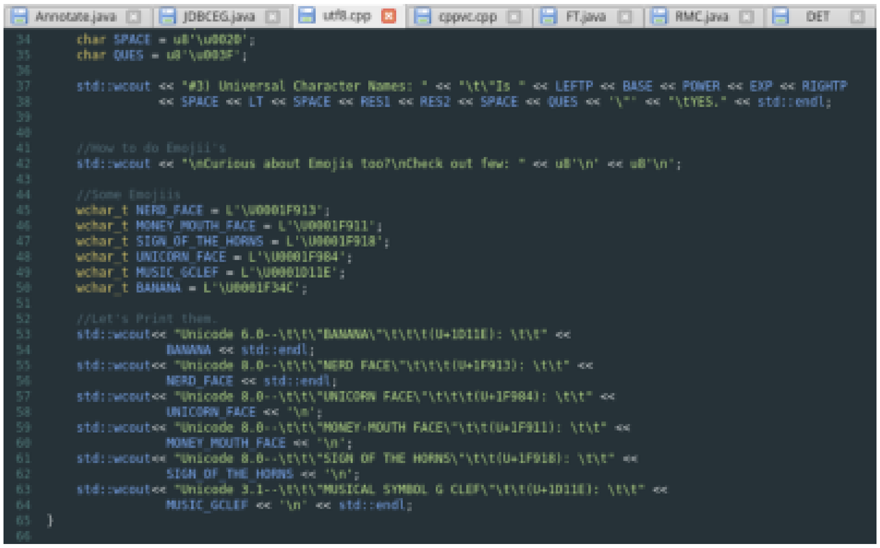


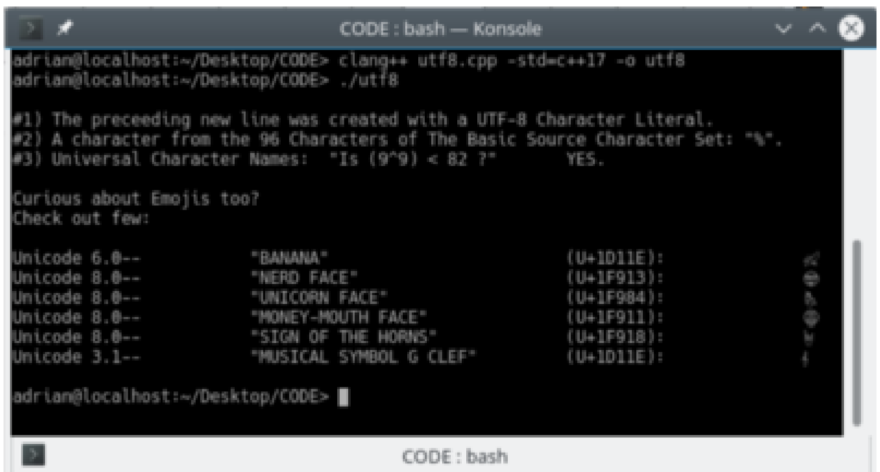
This works with both LLVM/Clang as well as GCC.

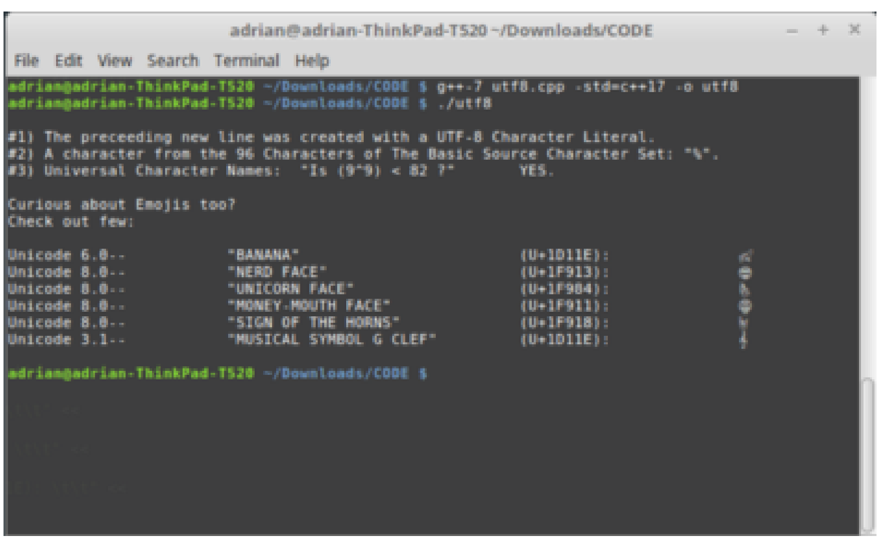
2) UTF 8 Character Literals

C++17 introduced UTF-8 Character Literals.





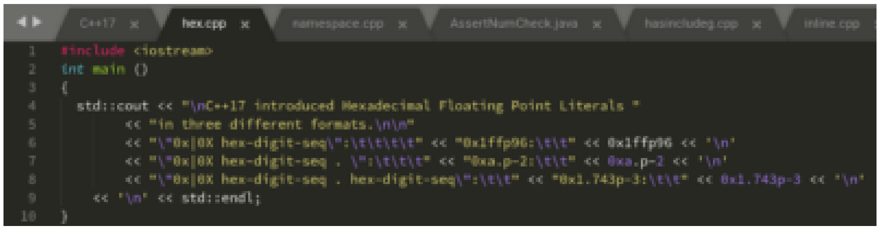


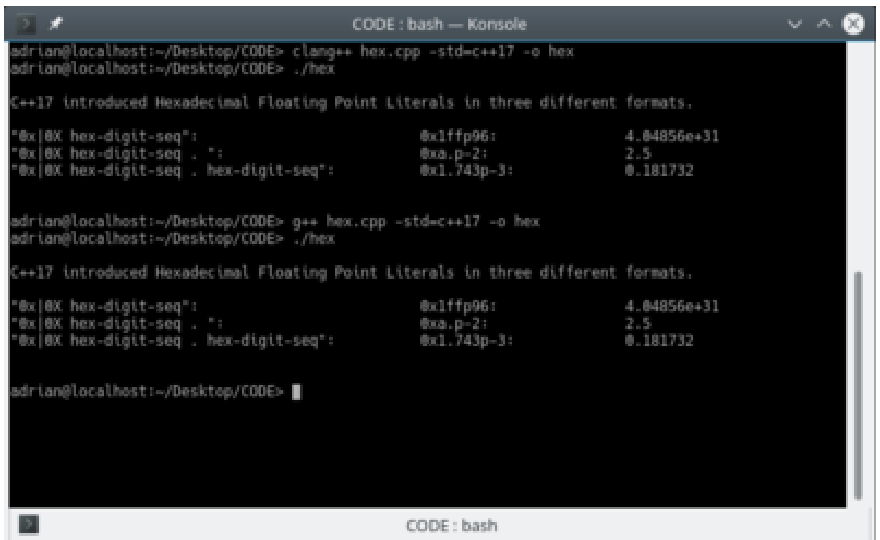


From Linux Mint.

3) Hexadecimal Floating Point Literals

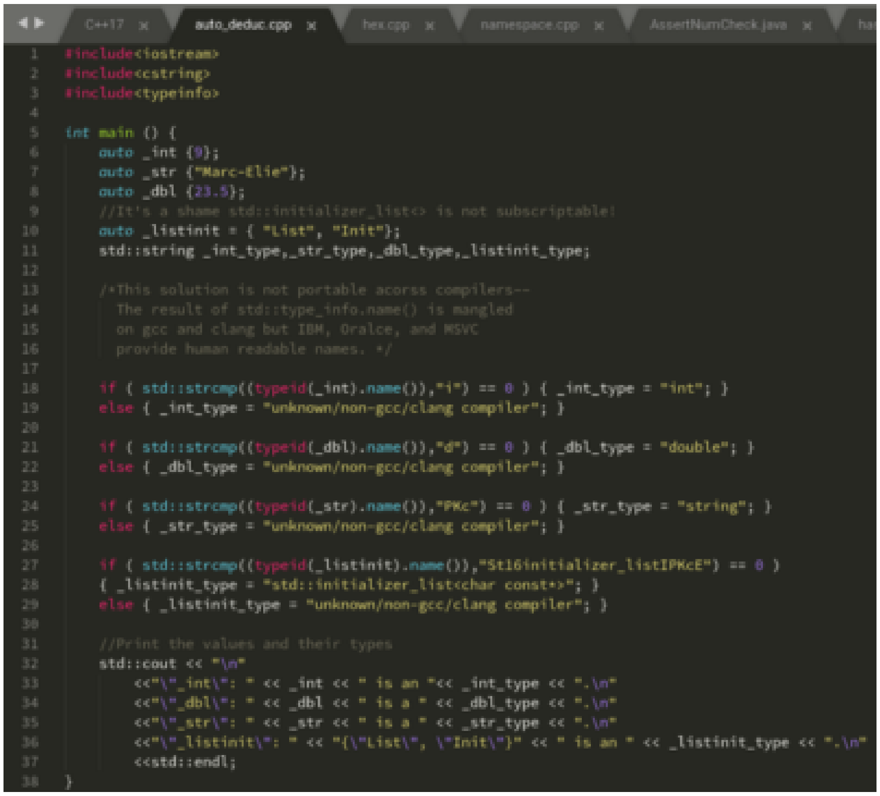
C++17 introduced support for Hexadecimal Floating Point Literals. You may find out more, here.

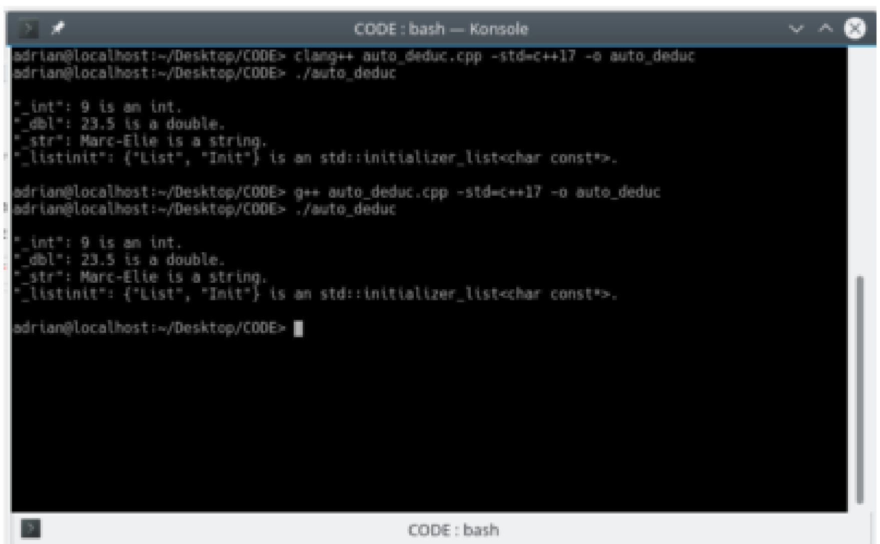




4) New rules for deduction of single member list using auto

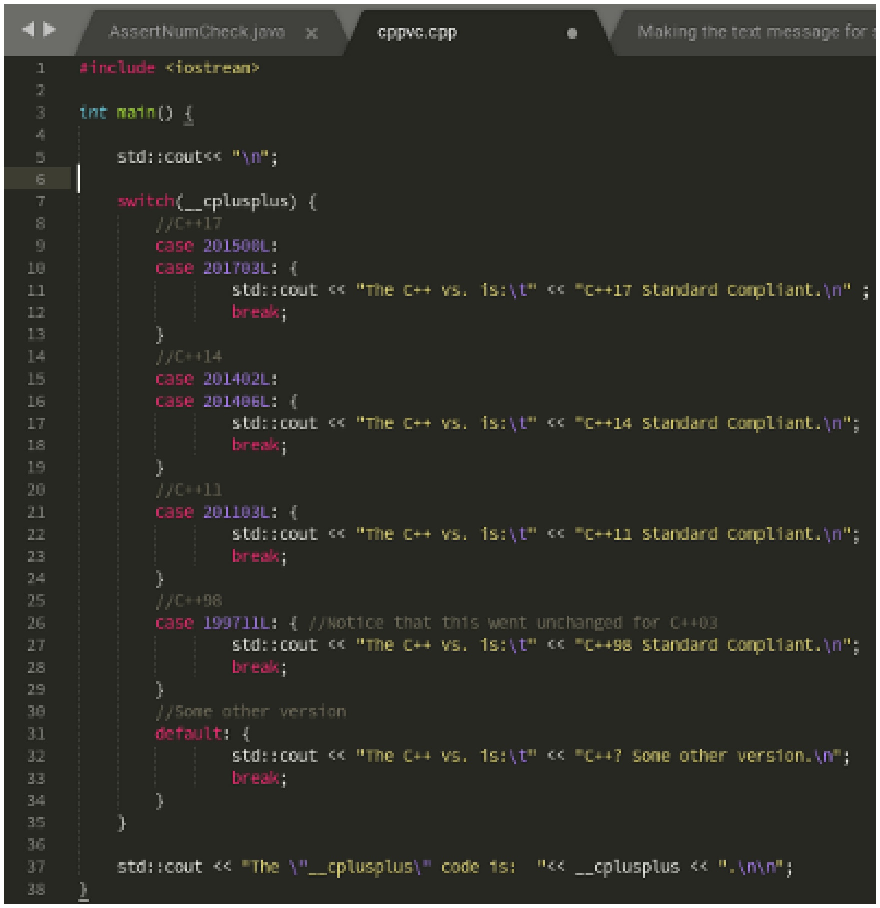
C++17 introduced a more common sense deduction of types using auto for single member lists. Previously, the deduction of single member lists evaluated to std::initializer_list<x> where x was the actual type originally in the list. In C++17 this is more intuitively deduced directly to x for single member lists.

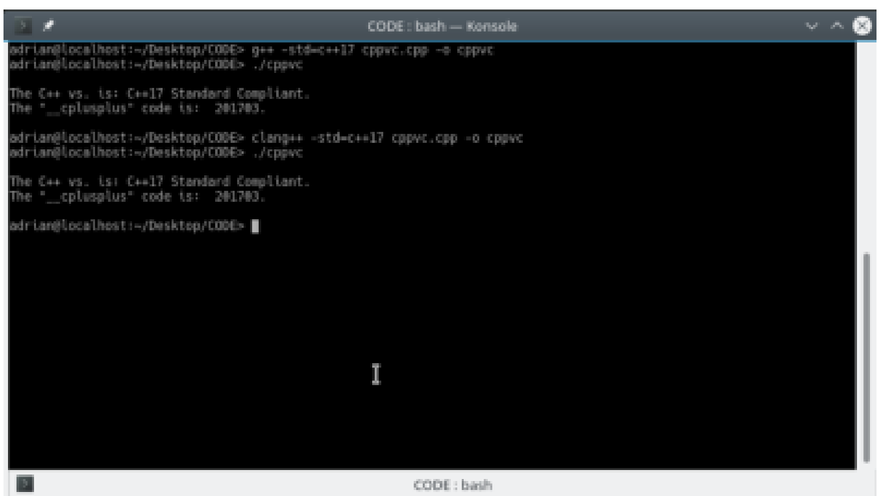




5) Update to __cplusplus value

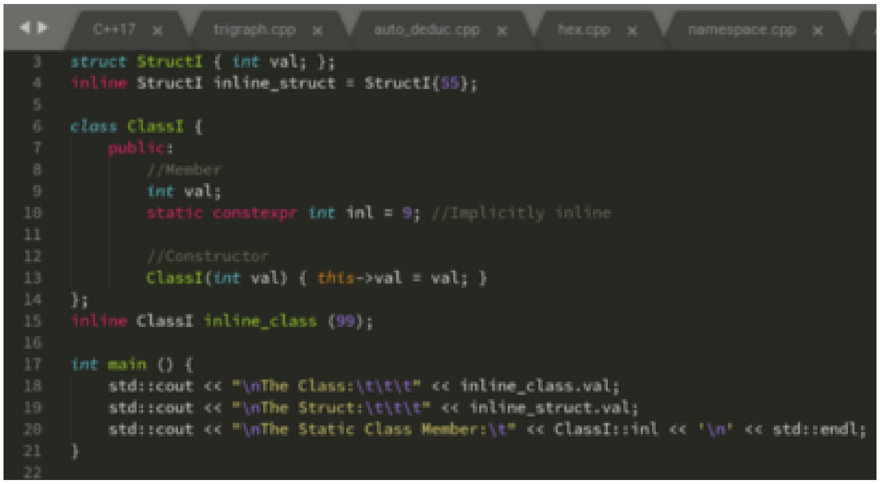
The value of the predefined MACRO __cplusplus has chanaged to 201703L reflecting the update in the language standard.



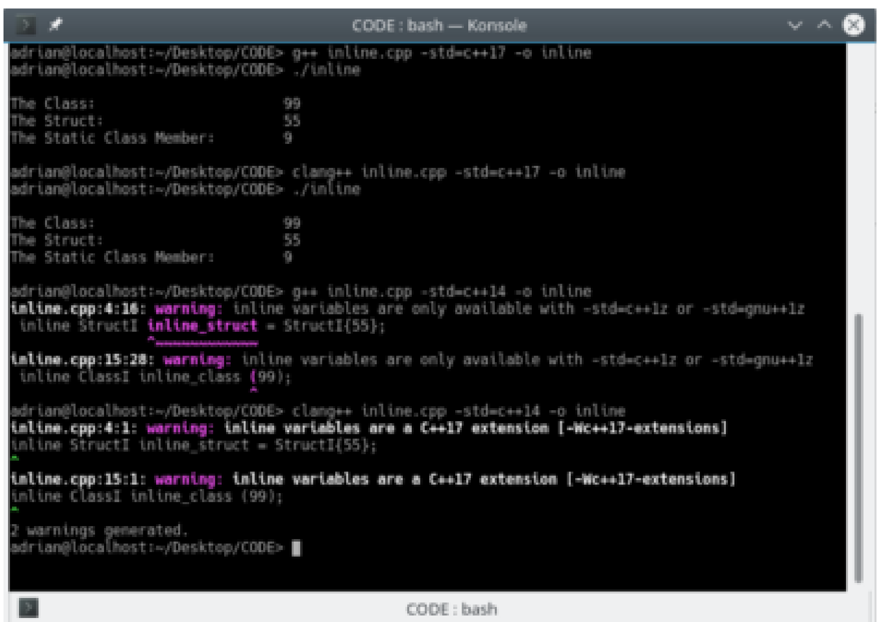


6) Inline variables

C++17 introduced additional functionality to the inline specifier. You may now use the inline specifier with static class members or namespace-scope variables to declare the variable as an inline variable. Also, a static class member variable (not a namespace -scope variable) marked constexpr is implicitly an inline variable [8] . This functionality was previously only available for functions. You will want to read more about this here.



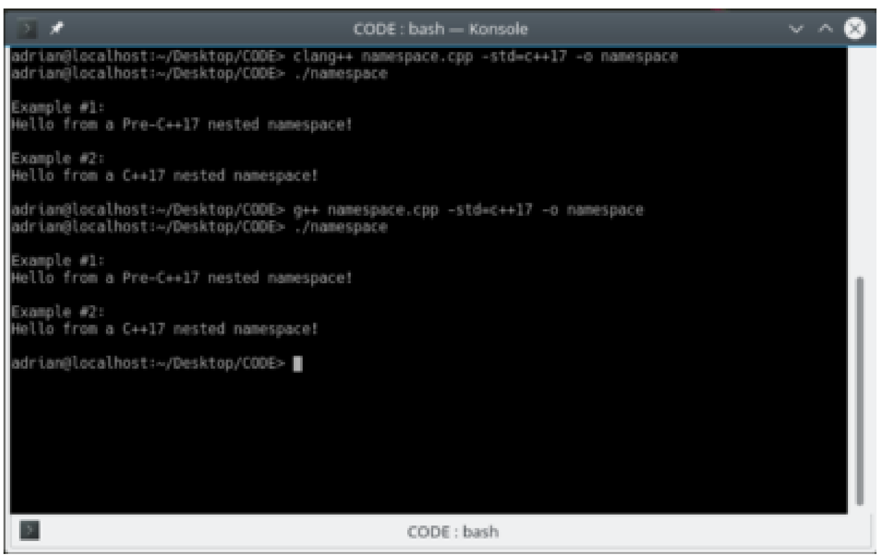
Notice the warnings with C++14.



7) New Syntax for Nested Namespace definitions

C++17 introduced the use of the scope resolution operator to create nested namespaces. This makes for much less verbose code.

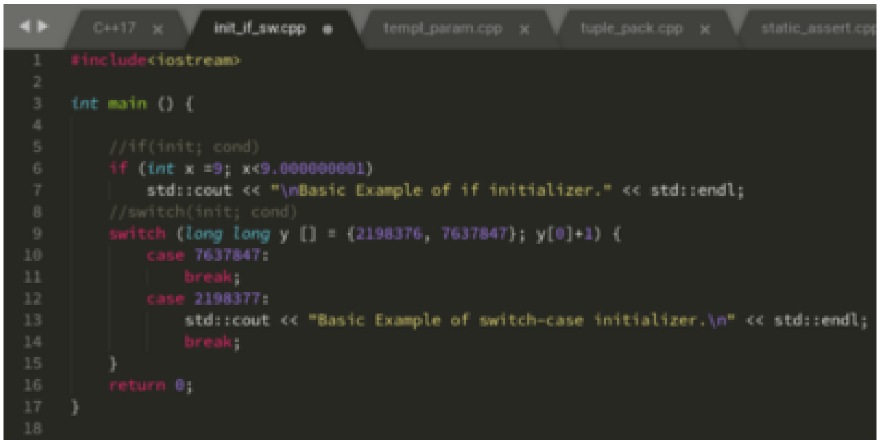
```
#include <iostream>

namespace A {
    namespace B {
        namespace C {
            void foo() {
                std::cout << "\nHello from a Pre-C++17 nested " <<
                "namespace!\n" << std::endl;
            }
        }
    }
}

namespace D::E::F {
    void foo2() {
        std::cout << "Hello from a C++17 nested namespace!" <<
        '\n' << std::endl;
    }
}

int main () {
    std::cout << "\nExample #1:";
    A::B::C::foo();
    std::cout << "Example #2:\n";
    D::E::F::foo2();
}
```

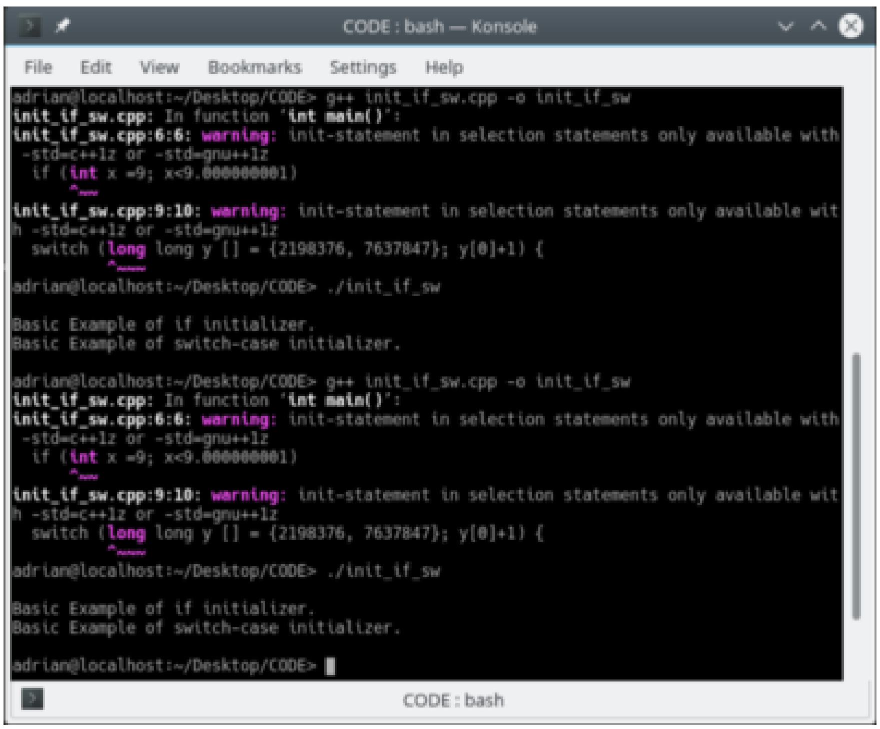


```
adrian@localhost:~/Desktop/CODE> clang++ namespace.cpp -std=c++17 -o namespace
adrian@localhost:~/Desktop/CODE> ./namespace

Example #1:
Hello from a Pre-C++17 nested namespace!

Example #2:
Hello from a C++17 nested namespace!

adrian@localhost:~/Desktop/CODE> g++ namespace.cpp -std=c++17 -o namespace
adrian@localhost:~/Desktop/CODE> ./namespace

Example #1:
Hello from a Pre-C++17 nested namespace!

Example #2:
Hello from a C++17 nested namespace!

adrian@localhost:~/Desktop/CODE>
```

8) Initializers added to if/switch statements

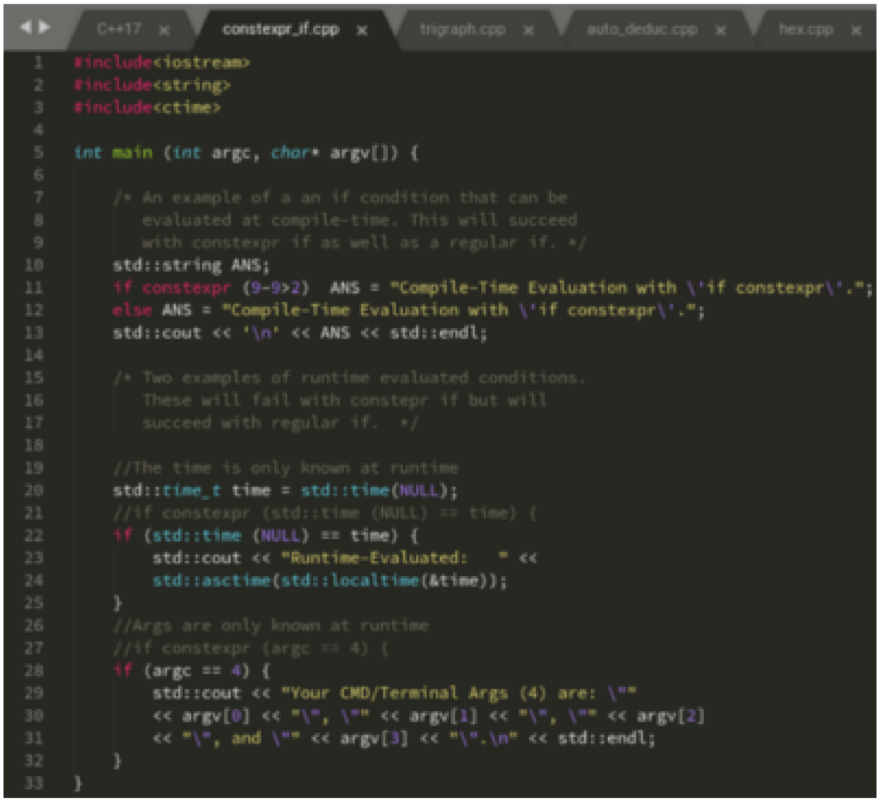
C++17 introduced for initializers in if and switch statements. This is allows for more concise syntax for common coding activities such as initializing a value outside of an if statement. Often, what we really want is for the variable to be local to the if statement or switch-case statement. Initializers solve this design issue.



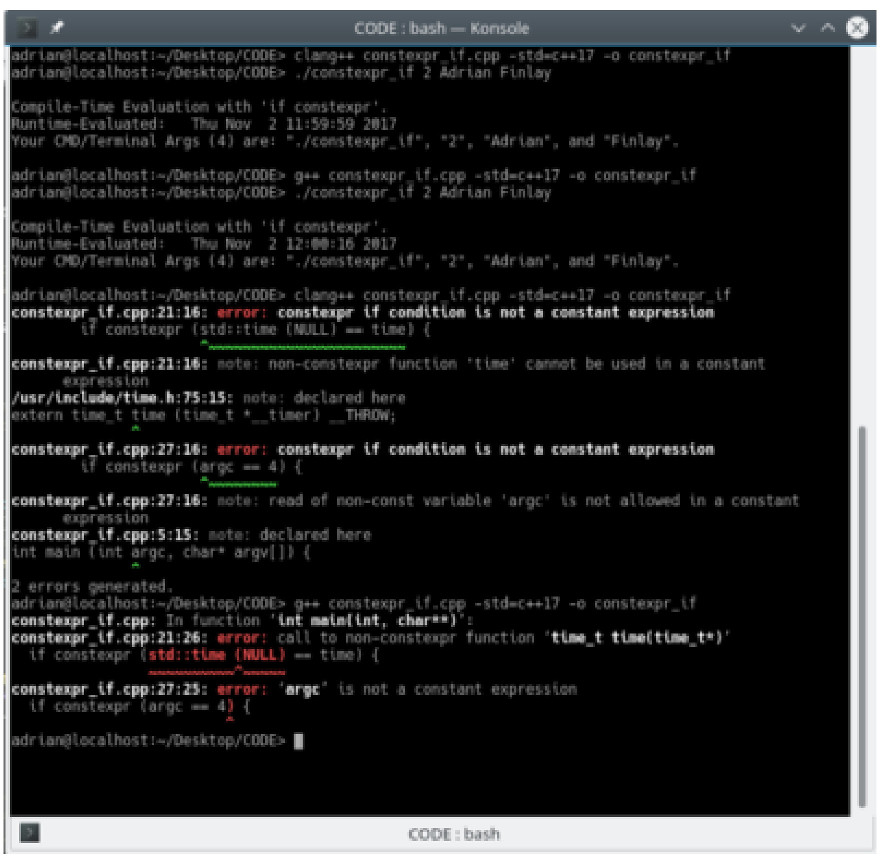


9) constexpr if

C++17 introduced **constexpr if** statements. This allows for explicit compile time evaluation of the if condition. A list of constant expressions are available here.



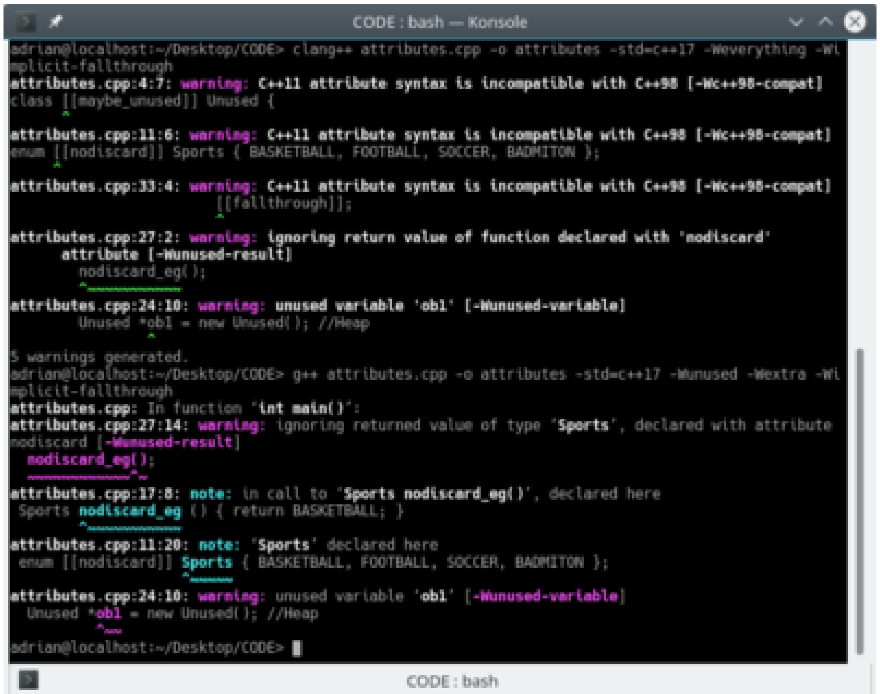
Notice the failure if we use **if constexpr**:



10) New standard attributes [[fallthrough]], [[maybe_unused]] & [[nodiscard]]

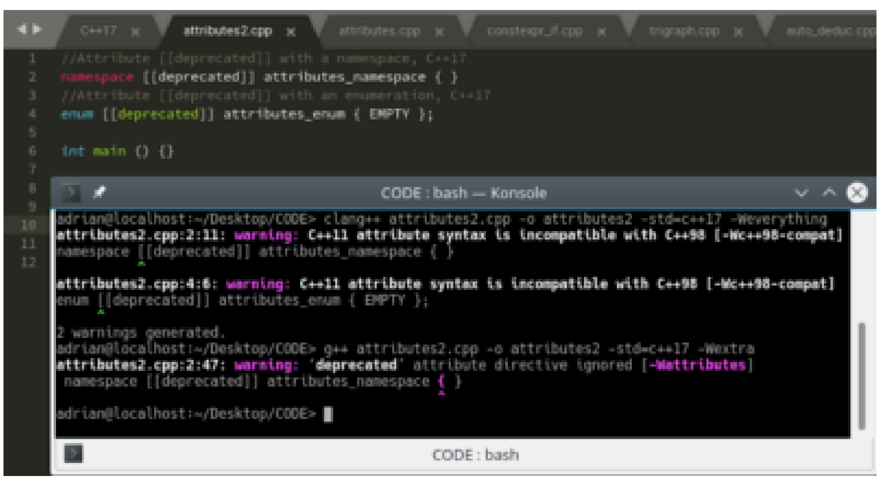
C++17 introduced three new standard attributes. You will see a demonstration of their use below. A list of the standard attributes, including an explanation of the three new standard attributes are available here.

```
#include <iostream>

// [[maybe_unused]]
class [[maybe_unused]] Unused {
    public:
        Unused () {
            std::cout <<"Unsued object" << std::endl;
        }
};
// [[nodiscard]]
enum [[nodiscard]] Sports { BASKETBALL, FOOTBALL, SOCCER, BADMITON };

//Function Prototype
Sports nodiscard_eg ();

// [[nodiscard]]
Sports nodiscard_eg () { return BASKETBALL; }

//main()
int main () {
    std::cout << std::endl;
    // [[maybe_unused]]
    Unused ob2; //Stack
    Unused *ob1 = new Unused(); //Heap

    // [[nodiscard]]
    nodiscard_eg();

    //[[fallthrough]]
    switch (1) {
        case 1:
            std::cout << "This fall through is intentional.\n" << std::endl;
            [[fallthrough]];
        case 2:
            break;
    }
}
```



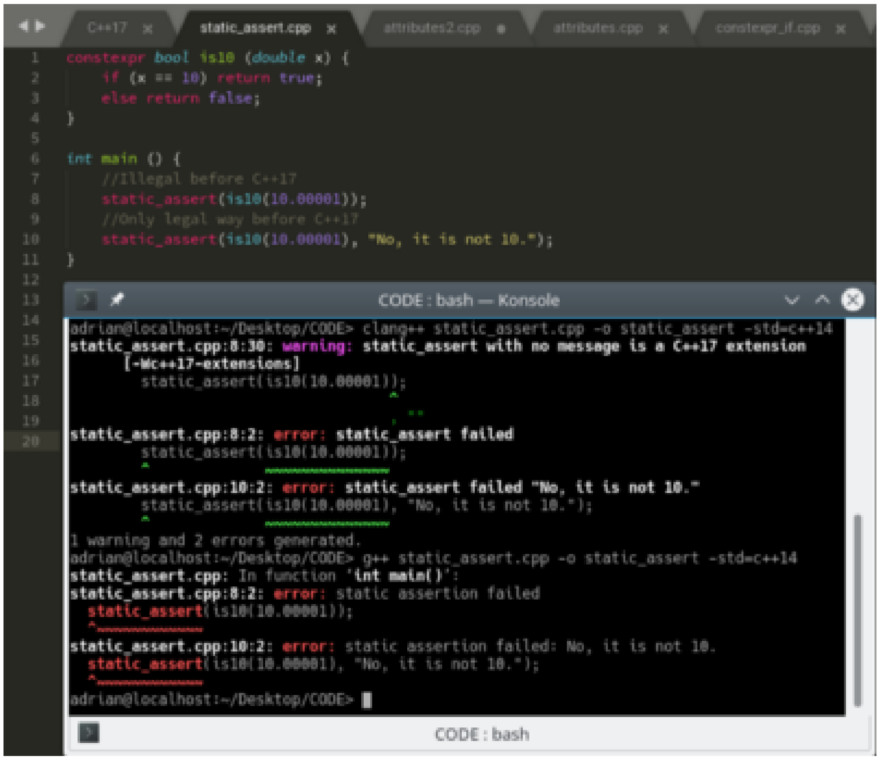
11) Attributes for Enumerator & Namespaces

C++17 introduced support for support for Attributes on Enumerators & Namespaces, which were formerly illegal.



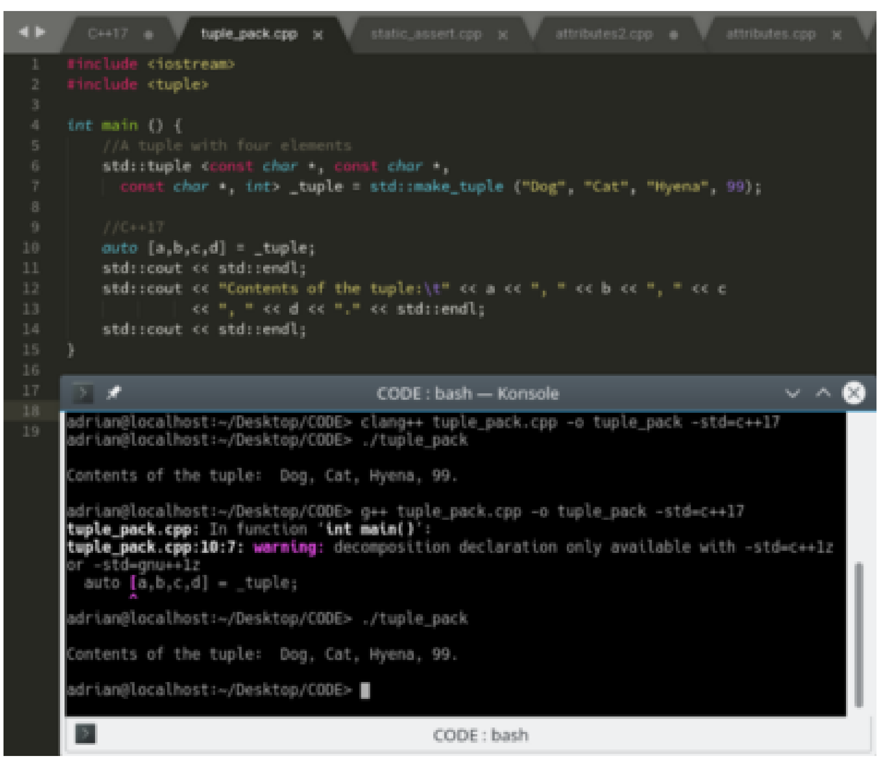
12) Error message for static_assert now optional

With C++17, the error message in the keyword **static_assert** is now optional. Notice the warning in C++14.



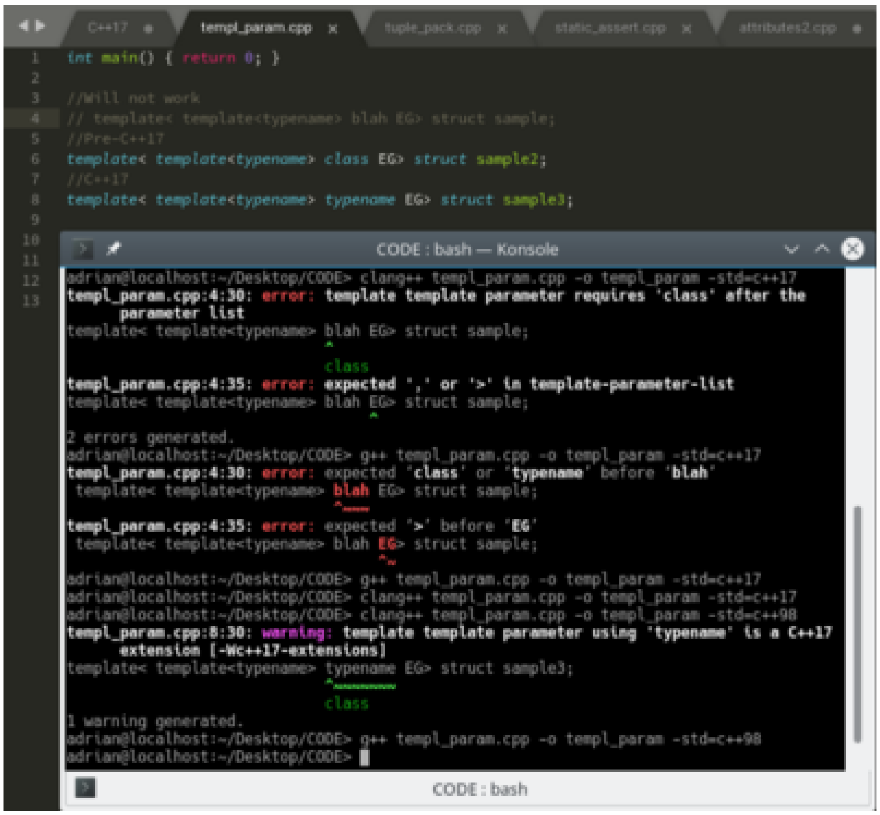
13) Structured binding declarations

C++17 introduced initialization by deconstruction of a tuple like object with auto. The values are bound to the original object.



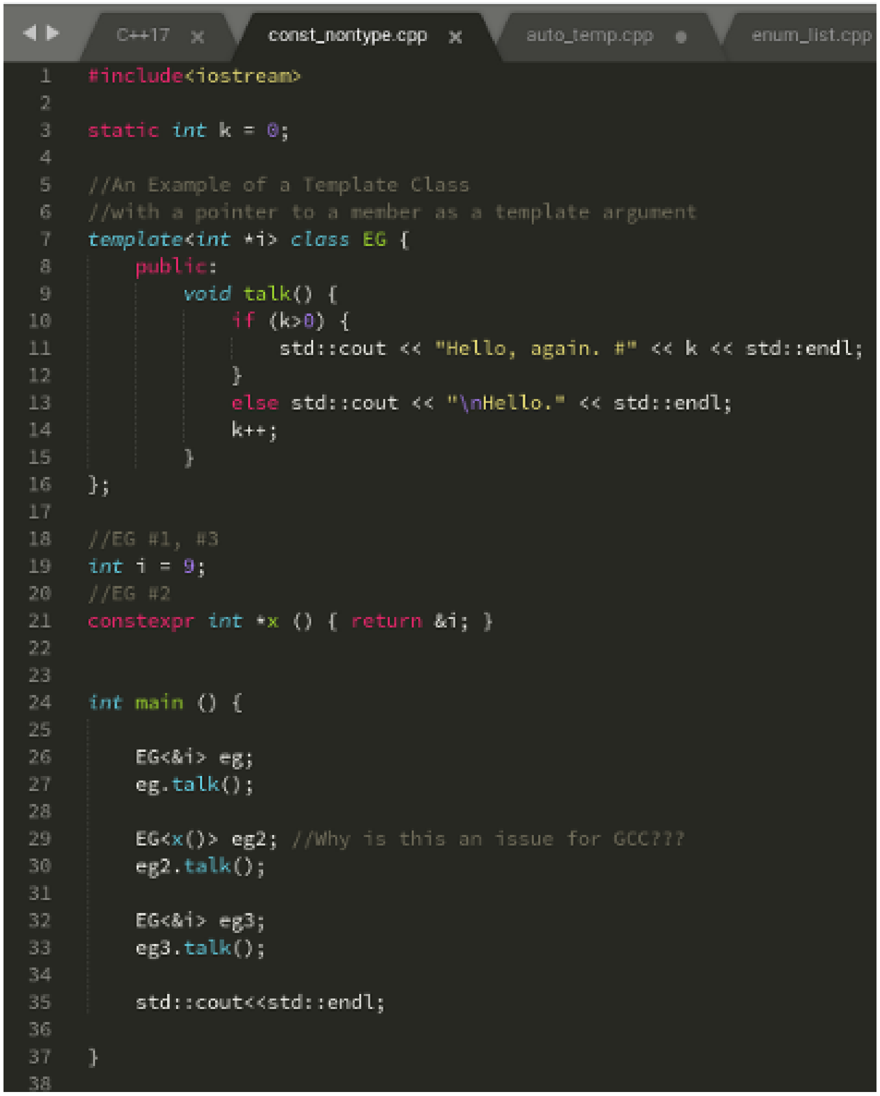
14) Keyword typename now allowed in lieu of class in a template's template paramater

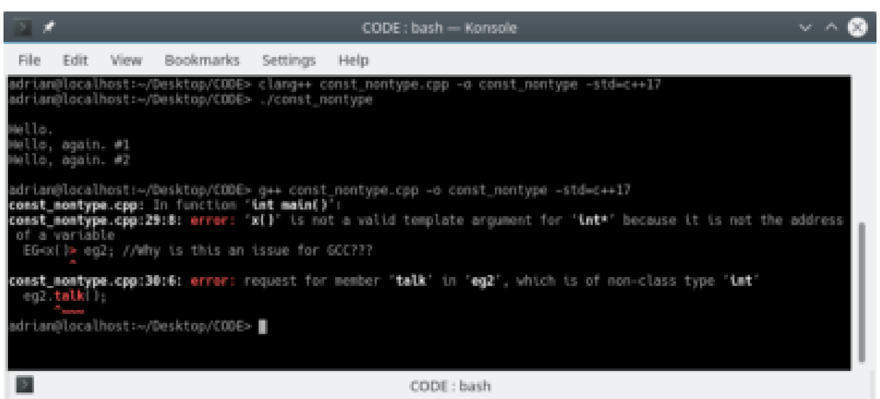
C++17 now allows the use of the keyword typename in lieu of class in a template's template parameter. Curiously enough, while Clang, by default, will warn you about the potential illegal use of the keyword template in the aforementioned situation in C++98, GCC does not does not do so by default.



15) Constant evaluation for non-type template arguments

C++17 now allows constant evaluation for non-type template arguments. You should read more about this, here.





For some reason GCC does not play well when I call the function within the template parameter declaration. However it works on other compilers, and clang has no problem with it. I tried many things (providing a public, default constructor/destructor) to no avail. GCC is reputed as being a strict compiler. If you can figure out the issue, please inform me in the comments.

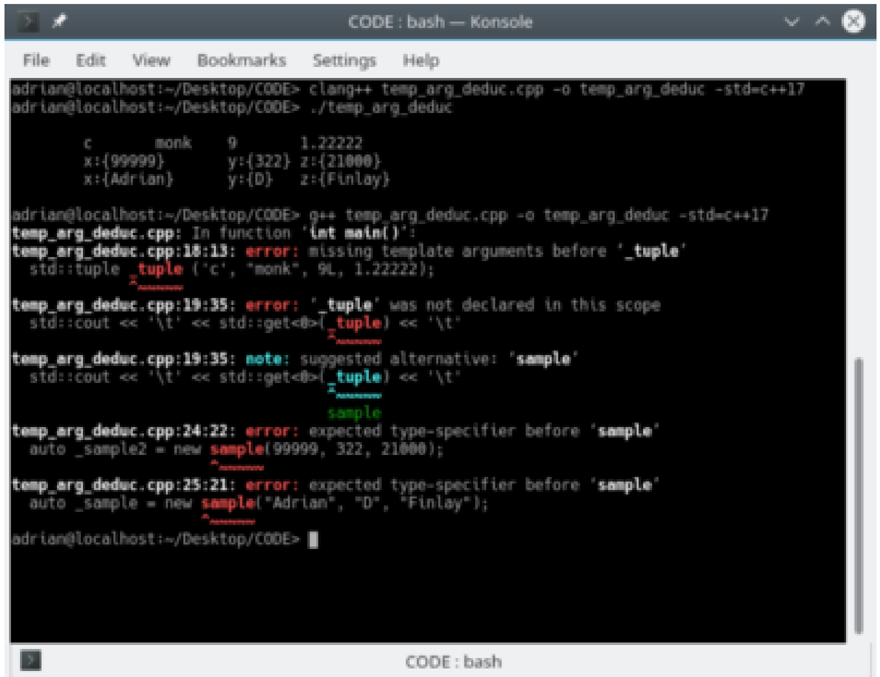
UPDATE: Many thanks to the people who have tested the code with GCC, including @Ciel who ran it on SUSE Linux with GCC 7.2.1. It turns out the issue may be local to my installation.

16) Class template argument deduction

C++17 introduces argument deduction for class template instantiation. Users of languages such as Java, C# may find this familiar. You will want to read more about this feature here. Some basic usage is demonstrated below.

C++17 | temp_arg_deduc.cpp | const_nontype.cpp | init_if_

```cpp
#include<iostream>
#include<tuple>

template<typename Tn> class sample {
    public:
        sample (Tn x, Tn y, Tn z) {
            std::cout << "\tx:{" << x << "}\t"
                << "y:{" << y << "}\t" << "z:{"
                << z << "}\t" << std::endl;
        }
};

int main () {
    std::cout << std::endl;

    // Declaration that specifies initialization
    // of a variable and variable template
    std::tuple _tuple ('c', "monk", 9L, 1.22222);
    std::cout << '\t' << std::get<0>(_tuple) << '\t'
        << std::get<1>(_tuple)<< '\t' << std::get<2>(_tuple)
        << '\t' << std::get<3>(_tuple) << '\t' << std::endl;

    //Dynamic allocator Expression
    auto _sample2 = new sample(99999, 322, 21000);
    auto _sample = new sample("Adrian", "D", "Finlay");
    // This won't compile. Types must be the same.
    // auto _sample = new sample(99, "Monk", 32);

    std::cout << std::endl;
    return 0;
}
```



It appears that GCC hasn't yet implemented this feature. I could be wrong, but it appears that way. Perhaps there is a switch that needs to be enabled. If you can get this code to run on GCC, do let me know.

17) Extensions on over-aligned Memory Allocation

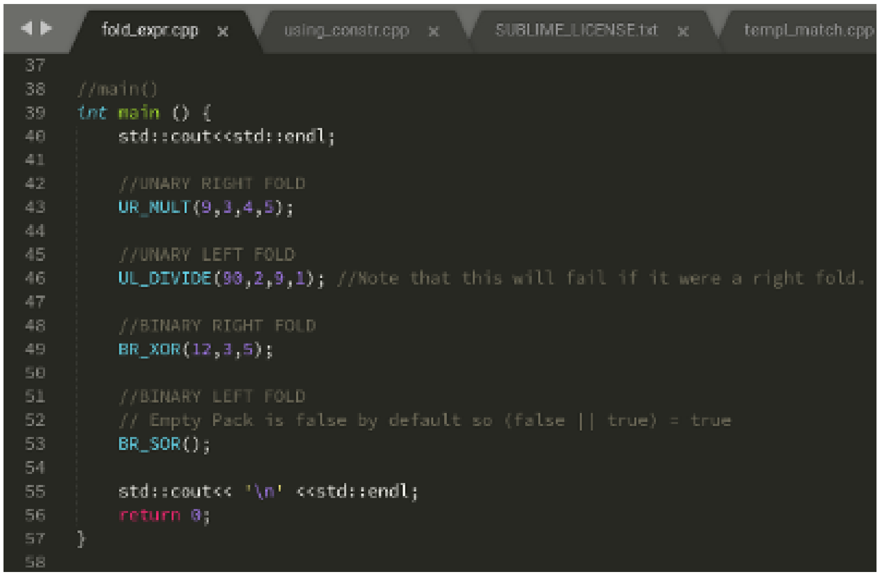
C++17 overloads the new operator to provide support for correctly dynamically allocating over-aligned data. Intel's compiler had supported this feature by way of their own work <aligned_new>. You may view the paper here. I also strongly recommend checking out these pages for more understanding as to the changes in memory management C++17: std::aligned_alloc , std::align_val_t, std::align. Lastly, you should look at the description about how the new operator has been overloaded to reflect these changes.

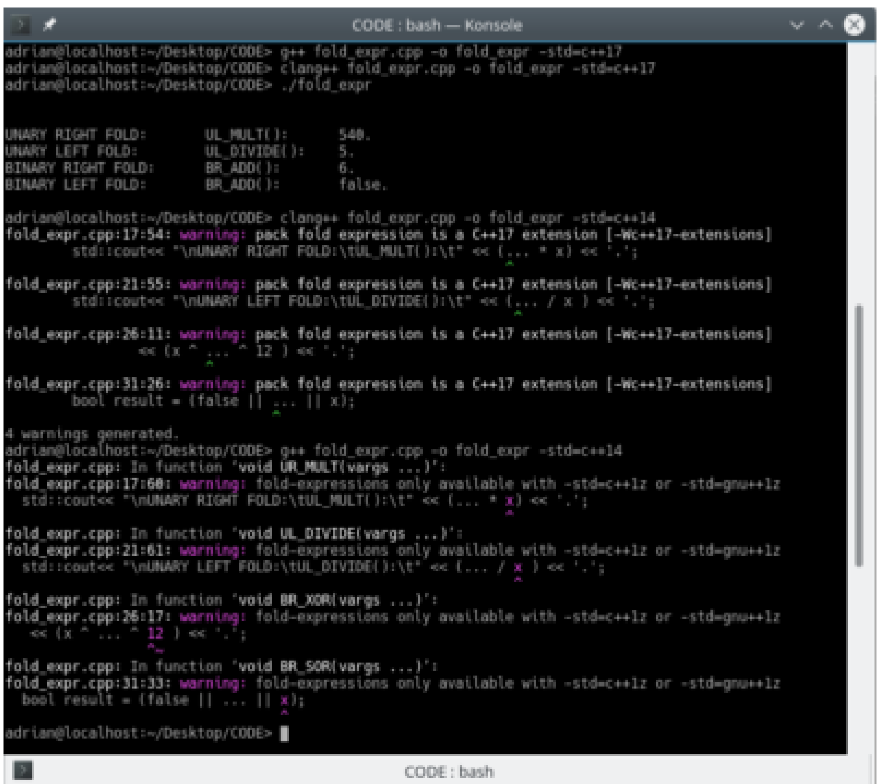
18) Fold Expressions

One of the major new features of C++, C++17 introduces Fold Expressions, a mechanism which reduces a parameter pack over a binary operator. While we will not cover all the aspects of fold expressions (you should **absolutely** do so yourself, see this page), we will show some basic use. There are binary and unary folds.

fold_expr.cpp × using_constr.cpp × SUBLIME_LICENSE.txt × templ_match.cpp

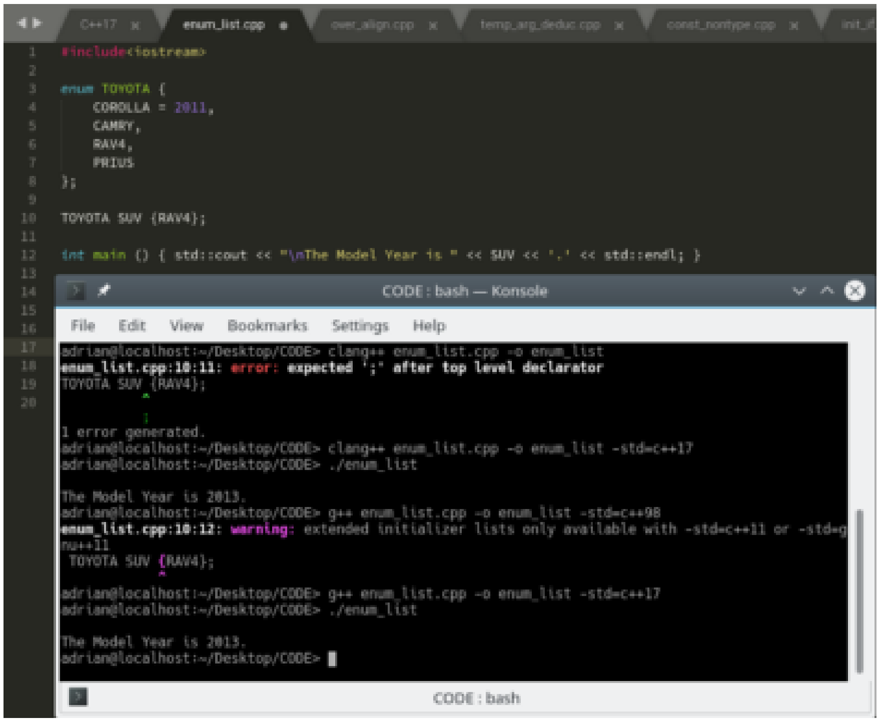
```cpp
#include<iostream>

/* Function Prototypes */

//Unary Right Fold ( pack op ... )
template<typename ...vargs> void UR_MULT (vargs... x);
//Unary Left Fold ( ... op pack )
template<typename ...vargs> void UL_DIVIDE (vargs... x);
//Binary Right Fold ( pack op ... op init )
template<typename ...vargs> void BR_XOR (vargs... x);
//Binary Right Fold ( op init ... pack op )
template<typename ...vargs> void BL_X (vargs... x);

//Function Definitions
template<typename ...vargs>
void UR_MULT (vargs... x) {
    std::cout<< "\nUNARY RIGHT FOLD:\tUL_MULT():\t" << (... * x) << '.';
}
template<typename ...vargs>
void UL_DIVIDE (vargs... x) {
    std::cout<< "\nUNARY LEFT FOLD:\tUL_DIVIDE():\t" << (... / x ) << '.';
}
template<typename ...vargs>
void BR_XOR (vargs... x) {
    std::cout<< "\nBINARY RIGHT FOLD:\tBR_ADD():\t"
        << (x ^ ... ^ 12 ) << '.';
}
template<typename ...vargs>
void BR_SOR (vargs... x) {
    std::cout<< "\nBINARY LEFT FOLD:\tBR_ADD():\t";
    bool result = (false || ... || x);
    if (result == 1)
        std::cout<< "true" << '.';
    else
        std::cout<< "false" << '.';
}

```





19) List-style Initialization of Enumerations

C++17 now introduces an optional attribute specifier sequence in initializing enumerations.

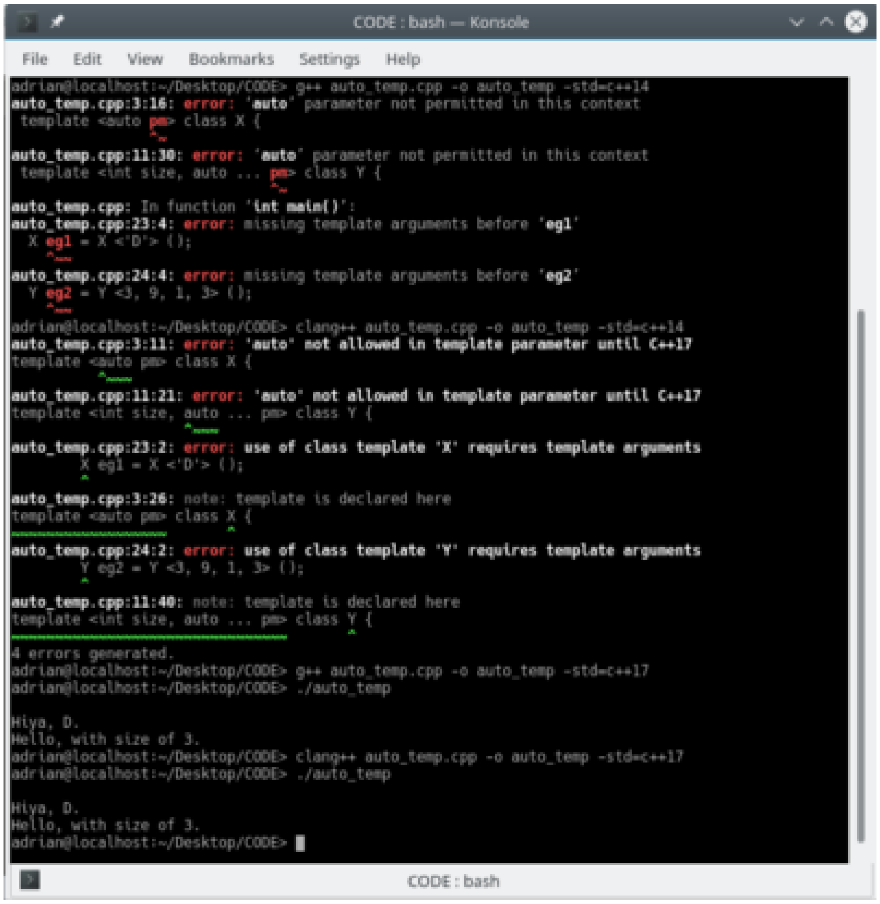


20) Specifying non-type template parameters with auto

C++17 now allows you to specify non-type template parameters with auto.

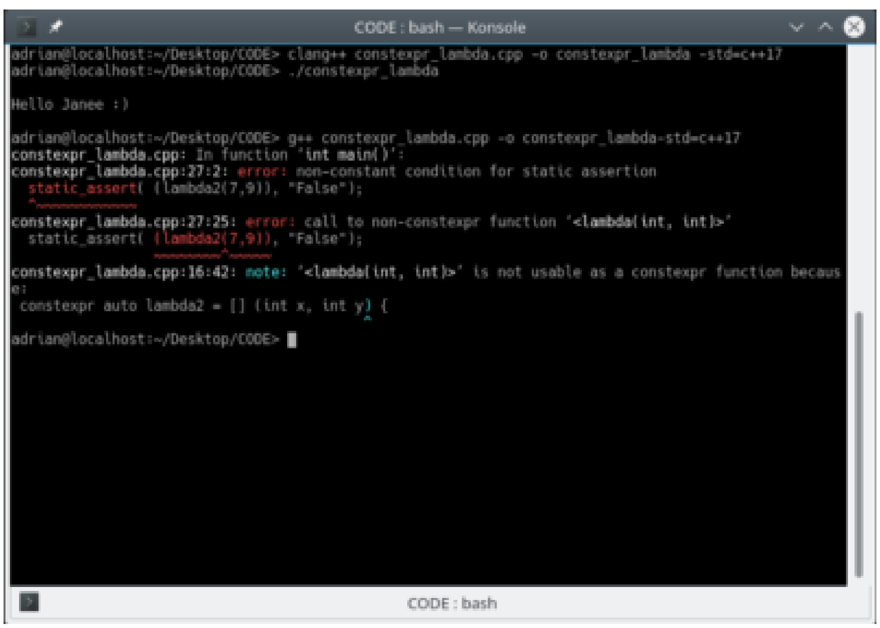
C++17 auto_temp.cpp enum_list.cpp

```cpp
#include<iostream>

template <auto pm> class X {
    public:
        X () {
            std::cout << "\n" << "Hiya, "
                << pm << '.' << std::endl;
        }
};

template <int size, auto ... pm> class Y {
    public:
        Y () {
            std::cout << "Hello, "
                << "with size of " << size
                << '.' << std::endl;
            //Expansion of Variadic Arguments
            //Not allowed here.
        }
};

int main () {
    X eg1 = X <'D'> ();
    Y eg2 = Y <3, 9, 1, 3> ();
}
```



21) constexpr lambda expressions

C++17 now allows you to explicitly specify a lambda expression as constexpr. In the absence of constexpr, had a lambda qualified, it would have been treated as constexpr anyway. This is still true. Read more about this, here.

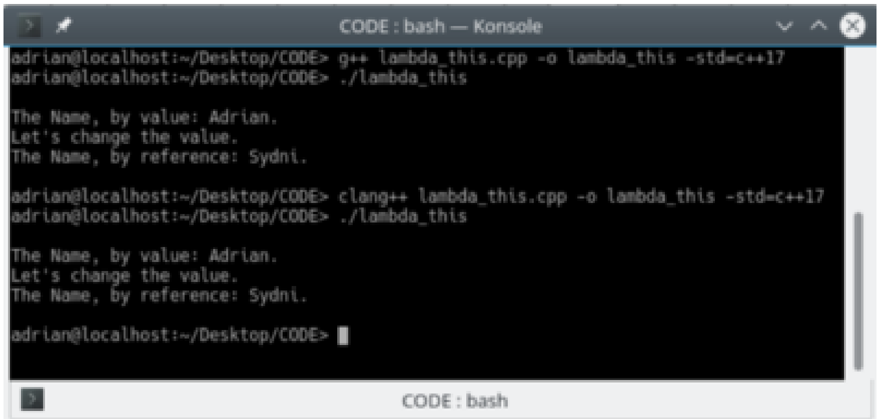


For some reason GCC gave me errors when attempting to compile/link/run but it seems to be a local issue, as other SUSE Linux users with similar configurations have built an executable with no errors. Many thanks to Ciel for assistance with this.

22) Lambda this by value (*this)

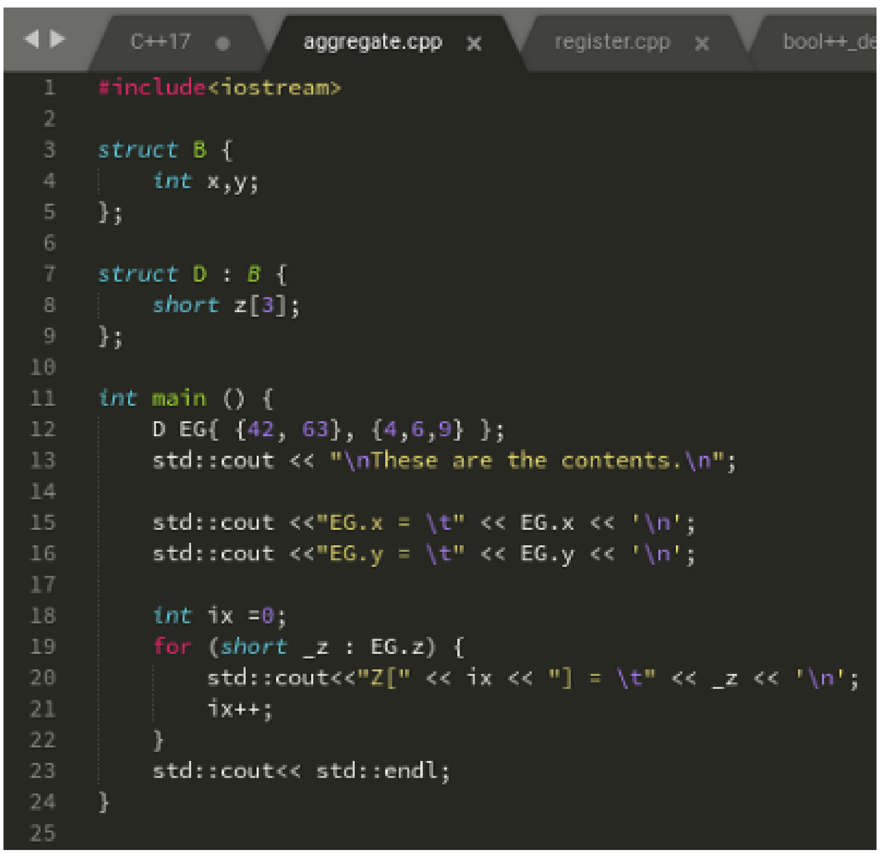
C++17 introduced the ability by a lambda expression to capture it's invoking object by value in addition to by reference.

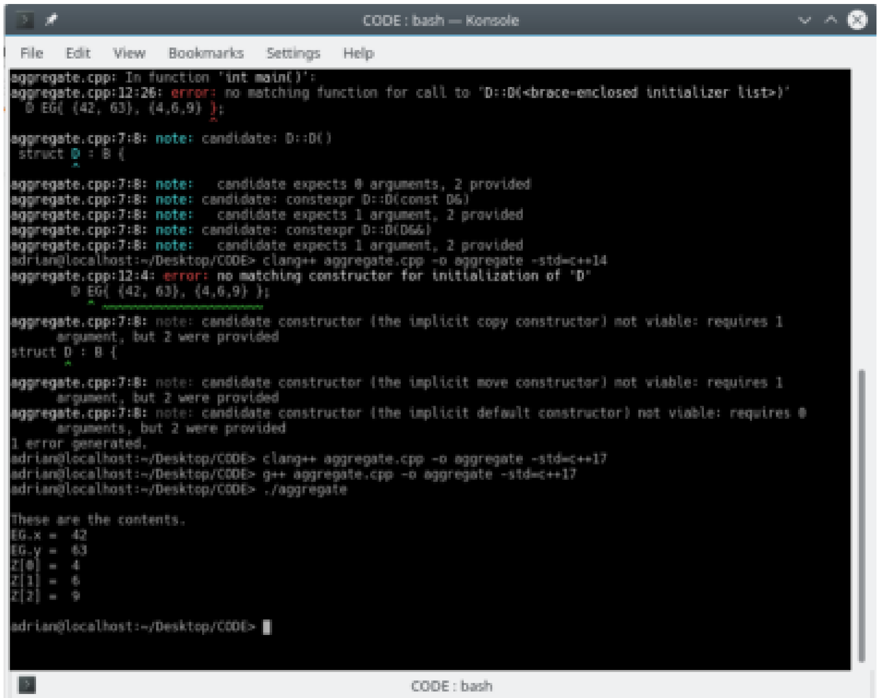
```cpp
#include <iostream>
#include <string>

class Obj {
    private:
        std::string name = "Adrian";
    public:
        //Lambda by Reference, Pre-C++17
        auto byRef () {
            return [this] { return name; };
        }

        //Lambda by Value, C++17
        auto byVal () {
            return [*this] { return name; };
        }

        //Set name
        void setName(std::string name) {
            std::cout<<"Let's change the value.\n";
            this->name=name;
        }
};


int main () {
    Obj ob;

    auto val = ob.byRef();
    std::cout<< '\n' << "The Name, by value: " << val() << ".\n";
    //Let's change name.
    ob.setName("Sydni");

    auto ref = ob.byVal();
    std::cout<< "The Name, by reference: " << ref() << ".\n";

    std::cout<< std::endl;
}
```



23) Extending Aggregate Initialization to Base Types

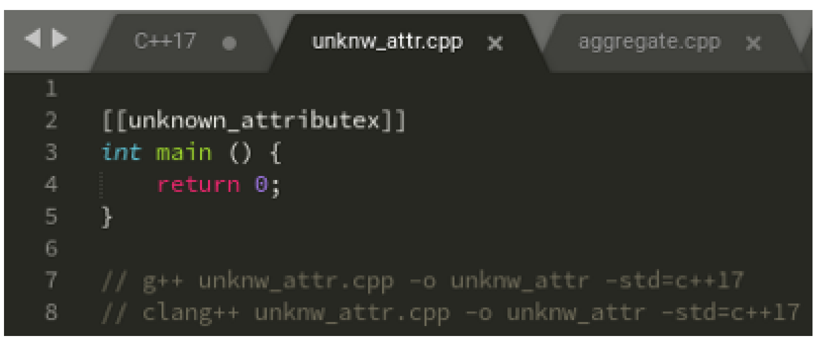
C++17 introduced aggregate initialization of base types. You may want to read more about this, here.

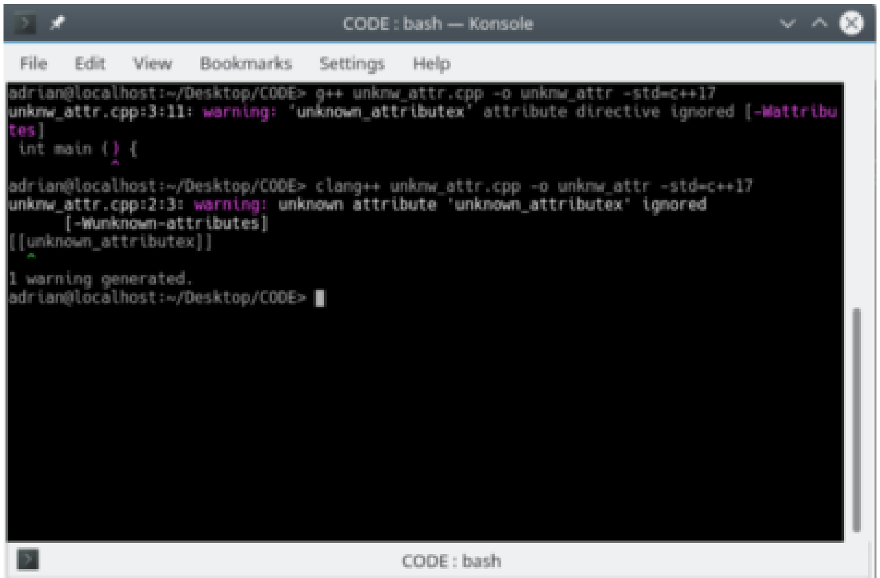




24) Unknown Attributes Required to be Ignored

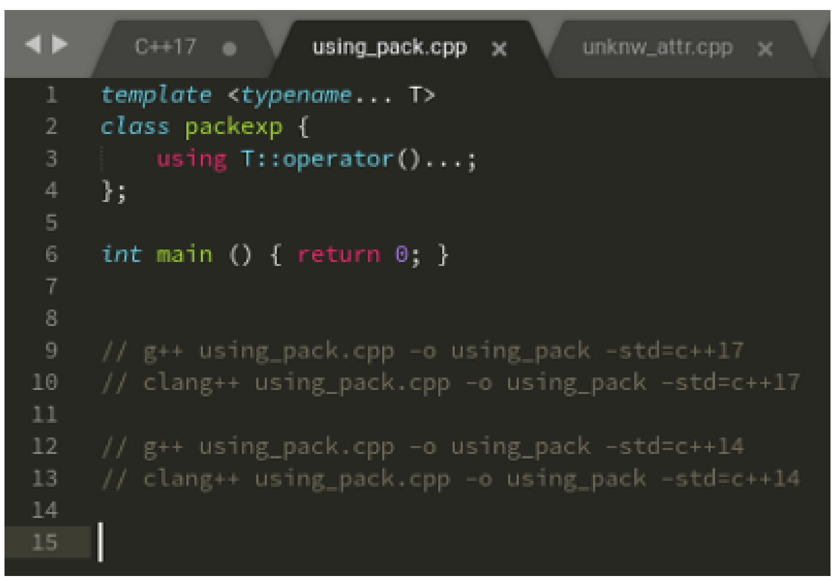
C++17 now mandates that unknown attributes are required to be ignored.

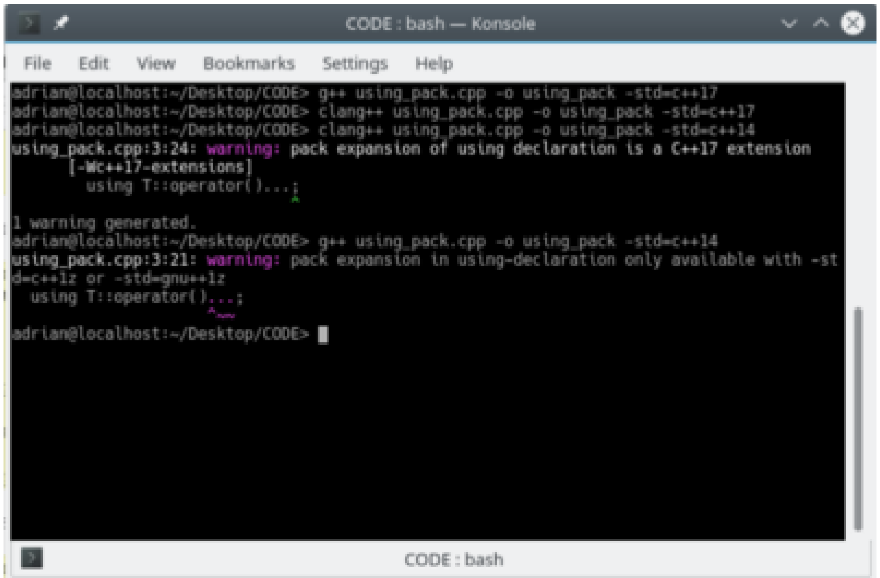




25) Pack Expansions legal in using declarations

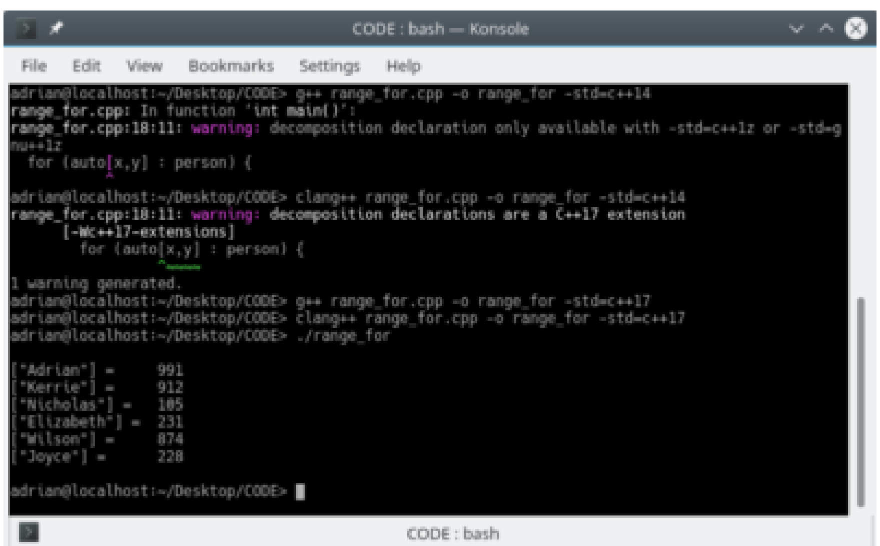
C++17 now allows using declarations to make use of pack expansions. You should read more about the motivations of this feature, here, as I will only show it's basic use.





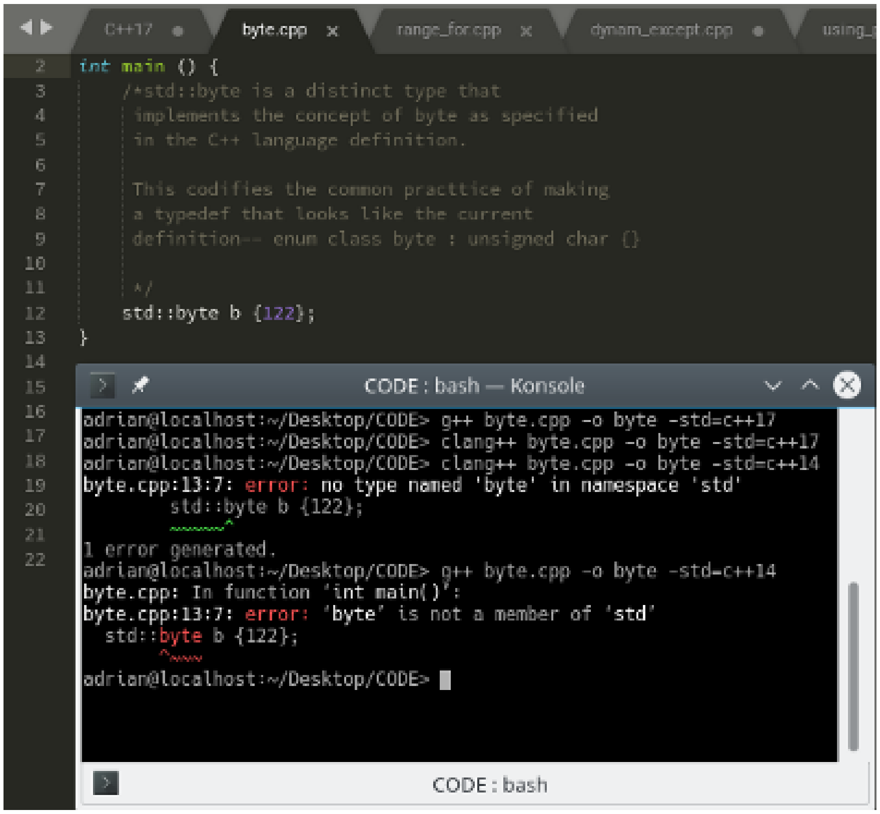
26) Generalization of Range-based for loop

C++17 generalizes the range-based for loop by relaxing the requirement that the beginning and ending types be of the same type. Consequently, You might want to see how this expands to a normal for loop, here. You may also find the changes in std::for_each and for_each_n.



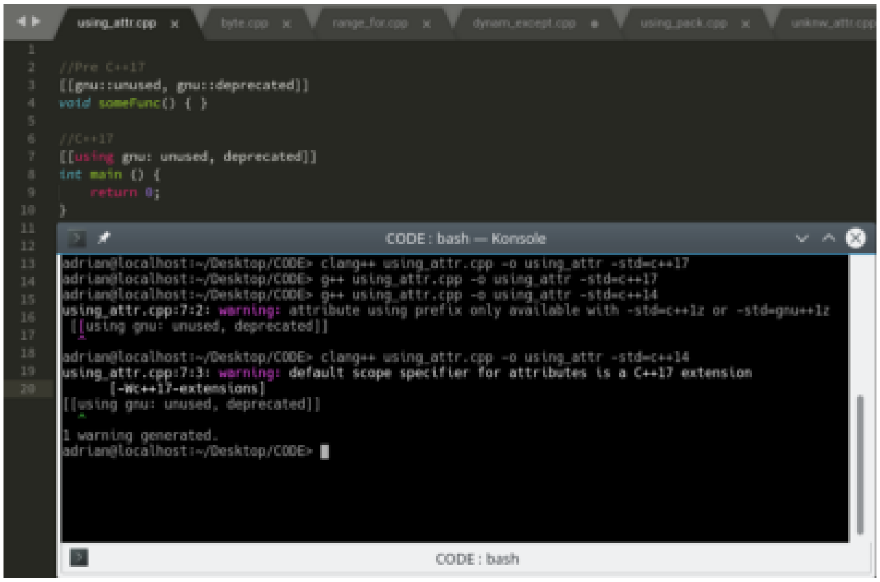
27) The byte data type

C++17 introduces the byte data type. std::byte is included in the cstddef header and is not an actual built in data type as byte and short are, for example. It is analogous to std::string, in a way.



28) Using attribute namespaces without repetition

C++17 cleaned up attribute namespace syntax, to make for cleaner expression. For attributes in the same namespace, one can specify several.



29) Stricter Order of Evaluation Rules

C++17 puts into law practices that have been expressed in code in the past but never explicitly stated in the standard, in an effort to avoid undefined behavior. Longtime C++ expert Herb Sutter is an author of the paper describing this in more detail, which you should **definitely read**.

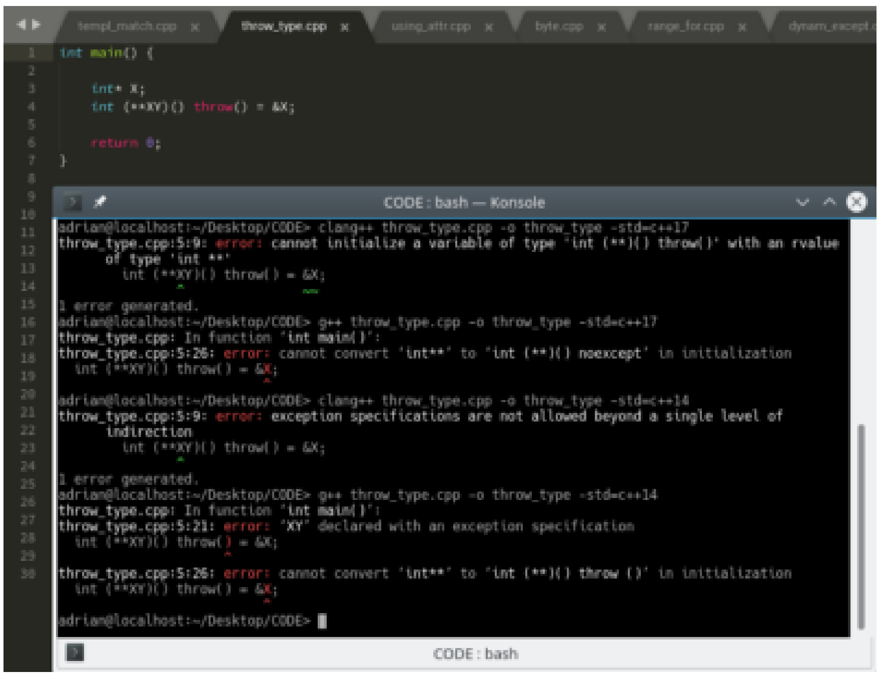
The new changes are [11]:

- "Postfix expressions are evaluated from left to right. This includes functions calls and member selection expressions."
- "Assignment expressions are evaluated from right to left. This includes compound assignments."
- "Operands to shift operators are evaluated from left to right."
- "The order of evaluation of an expression involving an overloaded operator is determined by the order associated with the corresponding built-in operator, not the rules for function calls. This rule is to support generic programming and extensive use of overloaded operators, which are distinctive features of modern C++."
- "The function is evaluated before all its arguments, but any pair of arguments (from the argument list) is indeterminately sequenced; meaning that one is evaluated before the other but the order is not

specified; it is guaranteed that the function is evaluated before the argument"

30) Exception Specifications are part of type definitions

C++17 modified the language of the standard to allow exception specifications to be part of the type system, thus differentiating types by their exception specifiers, something that was not previously a part of the standard.



Notice that in C++17 the compiler complains because the rvalue and lvalue don't match. This is because they now represent different types. C++14, however, **accepted this**, and instead complained about XY being declared with an exception specification. Notice, lastly, that g++14 gave us the same error stating that this feature change is supposed to change in C++17, while clang++ did not. The likely situation is that this feature had long been implemented before it's introduction into the ISO C++17 standard.

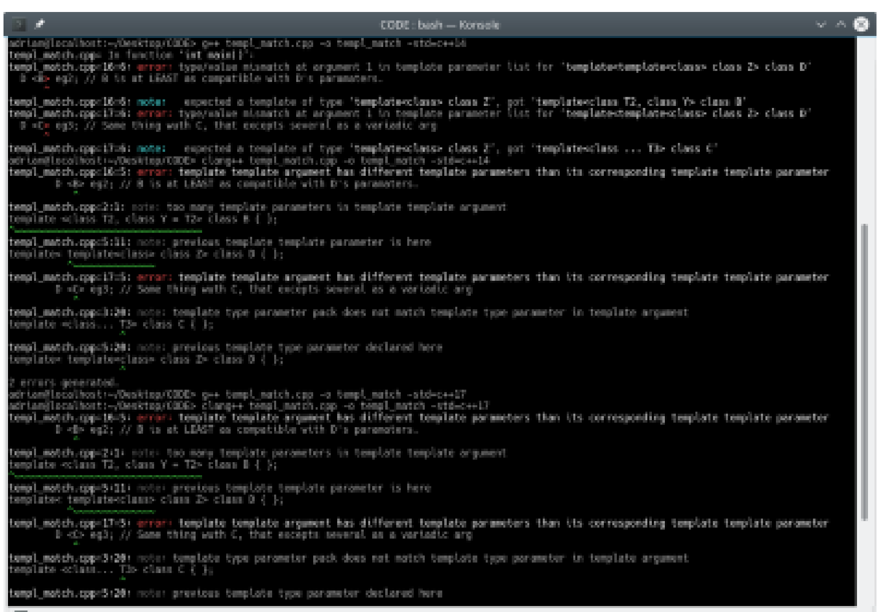
31) Template-Template Parameters match compatible arguments

C++17 provides support for matching template template parameters to a compatible argument. It's important to pay attention to the details of this paper. **The consequence of this is that any template**

argument list can be applied to the template-template parameter is also applicable to the argument template.

> *"This paper allows a template template-parameter to bind to a template argument whenever the template parameter is at least as specialized as the template argument. This implies that any template argument list that can legitimately be applied to the template template-parameter is also applicable to the argument template." [11]*

It does not appear that Clang/LLVM supports this feature fully, but I could be wrong. If I am, **do let me know in the comments below**. Notice that before C++17, this was an error, and after it, it is OK.



32) Guaranteed Copy Elision

C++17 now guarantees copy elision, standardizing a practice by certain compilers. That is, it "*omits copy constructors, resulting in zero-copy pass-by-value semantics*."[12] Consequently, if an expression within a call to the dynamic allocator returns a value compatible with the type invoked by the dynamic allocator (new X (new X())) then only **one call** rather than several, are made to the constructor of the type in question. The net effect is that less copies of objects are made, which could otherwise result in expensive operations. This tightens (in the standard) the behavior of objects when they are copied by value.This is similar to the behavior under the hood in languages like Java. This, among other things, have changed and you should read more about this, here.

33) Changes to Specification on Inheriting Constructors

C++17 introduces many updates to the wording regarding constructors. You should read about them, **here, here, & here.** This addition fixes eight core issues. Many of these changes involve the *using* keyword. One such example is:

```
struct BASE1 {
  BASE1(double x){}
};

struct BASE2 {
  BASE2(double x) {}
};

struct DERIVED : BASE1, BASE2 {
  using BASE1::BASE1;
  using BASE2::BASE2;
  /*
    Per ordinary overloading rules,
    DERIVED(double) overload BASE1(double)
    and BASE2(double). In C++17 this call is not
    ambigous. using CLASS:CONSTRUCTOR is different
    because instead of declaring a new set of constructors
    the base class consturctors are now visible, to make
    overloading possible.
  */
  DERIVED (double);
};
```

```cpp
int main () {
    //DERIVED::DOUBLE() CALLED
    DERIVED D(1.2);
    /*Because DERIVED:DOUBLE() is not defined, linking fails.*/
    /*
        GCC:
        g++ using_constr.cpp -o using_constr -std=c++17
        /tmp/ccWrGcsD.o: In function `main':
        using_constr.cpp:(.text+0x20): undefined reference to `DERIVED::DERIVED(double)'
        collect2: error: ld returned 1 exit status

        CLANG:
        clang++ using_constr.cpp -o using_constr -std=c++17
        /tmp/using_constr-d60afa.o: In function `main':
        using_constr.cpp:(.text+0x1c): undefined reference to `DERIVED::DERIVED(double)'
        clang-5.0: error: linker command failed with exit code 1 (use -v to see invocation)
    */
    return 0;
}
```

Notice that linking failed. Read more about this in the comments. This example was adapted from this **paper.**

Deprecated Language Features

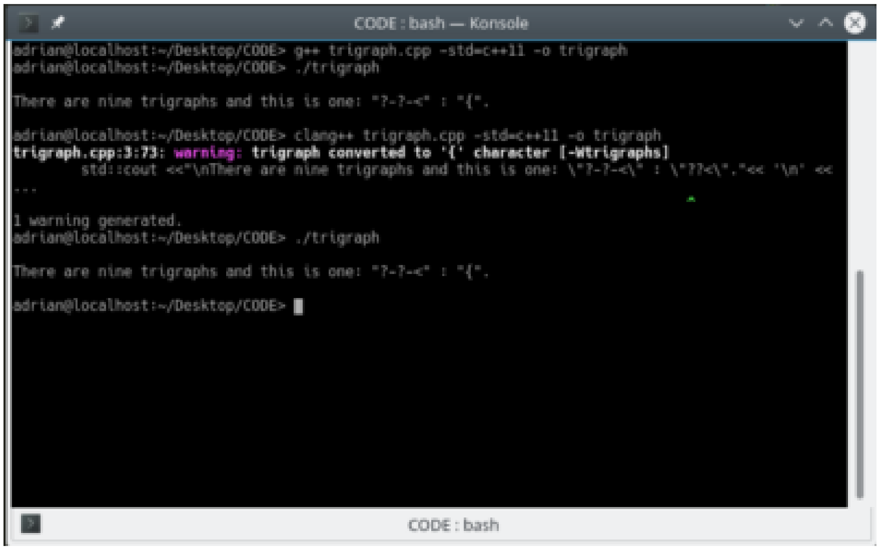
1) Removal of Trigraphs

C++17 brought in the removal of Trigraphs from the standard, though some compilers continue to provide support. MSVC, GCC, & Clang are among those that provide support with a command line switch. The switch to enable them in C++17 via GCC/Clang is **-trigraphs**.

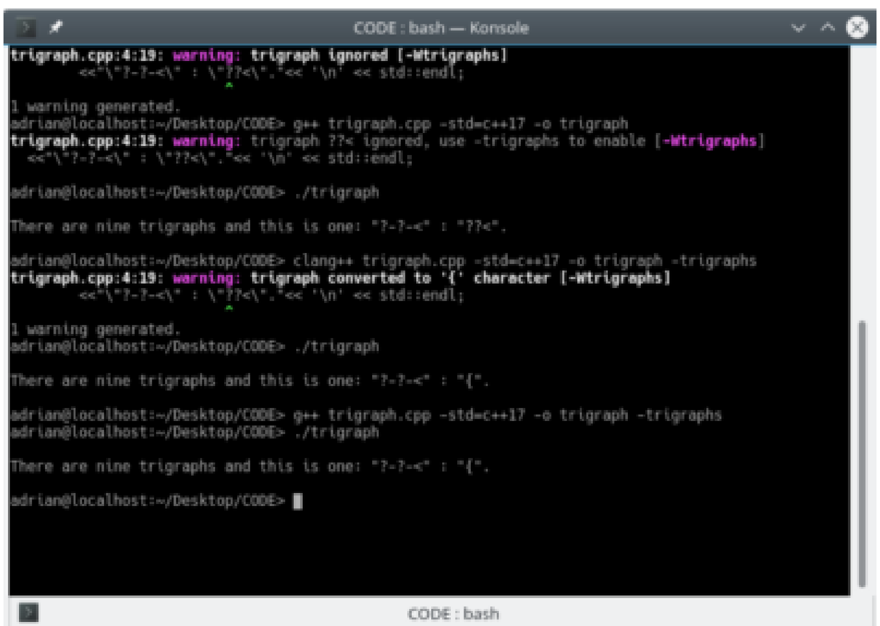
Trigraphs are a three character sequence beginning with **??** and ending in another element, like **!**. They allowed for C programs to be written in ISO 646:1983.There are 9 of them. The compiler recognizes a trigraph and replaces it with the value of a matching trigraph sequence. All of the trigraphs are punctuation symbols. C++ had inherited this feature from C.

```cpp
#include<iostream>
int main () {
    std::cout <<"\nThere are nine trigraphs and this is one: "
    <<"\"?-?-<\" : \"??<\"."<< '\n' << std::endl;
}

// clang++ trigraph.cpp -std=c++17 -o trigraph -trigraphs
// g++ trigraph.cpp -std=c++17 -o trigraph -trigraphs
```

Trigraphs work just fine under GCC/Clang C++11, albeit with a warning from Clang.

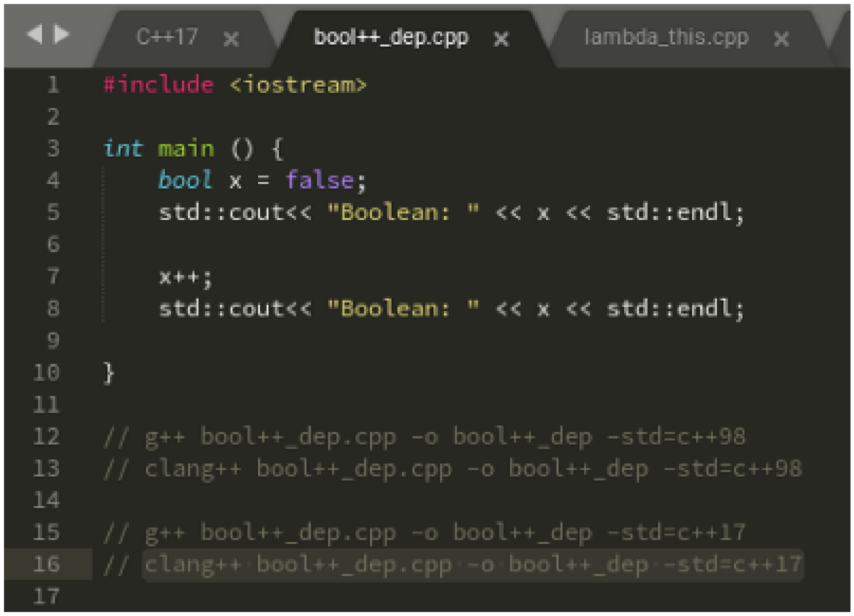


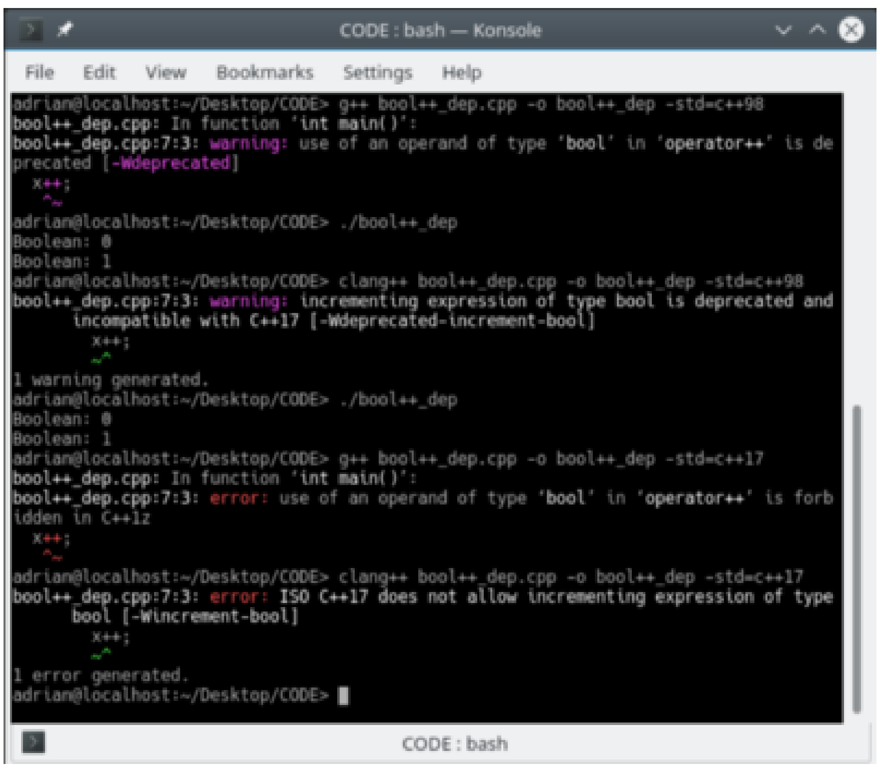
Unless we flip the switch, GCC/Clang C++17 will not recognize the trigraph.



2) Removal of deprecated Increment Operator (++) for bool type

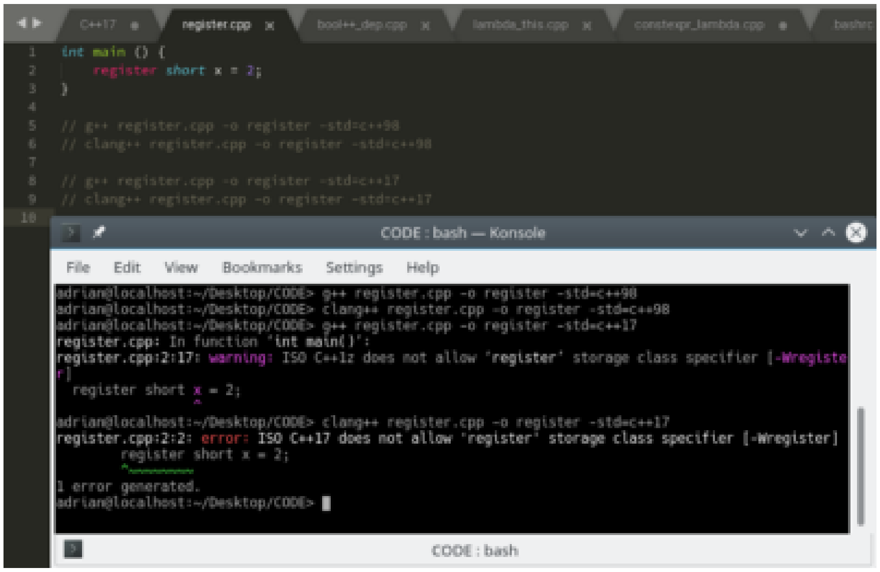
C++17 removed support for the increment operator for C++'s Boolean type. It had been deprecated since C++98 and has finally been removed. Notice that it will succeed with C++98 (albeit with a deprecation warning) but it will fail with C++17.





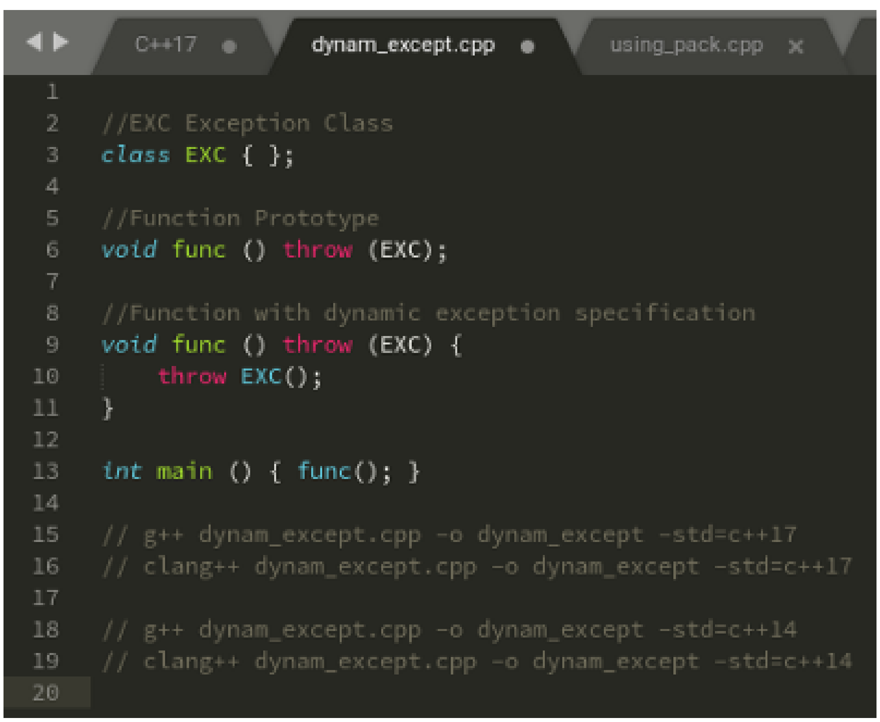
3) Removal of deprecated register keyword

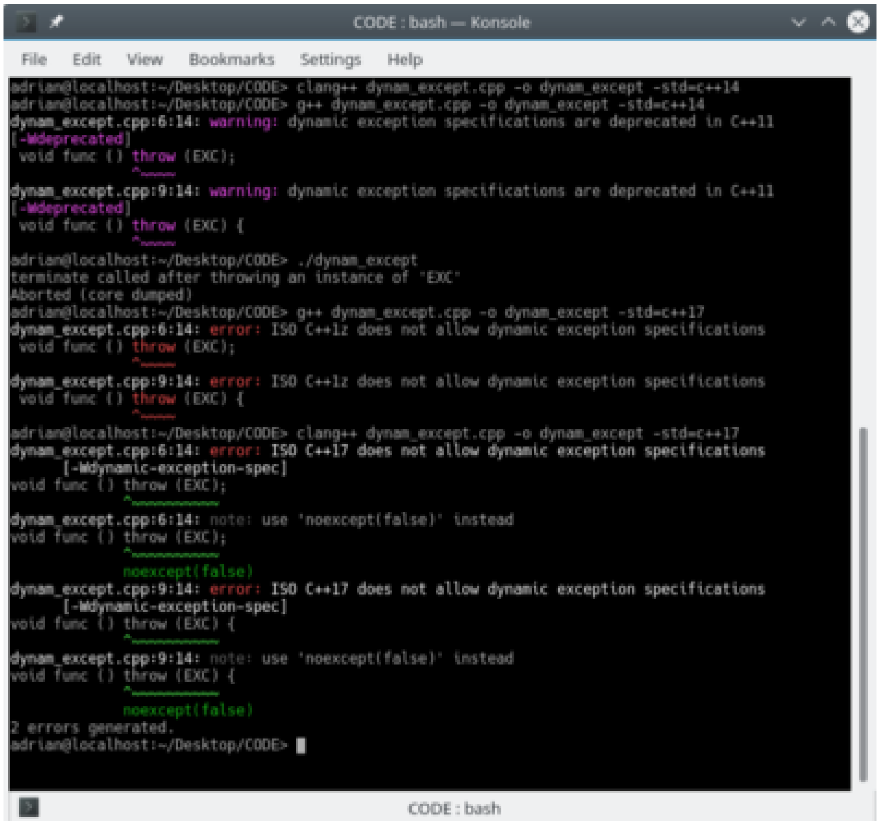
C++17 removed support for the increment operator for C++'s register keyword. It had been deprecated since C++11 and has finally been removed. Notice that it will succeed with C++98 but will fail with C++17.



4) Removal of deprecated Dynamic Exception Specifications

C++17 removed support for the dynamic exception specification. It had been deprecated since C++11 and has finally been removed. Notice that it will succeed with C++03 but will fail with C++17. It is recommended to use noexcept(true) in lieu of throw() and noexcept(false) in lieu of throw(…).





Note that clang being the less strict compiler did not warn us in C++14 while GCC did. In C++17 it is illegal by both, with Clang giving us more useful feedback.

Like the new features? Hate them? Did I forget some? Let me know in the comments below!

Looney Tunes Ending [4]

Works Cited

[1]—Open-STD: Working Draft, Standard for Programming Language C++

[2]—ISO CPP: Changes between C++14 and C++17 DIS

[3]—CppReference.com: C++ Compiler Support, C++17 Features

[4]—Stack Overflow: “What are the new features in C++17?”

[5]—ISO CPP: Changes between C++14 and C++17 DIS

[6]—GitHub: AnthonyCalandra/modern-cpp-features

[7]—Wikipedia: C++17

[8]—CppReference.com: inline specifier

[9]—ISOCPP: Current Status: Standard C+

[10]—OpenSTD: Refining Expression Evaluation Order for Idiomatic C++

[11]—OpenSTD: DR: Matching of template template-arguments excludes compatible templates

[12]—CppReference.com: copy elision