C++20 NEW FEATURES

Compiled with GCC C++2a Version Compiler

SEPTEMBER 14, 2020

CALSOFTINC
Bangalore

calsoft

Abstract

C++20 introduced many new features, In this document I have tried to cover almost all the new features are coming in C++20 version.

Prepared By

Haramohan.sahu@calsoftinc.com

Contents

Modules:

Modules is one of the five prominent features of C++20. Modules will overcome the restrictions of header files. For example, the separation of header and source files becomes as obsolete as the preprocessor. In the end, we will also have faster build times and an easier way to build packages.

C++20 introduces modules, a modern solution for componentization of C++ libraries and programs. A module is a set of source code files that are compiled independently of the translation units that import them. Modules eliminate or greatly reduce many of the problems associated with the use of header files, and also potentially reduce compilation times. Macros, preprocessor directives, and non-exported names declared in a module are not visible and therefore have no effect on the compilation of the translation unit that imports the module. You can import modules in any order without concern for macro redefinitions.

Declarations in the importing translation unit do not participate in overload resolution or name lookup in the imported module. After a module is compiled once, the results are stored in a binary file that describes all the exported types, functions and templates. That file can be processed much faster than a header file, and can be reused by the compiler every place where the module is imported in a project.

Modules can be used side by side with header files. A C++ source file can import modules and also #include header files.

How to do it in Microsoft Visual Studio 2019 (you must have 16.7.1 version or above).
Enable modules in the Microsoft C++ compiler, As of Visual Studio 2019 version 16.2, modules are not fully implemented in the Microsoft C++ compiler. You can use the modules feature to create single-partition modules and to import the Standard Library modules provided by Microsoft.
To enable support for modules, compile with /experimental:module and /std:c++latest.
In a Visual Studio project, right-click the project node in Solution Explorer and choose Properties. Set the Configuration drop-down to All Configurations,then choose Configuration Properties > C/C++ > Language > Enable C++ Modules (experimental).

A module and the code that consumes it must be compiled with the same compiler options.

Clang Compile:
clang++ -std=c++20 - -fmodules-ts -stdlib=libc++ -c math.cppm -Xclang -emit-module-interface -o math.pcm

Advantages of Modules:

1) Compile-time speedup:

Modules are only imported once and are literally for free. Compare this with M headers which are included in N translation units.
The combinatorial explosion means, that the header has to be parsed M*N times.

2) Isolation from the preprocessor macros:
we should get rid of the preprocessor macros. Using a macro is just text substitution excluding any C++ semantic. This has many negative consequences: For example, it may depend on in which sequence you include macros or macros, which can clash with already defined macros or names in your application. But in case of modules, it makes no difference, in which order you import modules.

3)Express the logical structure of your code:
Modules enable you to express the logical structure of your code. You can explicitly specify names that should be exported or not. You can bundle a few modules into a bigger module and provide them to your customer as a logical package.

4)No need for header files:
Because there is a module, there is no need to separate your source code into an interface and an implementation part. This means, modules just half the number of source files.

5)Get rid of ugly workarounds:
We are used to ugly workarounds such as "put an include guard around your header", or "write macros with LONG_UPPERCASE_NAMES".
To the contrary, identical names in modules will not clash.

Anyway. C++20's modules are dependent on the exact version of the compiler they are using. You can't use modules built by GCC on clang.
And you can't use modules built by GCC 9 on GCC 10.

***Example of* Module**

Put the following in hello.cc:

```
module;
#include <iostream>
#include <string_view>
export module hello;
export void greeter (std::string_view const &name)
{
  std::cout << "Hello " << name << "!\n";
}
```

and put the following in main.cc:

```
import hello;
int main (void)
{
  greeter ("world");
  return 0;
}
```

Now compile with:

```
g++ -fmodules-ts hello.cc main.cc
./a.out
Hello world!
```

2)

```
// helloworld.cpp
export module helloworld;  // module declaration
import <iostream>;         // import declaration

export void hello() {      // export declaration
    std::cout << "Hello world!\n";
}
// main.cpp
import helloworld;  // import declaration

int main() {
    hello();
}
```

First we will take a classical example of Old helloWorld.cpp

```
// helloWorld.cpp

#include <iostream>
int main() {
    std::cout << "Hello World" << std::endl;
}
```

```
g++ helloWorld.cpp -o helloWorld
wc -c helloWorld
8800 helloWorld
```

The size is more, reason there are lots of thing gets added into this helloWorld binary.

The classical* Build *Process:

The build process consists of three steps in this order:
1) preprocessing
2) compilation
3) linking.

***1)*Preprocessing**

The preprocessor handles the preprocessor directives such as #include and #define.
The preprocessor substitutes #inlude directives with the corresponding header files, and it substitutes the macros (#define). Thanks to directives such as #if, #else, #elif, #ifdef, #ifndef, and #endif parts of the source code can be included or excluded.

This straightforward text substitution process can be observed by using the compiler flag -E on GCC/Clang, or /E on Window

```
g++ -E helloWorld.cpp | wc -c
420880
```

The output of the preprocessor is the input for the compiler.

2)Compilation:
The compilation is separately performed on each output of the preprocessor. The compiler parses the C++ source code and converts it into assembly code. The generated file is called an object file and it contains the compiled code in binary form. The object file can refer to symbols, which don't have a definition.
Static Library:
The object files can be put in archives for later reuse. These archives are called static libraries.

The objects or translation units which the compiler produces are the input for the linker.
3) Linking:
The output of the linker can be an executable or a static or shared library. It's the job of the linker to resolve the references to undefined symbols. Symbols are defined in object files or in libraries. The typical error in this state is that symbols aren't defined or defined more than once. This build process consisting of the three steps is inherited from C. It works sufficiently good enough if you only have one translation unit. But when you have more than one translation unit, many issues can occur.

Issues of the Build Process:

There are flaws in the classical build process. Modules overcome these issues.

1) Repeated substitution of Headers:
let us create 2 files:

hello.cpp and hello.h

```
// hello.cpp

#include "hello.h"

void hello() {
   std::cout << "hello ";
}

// hello.h

#include <iostream>

void hello();
```

here also 2 files:
----
world.cpp and world.h

```
// world.cpp

#include "world.h"

void world() {
    std::cout << "world";
}
// world.h

#include <iostream>

void world();
```

helloWorld2.cpp

```
// helloWorld2.cpp

#include <iostream>

#include "hello.h"
#include "world.h"

int main() {

    hello();
    world();
    std::cout << std::endl;

}
```

Noe lets us compile:

```
g++ -E hello.cpp  | wc -c
659482
g++ -E world.cpp  | wc -c
659481
g++ -E helloWorld2.cpp  | wc -c
659593
```

This is a waste of compile-time.

In contrast, a module is only imported once and is literally for free.

Isolation from Preprocessor Macros:

Macros depend on in which sequence we include macros or macros can clash with already defined macros or names in your application. Imagine we have to headers webcolors.h and productinfo.h.

```
// webcolors.h

#define RED   0xFF0000


// productinfo.h

#define RED   0
```

When a source file client.cpp includes both headers, the value of the macro RED depends on the sequence the headers are included.
This dependency is very error-prone.
In contrast, it makes no difference, in which order you import modules.

Multiple Definition of Symbols:
ODR stands for the One Definition Rule and says in the case of a function.

A function can have not more than one definition in any translation unit. A function can have not more than one definition in the program. Inline functions with external linkage can be defined in more than one translation. The definitions have to satisfy the requirement that each definition has to be the same.

example:

```
// header.h

void func() {}
// header2.h

#include "header.h"
// main.cpp

#include "header.h"
#include "header2.h"

int main() {
}
```

The linker complains about the multiple definitions of func:

We are used to ugly workarounds such as put an include guard around your header. Adding the include guard FUNC_H to the header file header.h solves the issue.

```
// header.h

#ifndef FUNC_H
#define FUNC_H

void func(){}
```

#endif

In contrast, identical symbols with modules are very unlikely.

Coroutines:

Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Coroutines are functions that can suspend and resume their execution while keeping their state. The evolution of functions goes in C++ one step further.

What we see as a new idea in C++20 is quite old. Melvin Conway coined the term coroutine. He used it in his publication on compiler construction in 1963. Donald Knuth called procedures a special case of coroutines. With the new keywords co_await and co_yield, C++20 extends the execution of C++ functions with two new concepts.

Thanks to co_await expression expression, it is possible to suspend and resume the execution of the an expression. If you use co_await expression in a function func, the call auto getResult = func() does not block if the result of the function is not available.

Instead of resource-consuming blocking, you have resource-friendly waiting. co_yield expression expression allows it to write a generator function.

The generator function returns a new value each time. A generator function is a kind of data stream from which you can pick values. The data stream can be infinite. Consequentially, we are in the center of lazy evaluation.

What are Coroutines?

Coroutines are stackless functions designed for enabling co-operative Multitasking, by allowing execution to be suspended and resumed. Coroutines suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack.

This allows for sequential code that executes asynchronously (e.g. to handle non-blocking I/O without explicit callbacks), and also supports algorithms on lazy-computed infinite sequences and other uses. This is why coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, state machines, and pipes.

Why Coroutines is Required?

In order to read a file and parse it while reading into some meaningful data, one can either read it step by step at every line or may also load the entire content in memory, which would not be recommended for large text cases e.g text editors like Microsoft Word.

In general, to throw away the stack concept completely, it is needed. And also when we want things to do concurrently i.e. non-preemptive multitasking, we need coroutines for concurrency.

How Coroutines Works:

It rescheduled at specific points in the program and do not execute concurrently, programs using coroutines can avoid locking entirely.This is also considered as a benefit of event-driven or asynchronous programming.

It is almost similar to threads but one main difference is that threads are typically preemptively scheduled while coroutines are not and this is because threads can be rescheduled at any instant and can execute concurrently while coroutines rescheduled at specific points and
not execute concurrently.

Use Cases of Coroutines

- State Machines
- Actor Model
- Generators
- Communicating Sequential Processes
- Reverse Communication

State Machines: It is useful to implement state machines within a single subroutine, where the state is determined by the current entry or exit point of the procedure. This results in the more readable code as compared to the use of goto.

Actor Model: It is very useful to implement the actor model of concurrency. Each actor has its own procedures, but they give up control to the central scheduler, which executes them sequentially.

Generators: It is useful to implement generators which are useful for streams particularly input/output and for traversal of data structures.

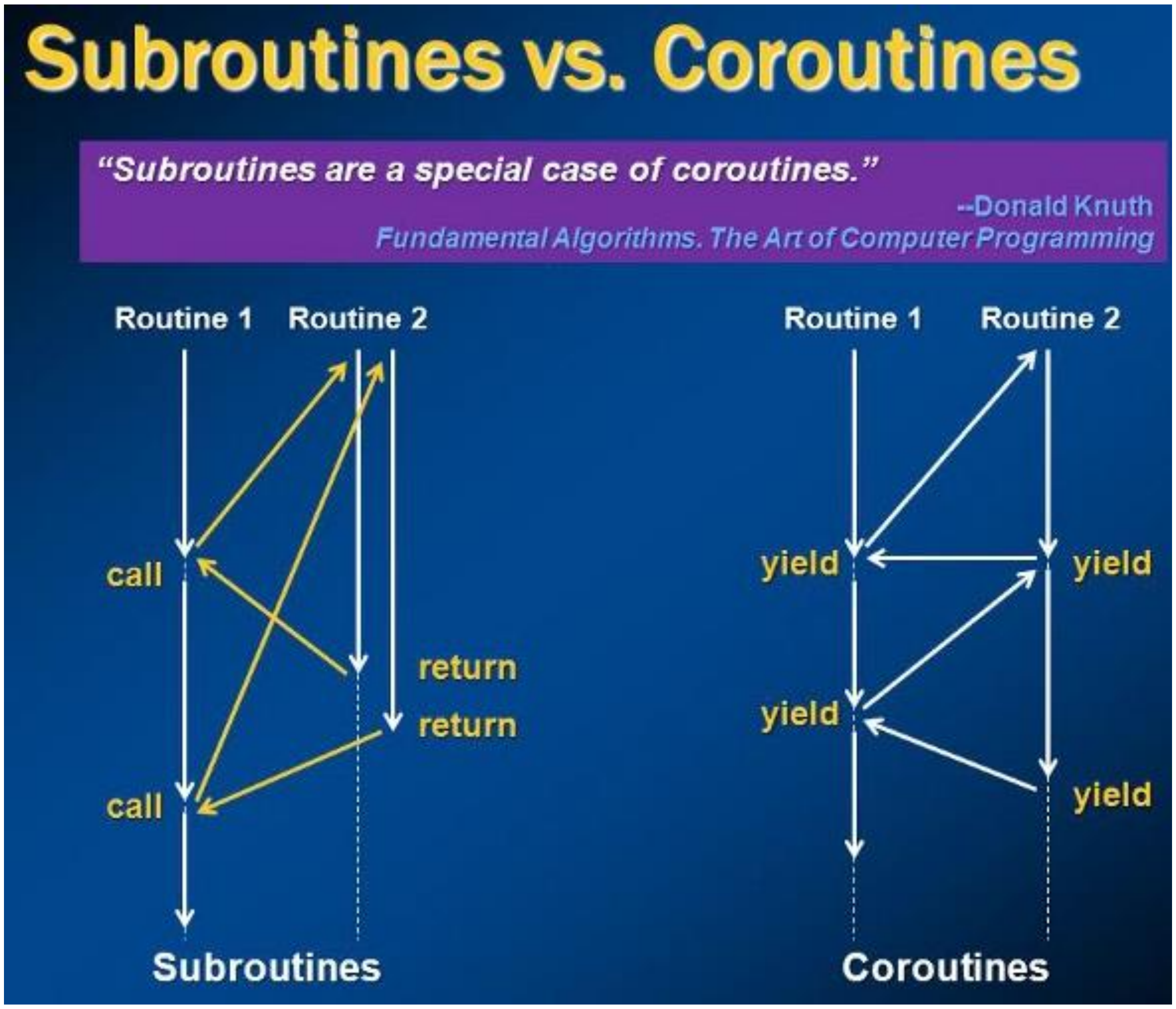
Communicating Sequential Processes: It is useful to implement communicating sequential processes where each sub-process is a coroutine.

Channel input/output and blocking operation yield coroutines and a scheduler unblock them on completion events.

Reverse Communication: They are useful to implement reverse communication which is commonly used in mathematical software, wherein a procedure needs the using process to make a computation.

Coroutines vs Subroutines?
With subroutines, execution begins at the start and finished on exit.
Subroutines are special cases of coroutines. Any subroutine can be translated to a coroutine which does not call ‘yield’ (relinquish control). Subroutines only return once and don't hold the complete state between invocations.

In contrast —
Coroutines can exit by calling other coroutines, which may later return to the point where they were invoked in the original coroutine; from the coroutine's point of view, it is actually not exiting but calling another coroutine. A coroutine instance holds state and varies between invocations.

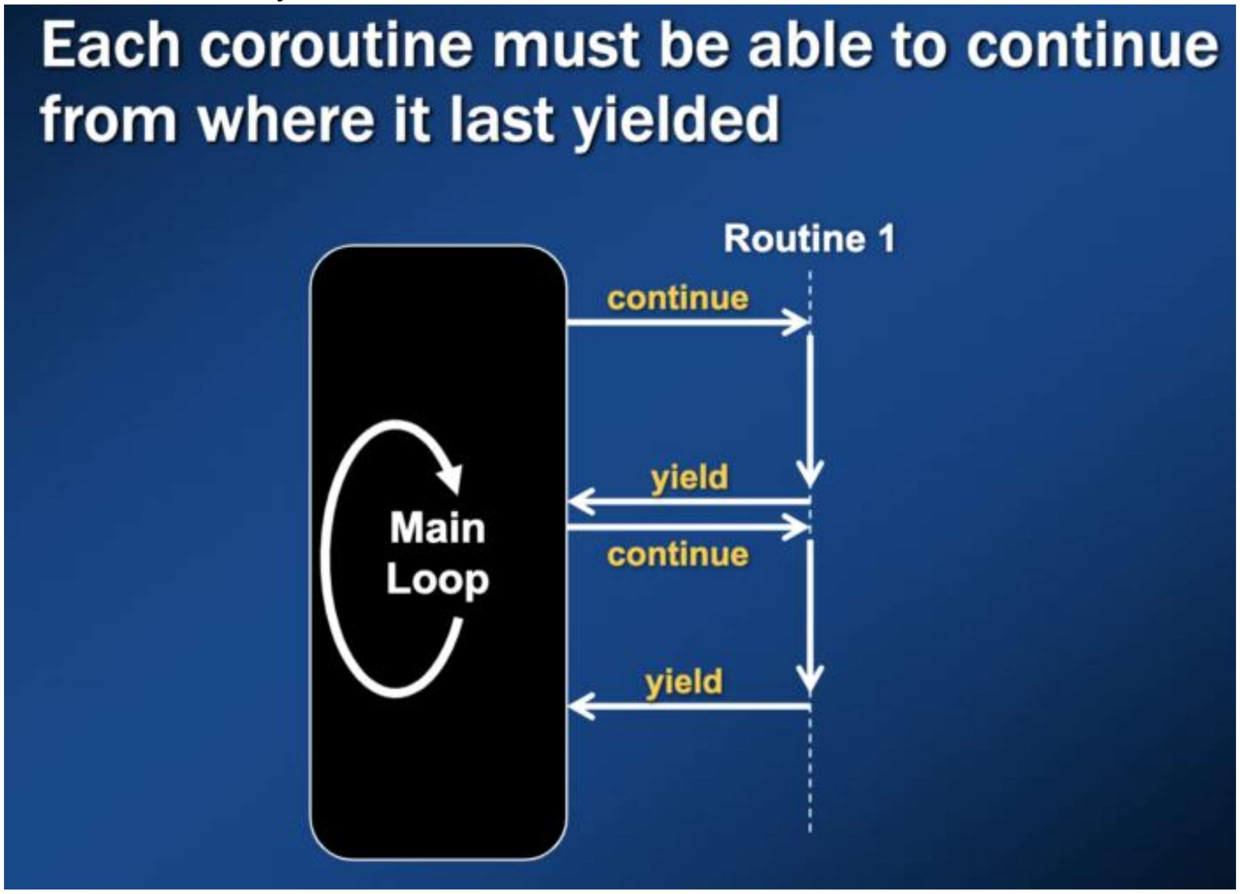

Subroutines vs. Coroutines
"Subroutines are a special case of coroutines."
--Donald Knuth
Fundamental Algorithms. The Art of Computer Programming
Routine 1
Routine 2
call
return
return
call
Subroutines
Routine 1
Routine 2
yield
yield
yield
yield
Coroutines

<u>Coroutines vs Threads</u>
Coroutines are designed to be lightweight threads. It uses lower RAM, because when you execute 1,000,000 concurrent routines, it doesn't have to create 1,000,000 threads. Coroutine can help you to optimise the threads usage, and make the execution more efficiency, and you don't need to care about the threads anymore. You can consider a coroutine as a runnable or task, which you can post into a handler and executed in a thread or threadpool.

- Coroutines provide concurrency but not parallelism [Important!]
- Switching between coroutines need not involve any system/blocking calls so no need for synchronization primitives such as mutexes, semaphores.

coroutines —

- provide asynchronicity and resource locking isn't needed.
- It is useful in functional programming techniques.
- increase locality of reference.


Each coroutine must be able to continue from where it last yielded
Routine 1
Main Loop
continue
yield
continue
yield

<u>Applications of Coroutines:</u>

Actor Model: They are very useful to implement the actor model of concurrency. Each actor has its own procedures, but they give up control to the central scheduler, which executes them sequentially.

Generators: It is useful to implement generators that are targeted for streams particularly input/output and for traversal of data structures. Reverse Communication: They are useful to implement reverse communication which is commonly used in mathematical software, wherein a procedure needs the using process to make a computation.

Example 1 — You have a consumer-producer relationship where one routine creates items and adds them to a queue and another removes items from the queue and uses them. For reasons of efficiency, you want to add and remove several items at once. The pseudo-code might look like this:

```
var q := new queue
coroutine produce
   loop
      while q is not full
         create some new items
         add the items to q
```

```
        yield to consume
coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```
The queue is then completely filled or emptied before yielding control to the other coroutine using the yield command.
(This example is often used as an introduction to multithreading, two threads are not a must need for this).

Coroutines in C++20
In c++20, coroutines are coming. A function is a coroutine if its definition does any of the following: uses the co_awaitoperator to suspend execution until resumed. uses the keyword co_yield to suspend execution returning a value. uses the keyword co_return to complete execution.

```
#include <iostream>
#include <vector>
// Coroutine gets called on need
generator<int> generateNumbers(int begin, int inc = 1) {

  for (int i = begin;; i += inc) {
    co_yield i;
  }

}

int main() {

    std::cout << std::endl;

    const auto numbers= generateNumbers(-10);

    for (int i= 1; i <= 20; ++i)
        std::cout << numbers << " "; // Runs finite = 20 times


    for (auto n: generateNumbers(0, 5)) // Runs infinite times
         std::cout << n << " "; // (3)

    std::cout << "\n\n";

}
```

Coroutines Restrictions:

1. Can’t return with variadic arguments
2. Can’t return using plain “return”
3. Can’t return placeholder (auto or Concept)
4. Can’t be constexpr functions.
5. Can’t be constructors or destructors.
6. Can’t be the main function.

Benefits of Coroutines

1. Implement in asynchronous programming.
2. Implement functional programming techniques.
3. Implement it because of poor support for true parallelism.
4. Pre-emptive scheduling can be achieved using coroutines.
5. Keep the system’s utilization high.
6. Requires less resource than threads.
7. Resource locking is less necessary.
8. Increases locality of reference.

A function is a coroutine if its definition does any of the following:

uses the co_await operator to suspend execution until resumed

```
task<> tcp_echo_server() {
  char data[1024];
  for (;;) {
    size_t n = co_await socket.async_read_some(buffer(data));
    co_await async_write(socket, buffer(data, n));
  }
}
```

uses the keyword co_yield to suspend execution returning a value

```
generator<int> iota(int n = 0) {
  while(true)
    co_yield n++;
}
```

uses the keyword co_return to complete execution returning a value

```
lazy<int> f() {
  co_return 7;
}
```

Every coroutine must have a return type that satisfies a number of requirements

co_await

The unary operator co_await suspends a coroutine and returns control to the caller. Its operand is an expression whose type must either define operator co_await, or be convertible to such type by means of the current coroutine's Promise::await_transform

co_yield

Yield-expression returns a value to the caller and suspends the current coroutine: it is the common building block of resumable generator functions

co_await promise.yield_value(expr)

The Concept Of Concepts:

Concepts make sure that data used within a template fulfill a specified set of criteria, and verifies this at the beginning of the compilation process. So as an example, instead of checking that an object is_integral, an object of type Integral is used. As a result, the compiler can provide a short and meaningful error message if the defined requirement of a concept isn't met, instead of dumping walls of errors and warnings from somewhere deep within the template code itself that won't make much sense without digging further into that code.

Concepts are named compile-time predicates which constrain types. A concept is a compile-time predicate (that is, something that yields a Boolean value). Constrains on the template parameters and meaningful compiler messages in a case on an error. Can also reduce the compilation time. It is an extension to the templates feature provided by the C++ programming language. Concepts are named Boolean predicates on template parameters, evaluated at compile time. A concept may be associated with a template (class template, function template, or member function of a class template),in which case it serves as a constraint: it limits the set of arguments that are accepted as template parameters.

Concepts are predicates that you use to express a generic algorithm's expectations on its template arguments.Concepts allow you to formally document constraints on templates and have the compiler enforce them.
As a bonus, you can also take advantage of that enforcement to improve the compile time of your program via concept-based overloading.

Any function can use keyword requires to tell compiler which concept(s) this function should fulfill. Using another words, "constrain the arguments' type in the template".

Apart from letting the compiler know what data is needed, it also shows rather clearly to other developers what data is expected, helping to avoid error messages in the first place, and avoids misunderstandings that lead to bugs later on. Going the other direction, Concepts can also be used to constrain the return type of template functions, limiting variables to a Concept rather than a generic auto type, which can be considered at C++'s void * return type.

A concept is a named set of requirements. The definition of a concept must appear at namespace scope.

The definition of a concept has the form

```
template < template-parameter-list >
concept concept-name = constraint-expression;
```

```
// concept
template <class T, class U>
concept Derived = std::is_base_of<U, T>::value;
```

<u>The main uses of concepts are:</u>

1) Introducing type-checking to template programming
2) Simplified compiler diagnostics for failed template instantiations
3) Selecting function template overloads and class template specializations based on type properties
4) Constraining automatic type deduction

//! https://www.modernescpp.com/index.php/thebigfour
//! https://dev.to/fenbf/c-20-cheatsheet-with-examples-440g

Example:
```
template <class T>
concept SignedIntegral = std::is_integral_v<T> &&
                         std::is_signed_v<T>;

template <SignedIntegral T> // no SFINAE here!
void signedIntsOnly(T val) { }
```

How concept helps?

<u>1) Compiler diagnostics:</u>
If a programmer attempts to use a template argument that does not satisfy the requirements of the template, the compiler will generate an error. When concepts are not used, such errors are often difficult to understand because the error is not reported in the context of the call, but rather in an internal, often deeply nested, implementation context where the type was used.

For example, std::sort requires that its first two arguments be random-access iterators. If an argument is not an iterator, or is an iterator of a different category, an error will occur when std::sort attempts to use its parameters as bidirectional iterators:

```
std::list<int> l = {2, 1, 3};
std::sort(l.begin(), l.end());
```

Typical compiler diagnostic without concepts is over 50 lines of output, beginning with a failure to compile an expression that attempts to subtract two iterators:

```
---------------- ERROR----------------------------------
In instantiation of 'void std::__sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = std::_List_iterator<int>; _Compare = __gnu_cxx::__ops::_Iter_less_iter]':
error: no match for 'operator-' (operand types are 'std::_List_iterator<int>' and 'std::_List_iterator<int>')
std::__lg(__last - __first) * 2,
---------------------------------------------------------
```

If concepts are used, the error can be detected and reported in the context of the call:

```
error: cannot call function 'void std::sort(_RAIter, _RAIter) [with _RAIter = std::_List_iterator<int>]'
note:   concept 'RandomAccessIterator()' was not satisfied
```

Here in below example;

```
template<typename T, bool enable = true>
class Sample
{
public:
    int DisableThisMethodOnRequest() requires(enable) { return 42; }
};


template<typename T, bool enable = true>
class Sample_1
{
public:
    std::enable_if_t<enable, int> DisableThisMethodOnRequest() { return 42; }
};


int main()
{

    Sample <int, true> ss;
    cout << " Sanple = " << ss.DisableThisMethodOnRequest() << endl;
    return 0;
}
```

<u>some example:</u>

```
// placeholdersDraft.cpp

#include <iostream>
#include <type_traits>
#include <vector>
#include<concepts>
using namespace std;

template<typename T>                              // (1)
concept Integral = std::is_integral<T>::value;


Integral auto getIntegral(int val) {              // (2)
    return val;
}
class A {};


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

    std::vector<int> vec{ 11, 22, 33, 44, 55 };
    for (Integral auto i : vec)
        std::cout << i << " ";                          // (3)

    Integral auto b = true;                             // (4)
    std::cout << b << std::endl;

    Integral auto integ = getIntegral(10);              // (5)
    std::cout << integ << std::endl;

    auto integ1 = getIntegral(12.90);                   // (6)
    std::cout << integ1 << std::endl;

  // Integral auto aa = A();
             // std::cout << aa << std::endl; // This will cause error.
    std::cout << std::is_integral<A>::value << '\n';

    std::cout << std::endl;

}
```

concept exampe where we can have different syntactical sugar of concept use .

```
// placeholdersDraft.cpp

#include <iostream>
#include <type_traits>
#include <vector>
#include<concepts>
using namespace std;
struct Dog {};

struct Husky : public Dog {};
struct Poodle :public Dog {};

template<class T>
struct is_husky {
            static constexpr bool result = false;
};
template<>
struct is_husky<Husky> {
            static constexpr bool result = true;
};

template<typename T>
concept  ExceptHusky = !is_husky<T>::result; // If T is husky, this concept returns false


template<ExceptHusky T>
```

```
void DogCanBeSmartExceptHusky(const T&& dog) {};



//also works. This is for multiple require expressions, they can use ||, &, ! such operators to join
template<class T>
//requires ExceptHusky<T>
//! like below too we can write
requires is_husky<T>::result
void DogCanBeSmartExceptHusky_1(const T&& dog) {};



int main()
{
	DogCanBeSmartExceptHusky(Poodle{});
	//DogCanBeSmartExceptHusky(Husky{});
	// Error message here:
	// constraints not satisfied
	//'!(is_husky<Husky>::result)' evaluated to false
}
```

ExceptHusky is a concept that is required by DogCanBeSmartExceptHusky function. Function can be instantiated only when all requirements(constraints) has been fulfilled (final result of require list is true).

Concept also improves the readability of the code, especially in heavy use of template (**SFINAE**).

Requires clauses:

The keyword requires is used to introduce a requires-clause, which specifies constraints on template arguments or on a function declaration.

```
template<typename T>
void f(T&&) requires Eq<T>; // can appear as the last element of a function declarator

template<typename T> requires Addable<T> // or right after a template parameter list
T add(T a, T b) { return a + b; }
```

In this case, the keyword requires must be followed by some constant expression (so it's possible to write requires true),
but the intent is that a named concept (as in the example above) or a conjunction/disjunction of named concepts or a requires-expression is used.

Example:
========
The requires keyword is used either to start a requires clause or a requires expression:

```
template <typename T>
  requires my_concept<T> // `requires` clause.
void f(T);
```

```
template <typename T>
concept callable = requires (T f) { f(); }; // `requires` expression.

template <typename T>
  requires requires (T x) { x + x; } // `requires` clause and expression on same line.
T add(T a, T b) {
  return a + b;
}
```

Requires expressions:

```
template<typename T>
concept Addable = requires (T x) { x + x; }; // requires-expression

template<typename T> requires Addable<T> // requires-clause, not requires-expression
T add(T a, T b) { return a + b; }

template<typename T>
    requires requires (T x) { x + x; } // ad-hoc constraint, note keyword used twice
T add(T a, T b) { return a + b; }
```

The syntax of requires-expression is as follows:

```
requires { requirement-seq }
requires ( parameter-list(optional) ) { requirement-seq }
```

Each requirement in the requirements-seq is one of the following:

1)simple requirement:

Simple requirements - asserts that the given expression is valid.

```
template <typename T>
concept callable = requires (T f) { f(); };
```

2) type requirements:

Type requirements - denoted by the typename keyword followed by a type name, asserts that the given type name is valid.

```
struct foo {
  int foo;
};

struct bar {
  using value = int;
  value data;
};

struct baz {
  using value = int;
  value data;
```

};

//SFINAE:- "Substitution Failure Is Not An Error"

```
// Using SFINAE, enable if `T` is a `baz`.
template <typename T, typename = std::enable_if_t<std::is_same_v<T, baz>>>
struct S {};

template <typename T>
using Ref = T&;

template <typename T>
concept C = requires {
                    // Requirements on type `T`:
  typename T::value; // A) has an inner member named `value`
  typename S<T>;     // B) must have a valid class template specialization for `S`
  typename Ref<T>;   // C) must be a valid alias template substitution
};

template <C T>
void g(T a);

g(foo{}); // ERROR: Fails requirement A.
g(bar{}); // ERROR: Fails requirement B.
g(baz{}); // PASS.
```

3) compound requirements:

Compound requirements - an expression in braces followed by a trailing return type or type constraint.

```
template <typename T>
concept C = requires(T x) {
  {*x} -> typename T::inner; // the type of the expression `*x` is convertible to `T::inner`
  {x + 1} -> std::same_as<int>; // the expression `x + 1` satisfies `std::same_as<decltype((x + 1))>`
  {x * 1} -> T; // the type of the expression `x * 1` is convertible to `T`
};
```

4)nested requirements:

Nested requirements - denoted by the requires keyword, specify additional constraints (such as those on local parameter arguments).

```
template <typename T>
concept C = requires(T x) {
  requires std::same_as<sizeof(x), size_t>;
};
```

C++ keywords: requires:

specifies a constant expression on template parameters that evaluate a requirement(since C++20)
in a template declaration, specifies an associated constraint (since C++20)

Concept Library:

Concepts are also provided by the standard library for building more complicated concepts. Some of these include:

Core language concepts:

- same_as - specifies two types are the same.
- derived_from - specifies that a type is derived from another type.
- convertible_to - specifies that a type is implicitly convertible to another type.
- common_with - specifies that two types share a common type.
- integral - specifies that a type is an integral type.
- default_constructible - specifies that an object of a type can be default-constructed.

Comparison concepts:

- boolean - specifies that a type can be used in Boolean contexts.
- equality_comparable - specifies that operator== is an equivalence relation.

Object concepts:

- movable - specifies that an object of a type can be moved and swapped.
- copyable - specifies that an object of a type can be copied, moved, and swapped.
- semiregular - specifies that an object of a type can be copied, moved, swapped, and default constructed.
- regular - specifies that a type is regular, that is, it is both semiregular and equality_comparable.

Callable concepts:

- invocable - specifies that a callable type can be invoked with a given set of argument types.
- predicate - specifies that a callable type is a Boolean predicate.

Ranges library:

It is defined in header <ranges>

```
namespace std {
   namespace views = ranges::views;
}
```

The namespace alias std::views is provided as a shorthand for std::ranges::views.

Ranges are essentially iterators that cover a sequence of values in collections such as lists or vectors, but instead of constantly dragging the beginning and end of the iterator around, ranges just keep them around internally.

Just as Concepts, Ranges have also moved from experimental state to the language standard in C++20, as Ranges depend on Concepts and uses them to improve the old iterator handling by making it possible to add constraints to the handled values, with the same benefits.

On top of constraining value types, Ranges introduce Views as a special form of a range, which allows data manipulation or filtering on a range, returning a modified version of the initial range's data as yet another range. This allows them to be chained together. Say you have a vector of integers and you want to retrieve all even values in their squared form — ranges and views can get you there.

With all of these changes, the compiler will be of a lot more assistance for type checking and will present more useful error messages.

The ranges library is the first customer of concepts. It supports algorithms which can operate directly on the container; you don't need iterators to specify a range can be evaluated lazily can be composed

To make it short: The ranges library supports functional patterns. The following functions show function composition with the pipe symbol.

```
#include <vector>
#include <ranges>
#include <iostream>

int main(){
  std::vector<int> ints{0, 1, 2, 3, 4, 5};
  auto even = [](int i){ return 0 == i % 2; };
  auto square = [](int i) { return i * i; };

  for (int i : ints | std::view::filter(even) |
                      std::view::transform(square)) {
    std::cout << i << ' ';              // 0 4 16
  }
}
```

even is a lambda function which returns if a i is even and the lambda function **square** maps i to its square. The rest ist function composition which you have to read from left to right: for (int i : ints | std::view::filter(even) | std::view::transform(square)).

Apply on each element of ints the filter even and map each remaining element to its square.

Type classification:

Objects, references, functions including function template specializations, and expressions have a property called type.

The C++ type system consists of the following types:
----

fundamental types (see also std::is_fundamental):

- the type void (see also std::is_void);
- the type std::nullptr_t (since C++11) (see also std::is_null_pointer);
- arithmetic types (see also std::is_arithmetic):
  - floating-point types (float, double, long double) (see also std::is_floating_point);
  - integral types (see also std::is_integral):
    - the type bool;
    - character types:

            - narrow character types:
                - ordinary character types (char, signed char, unsigned char)
                - the type char8_t (since C++20)
            - wide character types (char16_t (since C++11), char32_t (since C++11), wchar_t);
        - signed integer types (short int, int, long int, long long int);
        - unsigned integer types (unsigned short int, unsigned int, unsigned long int, unsigned long long int);

<u>compound types (see also std::is_compound):</u>

- reference types (see also std::is_reference):
    - lvalue reference types (see also std::is_lvalue_reference):
        - lvalue reference to object types;
        - lvalue reference to function types;
    - rvalue reference types (see also std::is_rvalue_reference):
        - rvalue reference to object types;
        - rvalue reference to function types;
    - pointer types (see also std::is_pointer):
        - pointer-to-object types;
        - pointer-to-function types;
    - pointer-to-member types (see also std::is_member_pointer):
        - pointer-to-data-member types (see also std::is_member_object_pointer);
        - pointer-to-member-function types (see also std::is_member_function_pointer);
- array types (see also std::is_array);
- function types (see also std::is_function);
- enumeration types (see also std::is_enum);
- class types:
    - non-union types (see also std::is_class);
    - union types (see also std::is_union).

<u>Some examples of integral type:</u>

std::is_integral
Defined in header <type_traits>
template< class T >
struct is_integral;

Checks whether T is an integral type. Provides the member constant value which is equal to true, if T is the type bool, char, char8_t, char16_t, char32_t, wchar_t, short, int, long, long long, or any implementation-defined extended integer types, including any signed, unsigned, and cv-qualified variants. Otherwise, value is equal to false.

class A {};

enum E : int {};

```
int main()
{
    std::cout << std::boolalpha;
    std::cout << std::is_integral<A>::value << '\n';
    std::cout << std::is_integral<E>::value << '\n';
    std::cout << std::is_integral<float>::value << '\n';
    std::cout << std::is_integral<int>::value << '\n';
    std::cout << std::is_integral<bool>::value << '\n';
}
```

Output:

```
false
false
false
true
true
```

LIke this, You can see below thing:
----
integral (C++20) : specifies that a type is an integral type
is_integer() identifies integer types
is_floating_point() : checks if a type is a floating-point type
is_arithmetic(): checks if a type is an arithmetic type

CONSTEVAL:

A new keyword that specifies an immediate function – functions that produce constant values, at compile time only. In contrast to constexpr function, they cannot be called at runtime.

```
consteval int add(int a, int b) { return a+b; }
constexpr int r = add(100, 300);
```

Immediate functions also called as consteval.

Similar to constexpr functions, but functions with a consteval specifier must produce a constant. These are called immediate functions.

```
consteval int sqr(int n) {
  return n * n;
}

constexpr int r = sqr(100); // OK
int x = 100;
int r2 = sqr(x); // ERROR: the value of 'x' is not usable in a constant expression
                 // OK if `sqr` were a `constexpr` function
```

Example:
========

```
consteval int sqr(int n) {
   return n * n;
}
```

consteval creates a so-called immediate function. Each invocation of an immediate function creates a compile-time constant. A consteval (immediate) function is executed at compile-time.consteval cannot be applied to destructors or functions which allocate or deallocate. As all these happend at runtime.
You can only use at most one of consteval, constexpr, or constinit specifier in a declaration.
An immediate function (consteval) is implicit inline and has to fulfill the requirements of a constexpr function.

So, a consteval function is guaranteed to be evaluated at compile time.
A constexpr function, on the other hand, can be called at either run time or compile time.
So all consteval function are a constexpr function But reverse are not true.
A consteval function can only invoke a constexpr function but not the other way around

some more example:

```
constexpr int foo(int factor) {
   return 123 * factor;
}

const int const_factor = 10;
int non_const_factor = 20;

const int first = foo(const_factor);
const int second = foo(non_const_factor);
```

Here, first will be evaluated at compile time as all expressions and values involved are constants and as such known at compile time, while second will be evaluated at run time since non_const_factor itself is not a constant.

If we declare foo() as conseval,, then It will cause error at below line.
const int second = foo(non_const_factor); // error.
So we can say that consteval functions as an alternative for macro functions.

```
// constevalSqr.cpp

#include <iostream>

consteval int sqr(int n) {
   return n * n;
}

int main() {

   std::cout << "sqr(5): " << sqr(5) << std::endl;     // (1)

   const int a = 5;                                    // (2)
   std::cout << "sqr(a): " << sqr(a) << std::endl;
```

```
    int b = 5;                                  // (3)
    // std::cout << "sqr(b): " << sqr(b) << std::endl; ERROR

}
```

constinit:

constinit can be applied to variables with static storage duration or thread storage duration.
constinit ensures for this kind of variable (static storage duration or thread storage duration) that they are initialized at compile-time. constinit cannot be used together with constexpr or consteval. If the decorated variable is not initialized at compile-time, the program is ill-formed (i.e. does not compile).
Using constinit ensures that the variable is initialized at compile-time, and that the static initialization order fiasco cannot take place.

A constexpr or const declared variable can be created as a local but a constinit declared variable not.

```
// constexprConstinit.cpp

#include <iostream>

constexpr int constexprVal = 1000;
constinit int constinitVal = 1000;

int incrementMe(int val){ return ++val;}

int main() {

    auto val = 1000;
    const auto res = incrementMe(val);                          // (1)
    std::cout << "res: " << res << std::endl;

    // std::cout << "res: " << ++res << std::endl;              ERROR (2)
    // std::cout << "++constexprVal++: " << ++constexprVal << std::endl; ERROR (2)
    std::cout << "++constinitVal++: " << ++constinitVal << std::endl;      // (3)

    constexpr auto localConstexpr = 1000;                       // (4)
    // constinit auto localConstinit = 1000; ERROR

}
```

STD::span:

// std::span
//! A span is a view (i.e. non-owning) of a container providing bounds-checked access to a contiguous group of elements.
//! Since views do not own their elements they are cheap to construct and copy --
//! a simplified way to think about views is they are holding references to their data. Spans can be dynamically-sized or fixed-sized.

```
void f(std::span<int> ints) {
        std::for_each(ints.begin(), ints.end(), [](auto i) {
                cout << " i :" << i << endl;
        });
}

        std::vector<int> v = { 1, 2, 3 };
        f(v);
```

Example:
As opposed to maintaining a pointer and length field, a span wraps both of those up in a single container.

```
constexpr size_t LENGTH_ELEMENTS = 3;
int* arr = new int[LENGTH_ELEMENTS]; // arr = {0, 0, 0}

// Fixed-sized span which provides a view of `arr`.
std::span<int, LENGTH_ELEMENTS> span = arr;
span[1] = 1; // arr = {0, 1, 0}

// Dynamic-sized span which provides a view of `arr`.
std::span<int> d_span = arr;
span[0] = 1; // arr = {1, 1, 0}
constexpr size_t LENGTH_ELEMENTS = 3;
int* arr = new int[LENGTH_ELEMENTS];

std::span<int, LENGTH_ELEMENTS> span = arr; // OK
std::span<double, LENGTH_ELEMENTS> span2 = arr; // ERROR
std::span<int, 1> span3 = arr; // ERROR
```

three-way comparison operator:

A three-way comparison is a function that will give the entire relationship in one query. Traditionally, strcmp() is such a function.
Given two strings it will return an integer where < 0 means the first string is less, == 0 if both are equal and > 0 if the first string is greater.
It can give one of three results, hence it’s a three-way comparison.

C++20 — have a comparison operator that does a three-way comparison. It is commonly spelled <=> as it gives the result of <, == and > simultaneously.

And as <=> sort of looks like a spaceship, it is called the “spaceship operator”.

There’s a new three-way comparison operator, <=>. The expression a <=> b returns an object that compares <0 if a < b, compares >0 if a > b, and
compares ==0 if a and b are equal/equivalent.

lhs <=> rhs   (1)
The expression returns an object that

compares <0 if lhs < rhs
compares >0 if lhs > rhs
and compares ==0 if lhs and rhs are equal/equivalent.

The advantage is :

advantage of a three-way comparison over the mathematical relation is simple:
nstead of doing the whole !(a < b) && !(b < a) or a <= b && b <= a dance to figure out whether two elements are equal, you can just ask that directly. And the user still needs to write only one predicate.

example:
You can define the three-way comparison operator or request it from the compiler with =default.
In both cases you get all six comparison operators: ==, !=, <, <=, >, and >=.

```
// threeWayComparison.cpp

#include <compare>
#include <iostream>

struct MyInt {
   int value;
   explicit MyInt(int val): value{val} { }
   auto operator<=>(const MyInt& rhs) const {          // (1)
      return value <=> rhs.value;
   }
};

struct MyDouble {
   double value;
   explicit constexpr MyDouble(double val): value{val} { }
   auto operator<=>(const MyDouble&) const = default;   // (2)
};

template <typename T>
constexpr bool isLessThan(const T& lhs, const T& rhs) {
   return lhs < rhs;
}
```

```
int main() {

   std::cout << std::boolalpha << std::endl;

   MyInt myInt1(2011);
   MyInt myInt2(2014);

   std::cout << "isLessThan(myInt1, myInt2): "
            << isLessThan(myInt1, myInt2) << std::endl;

   MyDouble myDouble1(2011);
   MyDouble myDouble2(2014);

   std::cout << "isLessThan(myDouble1, myDouble2): "
            << isLessThan(myDouble1, myDouble2) << std::endl;

   std::cout << std::endl;

}
```

2)

The Compiler-Generated Spaceship Operator

The compiler-generated three-way comparison operator needs the header <compare>, is implicit constexpr and noexcept. Additionally,
it performs a lexicographical comparison. What? Let me start with constexpr

```
// threeWayComparisonAtCompileTime.cpp

#include <compare>
#include <iostream>

struct MyDouble {
   double value;
   explicit constexpr MyDouble(double val): value{val} { }
   auto operator<=>(const MyDouble&) const = default;
};

template <typename T>
constexpr bool isLessThan(const T& lhs, const T& rhs) {
   return lhs < rhs;
}

int main() {

   std::cout << std::boolalpha << std::endl;


   constexpr MyDouble myDouble1(2011);
```

```
    constexpr MyDouble myDouble2(2014);

    constexpr bool res = isLessThan(myDouble1, myDouble2); // (1)

    std::cout << "isLessThan(myDouble1, myDouble2): "
              << res << std::endl;

    std::cout << std::endl;

}
```

3)

```
#include <compare>
#include <iostream>

int main() {
    double foo = -0.0;
    double bar = 0.0;

    auto res = foo <=> bar;

    if (res < 0)
        std::cout << "-0 is less than 0";
    else if (res == 0)
        std::cout << "-0 and 0 are equal";
    else if (res > 0)
        std::cout << "-0 is greater than 0";
}
```

Designated Initializers:

Explicit member names in the initializer expression. C-style designated initializer syntax.
Any member fields that are not explicitly listed in the designated initializer list are default-initialized.

example 1:

```
struct A {
  int x;
  int y;
  int z = 123;
};

A a {.x = 1, .z = 2}; // a.x == 1, a.y == 0, a.z == 2
```

example 2:
==========

```
struct S {
            int a;
            int b;
            int c;
};
```

```
S test {
.a = 1,
.b = 10,
.c = 2
};
```

Range-based for loop with initializer:

This feature simplifies common code patterns, helps keep scopes tight, and offers an elegant solution to a common lifetime problem.
Create another variable in the scope of the for loop:

```
for (auto v = std::vector{1, 2, 3}; auto& e : v) {
  std::cout << e;
}
// prints "123"
```

Before the c++20:
=================

```
{
  T thing = f();
  for (auto& x : thing.items()) {
    // Note: “for (auto& x : f().items())” is WRONG
    mutate(&x);
    log(x);
  }
}
```

With the proposal c++20

```
for (T thing = f(); auto& x : thing.items()) {
  mutate(&x);
  log(x);
}
```

Before the c++20:

```
{
  std::size_t i = 0;
  for (const auto& x : foo()) {
    bar(x, i);
    ++i;
  }
}
```

With the proposal c++20:

```
for (std::size_t i = 0; const auto& x : foo()) {
  bar(x, i);
  ++i;
}
```

C++20 Attributes:

Purpose of attributes in C++11:
Lets recap the purpose of attributes in c++ first:

Basically it is used to enforce constraint on the code itself. Here constraint refers to a condition, that the arguments of a particular function must meet for its execution (precondition). In previous versions of C++, the code for specifying constraints was written in this manner.

```
int f(int i)
{
   if (i > 0)
      return i;
   else
      return -1;

   // Code
}
```

<u>Using atributes:</u>

1)
```
int f(int i)[[expects:i > 0]]
{
   // Code
}
```

2)
```
int f(int i)
{
   switch (i) {
   case 1:
      [[fallthrough]];
      [[likely]] case 2 : return 1;
   }
   return -1;
}
```

When the statement is preceded by likely compiler makes special optimizations with respect to that statement which improves the overall performance of the code.
Some examples of such attributes are [carries_dependency], [likely], [unlikely].

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

```
int main()
{

    // Set debug mode in compiler or 'R'
    [[maybe_unused]] char mg_brk = 'D';

    // Compiler does not emit any warnings
    // or error on this unused variable
}
```

If we dont use "maybe_unused", then we will ger below warning from compiler.

```
main.cpp: In function 'int main()':
main.cpp:8:11: warning: unused variable 'mg_brk' [-Wunused-variable]
    8 |     char mg_brk = 'D';
```

C++20 attributes:

likely: For optimisation of certain statements that have more probability to execute than others.
Likely is now available in latest version of GCC compiler for experimentation purposes.

```
int f(int i)
{
    switch (i) {
    case 1:
        [[fallthrough]];
        [[likely]] case 2 : return 1;
    }
    return 2;
}
```

no_unique_address:

Indicates that this data member need not have an address distinct from all other non-static data members of its class. This means that
if the class consist of an empty type then the compiler can perform empty base optimisation on it.

```
// empty class ( No state!)
struct Empty {
};

struct X {
    int i;
    Empty e;
};

struct Y {
    int i;
    [[no_unique_address]] Empty e;
};
```

```
int main()
{
    // the size of any object of
    // empty class type is at least 1
    static_assert(sizeof(Empty) >= 1);

    // at least one more byte is needed
    // to give e a unique address
    static_assert(sizeof(X) >= sizeof(int) + 1);

    // empty base optimization applied
    static_assert(sizeof(Y) == sizeof(int));
}
```

likely and unlikely attributes

Provides a hint to the optimizer that the labelled statement is likely/unlikely to have its body executed.

```
int random = get_random_number_between_x_and_y(0, 3);
[[likely]] if (random > 0) {
  // body of if statement
  // ...
}

[[unlikely]] while (unlikely_truthy_condition) {
  // body of while statement
  // ...
}
```

Deprecate implicit capture of this:

//! http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0806r2.html

This inconsistency in defaults is confusing. Users may well know that there exists an inconsistency, but it is much harder to know which way round the inconsistency goes.
For this reason, we propose that users should never rely on implicit capture of *this via a [=]-capture-default.
(The implicit capture of *this via [&] continues to be idiomatic.)

```
[=] → [=,  this]: local variables by value, class members by reference
[=] → [=, *this]: everything by value
[&] → [&,  this]: everything by reference
[&] → [&, *this]: (this would be unusual)
```

Before it was like below:

```
struct Foo {
  int n = 0;
  auto f(int a) {
    return [=](int k) { return n + a * k; };
  }
};
```

After c++20:

```
struct Foo {
  int n = 0;
  auto f(int a) {
    return [=,*this](int k) { return n + a * k; };
  }
};
```

some more examaple:

```
struct int_value {
  int n = 0;
  auto getter_fn() {
    // BAD:
    // return [=]() { return n; };

    // GOOD:
    return [=, *this]() { return n; };
  }
};
```

Class types in non-type template parameters:

A non-type template argument provided within a template argument list is an expression whose value can be determined at compile time.
Such arguments must be constant expressions, addresses of functions or objects with external linkage, or addresses of static class members.
Non-type template arguments are normally used to initialize a class or to specify the sizes of class members.

Non-type parameters

A template non-type parameter is a special type of parameter that does not substitute for a type, but is instead replaced by a value. A non-type parameter can be any of the following:

- A value that has an integral type or enumeration
- 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 of Non type template parameter.

```
#include <iostream>

template <class T, int size> // size is the 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];
   }
};

// Showing how a function for a class with a non-type parameter is defined outside of the class
template <class T, int 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';
         return 0;
}
```

In the following example, a class template is defined that requires a non-type template int argument as well as the type argument:

```
template<class T, int size> class Myfilebuf
{
     T* filepos;
     static int array[size];
public:
     Myfilebuf() { /* ... */ }
     ~Myfilebuf();

};
```

In this example, the template argument size becomes a part of the template class name. An object of such a template class is created with both the type argument T of the class and the value of the non-type template argument size.

An object x, and its corresponding template class with arguments double and size=200, can be created from this template with a value as its second template argument:

```
Myfilebuf<double,200> x;
```

x can also be created using an arithmetic expression:

```
Myfilebuf<double,10*20> x;
```

The objects created by these expressions are identical because the template arguments evaluate identically. The value 200 in the first expression could have been represented by an expression whose result at compile time is known to be equal to 200, as shown in the second construction.

One more classic example:

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

struct foo {
   int aa=300;
          foo() = default;
          constexpr foo( int a) {
              aa=a;
          }
};

template <foo f>
auto get_foo() {
   cout<<f.aa<<"\n";;
          return f;
}

int main()
{

get_foo<foo{}>(); // uses implicit constructor
get_foo < foo{ 123 } > ();

}
```

constexpr virtual functions:

Virtual functions can now be constexpr and evaluated at compile-time. constexpr virtual functions can override non-constexpr virtual functions and vice-versa.

```
struct X1 {
  virtual int f() const = 0;
};

struct X2: public X1 {
  constexpr virtual int f() const { return 2; }
};

struct X3: public X2 {
  virtual int f() const { return 3; }
};

struct X4: public X3 {
  constexpr virtual int f() const { return 4; }
};

constexpr X4 x4;
x4.f(); // == 4
```

explicit(bool):

Conditionally select at compile-time whether a constructor is made explicit or not. explicit(true) is the same as specifying explicit.

```
struct foo {
  // Specify non-integral types (strings, floats, etc.) require explicit construction.
  template <typename T>
  explicit(!std::is_integral_v<T>) foo(T) {}
};

foo a = 123; // OK
foo b = "123"; // ERROR: explicit constructor is not a candidate (explicit specifier evaluates to true)
foo c {"123"}; // OK
```

using enum:

Motivation:
The single biggest deterrent to use of scoped enumerations is the inability to associate them with a using directive. Bring an enum's members into scope to improve readability.

See the problem first in previous release:

```
enum class CatState
{
    sleeping,
    napping,
    resting
};
```

I would like to use something equivalent to using namespace X so that I don't need to prefix all my state names with CatState::. In other words, I'd like to use sleeping instead of CatState::sleeping.

This above problem solved in C++20 using enum.

Before:

```
enum class rgba_color_channel { red, green, blue, alpha };

std::string_view to_string(rgba_color_channel channel) {
  switch (channel) {
    case rgba_color_channel::red:   return "red";
    case rgba_color_channel::green: return "green";
    case rgba_color_channel::blue:  return "blue";
    case rgba_color_channel::alpha: return "alpha";
  }
}
```

After:

```
enum class rgba_color_channel { red, green, blue, alpha };

std::string_view to_string(rgba_color_channel my_channel) {
  switch (my_channel) {
    using enum rgba_color_channel;
    case red:   return "red";
    case green: return "green";
    case blue:  return "blue";
    case alpha: return "alpha";
  }
}
```

C++20 enum one more example:

```
enum class CatState
{
    sleeping,
    napping,
    resting
};

std::string getPurr(CatState state)
{
    switch (state)
    {
        using enum CatState;
        // our states are accessible without the scope operator from now on

        case sleeping:     return {};     // instead of "case CatState::sleeping:"
        case napping:      return "purr";
        case resting:      return "purrrrrr";
    }
}
```

https://stackoverflow.com/questions/9450338/equivalent-of-using-namespace-x-for-scoped-enumerations
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1099r4.html

STRING FORMATTING

Speaking of error messages, or well, output in general, C++20 lannguage standard has std::format. Essentially this provides Python's string formatting functionality! Compared to the whole clumsiness of the cout shifting business, and the fact that using printf() in the context of C++ just feeling somewhat wrong, this is definitely a welcomed addition.

While the Python style formatting offers pretty much the same functionality as printf(), just in a different format string syntax, it eliminates a few redundancies and offers some useful additions, such as binary integer representation, and centered output with or without fill characters. However, the biggest advantage is the possibility to define formatting rules for custom types, on the surface this is like Python's __str__() or Java's toString() methods, but it also adds custom formatting types along the way.

Take strftime() as example — albeit it this is a C function, which behaves as snprintf(), the difference is that it defines custom, time-specific conversion characters for its format string, and expects struct tm as argument. With the right implementation, std::format could be extended to behave just like that, which is in fact what the upcoming addition to the std::chrono library is going to do.

Parameters
fmt - string view representing the format string. The format string consists of
ordinary characters (except { and }), which are copied unchanged to the output,
escape sequences {{ and }}, which are replaced with { and } respectively in the output, and
replacement fields.
Each replacement field has the following format:

introductory { character;
(optional) arg-id, a non-negative number;
(optional) a colon (:) followed by a format specification;
final } character.

args... - arguments to be formatted
loc - std::locale used for locale-specific formatting
Return value
A string object holding the formatted result.

examle:

```
std::string message = std::format("The answer is {}.", 42);

std::format("{} {}!", "Hello", "world", "something"); // OK, produces "Hello world!"

#include <iostream>
#include <format>

int main() {
    std::cout << std::format("Hello {}!\n", "world");
}
```

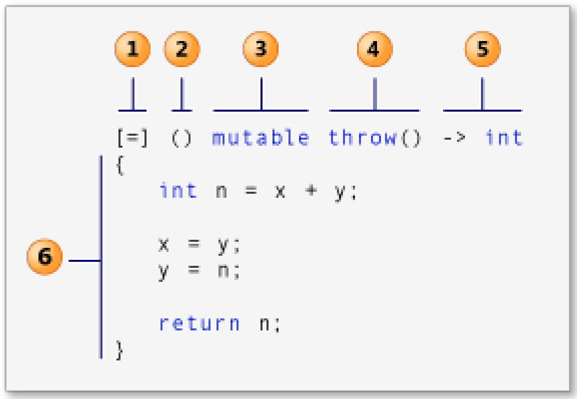
<u>std::format_to:</u>

```
#include <format>
#include <iostream>
#include <iterator>
#include <string>

auto main() -> int
{
    std::string buffer;

    std::format_to(
        std::back_inserter(buffer), //< OutputIt
        "Hello, C++{}!\n",          //< fmt
        "20");                      //< arg
    std::cout << buffer;
    buffer.clear();

    std::format_to(
        std::back_inserter(buffer), //< OutputIt
        "Hello, {0}::{1}!{2}",      //< fmt
        "std",                      //< arg {0}
        "format_to()",              //< arg {1}
        "\n",                       //< arg {2}
        "extra param(s)...");       //< unused
    std::cout << buffer;

    std::wstring wbuffer;
    std::format_to(
        std::back_inserter(wbuffer),//< OutputIt
        L"Hello, {2}::{1}!{0}",     //< fmt
        L"\n",                      //< arg {0}
        L"format_to()",             //< arg {1}
        L"std",                     //< arg {2}
        L"...is not..."             //< unused
        L"...an error!");           //< unused
    std::wcout << wbuffer;
}
```

<u>Output:</u>

```
Hello, C++20!
Hello, std::format_to()!
Hello, std::format_to()!
```

`[=, this]` as a lambda capture or lambda init-capture



```
[=] () mutable throw() -> int
{
   int n = x + y;

   x = y;
   y = n;

   return n;
}
```

1. *capture clause* (Also known as the *lambda-introducer* in the C++ specification.)
2. *parameter list* Optional. (Also known as the *lambda declarator*)
3. *mutable specification* Optional.
4. *exception-specification* Optional.
5. *trailing-return-type* Optional.
6. *lambda body*.

You can use the default capture mode (*capture-default* in the Standard syntax) to indicate how to capture any outside variables that are referenced in the lambda: [&] means all variables that you refer to are captured by reference, and [=] means they are captured by value. You can use a default capture mode, and then specify the opposite mode explicitly for specific variables. For example, if a lambda body accesses the external variable total by reference and the external variable factor by value, then the following capture clauses are equivalent:
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]

Allow `[=, this]` as a lambda capture in C++20;

```
struct Baz {
    auto foo() {
        return [=] { std::cout << s << std::endl; };
    }
    std::string s;
};
```

GCC 9:
warning: implicit capture of 'this' via '[=]' is deprecated in C++20
If you really need to capture this you have to write [=, this].

Bit operations

C++20 provides a new <bit> header which provides some bit operations including popcount.

```
std::popcount(0u); // 0
std::popcount(1u); // 1
std::popcount(0b1111'0000u); // 4
```

Math constants

Mathematical constants including PI, Euler's number, etc. defined in the <numbers> header.

```
std::numbers::pi; // 3.14159...
std::numbers::e; // 2.71828...
```

std::is_constant_evaluated

Predicate function which is truthy when it is called in a compile-time context.

```
constexpr bool is_compile_time() {
  return std::is_constant_evaluated();
}

constexpr bool a = is_compile_time(); // true
bool b = is_compile_time(); // false
```

std::make_shared supports arrays

```
auto p = std::make_shared<int[]>(5); // pointer to `int[5]`
// OR
auto p = std::make_shared<int[5]>(); // pointer to `int[5]`
```

starts_with and ends_with on strings

Strings (and string views) now have the starts_with and ends_with member functions to check if a string starts or ends with the given string.

```
std::string str = "foobar";
str.starts_with("foo"); // true
str.ends_with("baz"); // false
```

Check if associative container has element

Associative containers such as sets and maps have a contains member function, which can be used instead of the "find and check end of iterator" idiom.

```
std::map<int, char> map {{1, 'a'}, {2, 'b'}};
```

```
map.contains(2); // true
map.contains(123); // false

std::set<int> set {1, 2, 3};
set.contains(2); // true
```

std::bit_cast

A safer way to reinterpret an object from one type to another.

```
float f = 123.0;
int i = std::bit_cast<int>(f);
```

std::midpoint

Calculate the midpoint of two integers safely (without overflow).

```
std::midpoint(1, 3); // == 2
```

std::to_array

Converts the given array/"array-like" object to a std::array.

```
std::to_array("foo"); // returns `std::array<char, 4>`
std::to_array<int>({1, 2, 3}); // returns `std::array<int, 3>`

int a[] = {1, 2, 3};
std::to_array(a); // returns `std::array<int, 3>`
```

char8_t

Provides a standard type for representing UTF-8 strings.

```
char8_t utf8_str[] = u8"\u0123";
```

Lambda capture of parameter pack:

Capture parameter packs by value:

```
template <typename... Args>
auto f(Args&&... args){
   // BY VALUE:
   return [...args = std::forward<Args>(args)] {
      // ...
   };
}
```

Capture parameter packs by reference:

```
template <typename... Args>
auto f(Args&&... args){
   // BY REFERENCE:
   return [&...args = std::forward<Args>(args)] {
      // ...
```

```
    };
}
```

Some explanation of Lambda including C++20 example:

Lambda function:

What is lambda function?

A lambda function is short snippets of code that

not worth naming(unnamed, anonymous, disposable, etc. whatever you can call it),
and also not reused.

```
            [ capture list ] (parameters) -> return-type
            {
    method definition
            }
```

Usually, compiler evaluates a return type of a lambda function itself. So we don't need to specify a trailing return type explicitly i.e. -> return-type.
But in some complex cases, the compiler unable to deduce the return type and we need to specify that

Why Should We Use a Lambda Function?

```
struct print {
    void operator()(int element) {
        cout << element << endl;
    }
};

int main(void) {
    vector<int> v = {1, 2, 3, 4, 5};
    for_each(v.begin(), v.end(), print());
    return EXIT_SUCCESS;
}
```

This we can write using lambda like below:

```
int main(void) {
vector<int> v = {1, 2, 3, 4, 5};
for_each(v.begin(), v.end(), [](int element)
{
cout << element << endl; });
}
```

How Does Lambda Functions Works Internally?

```
[&i] ( ) { cout << i; }

// is equivalent to

struct anonymous {
    int &m_i;
    anonymous(int &i) : m_i(i) {}
    inline auto operator()() const {
        cout << m_i;
    }
};
```

The compiler generates unique class as above for each lambda function, which depend on compiler. Capture list will become a constructor argument in class, If you capture argument as value then corresponding type data member is created within the class.

Moreover, you can declare variable/object in the lambda function argument, which will become an argument to call operator i.e. operator().

Benefits of Using a Lambda Function:

lambda doesn't cost you performance & as fast as a normal function.
In addition, code becomes compact, structured & expressive.

Capture by Reference/Value:

```
--------------------------
int main() {
    int x = 100, y = 200;

    auto print = [&] { // Capturing everything by reference(not recommended though)
        cout << __PRETTY_FUNCTION__ << " : " << x << " , " << y << endl;
    };

    print();
    return EXIT_SUCCESS;
}
/* Output
main()::<lambda()> : 100 , 200
*/
```

In the above example, I have mentioned & in capture list. which captures variable x & y as reference. Similarly, = denotes captured by value, which will create data member of the same type within the closure and copy assignment will take place.

In addition, the parameter list is optional, you can omit the empty parentheses if you do not pass arguments to the lambda expression.

Lambda Capture List
The following table shows different use cases for the same:

[ ] ( ) { }	no captures
[=] ( ) { }	captures everything by copy(not recommended)
[&] ( ) { }	captures everything by reference(not recommended)
[x] ( ) { }	captures x by copy
[&x] ( ) { }	captures x by reference
[&, x] ( ) { }	captures x by copy, everything else by reference
[=, &x] ( ) { }	captures x by reference, everything else by copy

Passing Lambda as Parameter:

```
template <typename Functor>
void f(Functor functor) {
   cout << __PRETTY_FUNCTION__ << endl;
}

/* Or alternatively you can use this
void f(function<int(int)> functor) {
   cout << __PRETTY_FUNCTION__ << endl;
}
*/

int g() {
   static int i = 0;
   return i++;
}

int main() {
   auto lambda_func = [i = 0]() mutable { return i++; };
   f(lambda_func); // Pass lambda
   f(g);           // Pass function
}
/* Output
Function Type : void f(Functor) [with Functor = main()::<lambda(int)>]
Function Type : void f(Functor) [with Functor = int (*)(int)]
*/
```

Capture Member Variable in Lambda or This Pointer:

```
class Example {
   int m_var;
  public:
   Example() : m_var(10) {}
   void func() {
       [=]() { cout << m_var << endl; }(); // IIFE
   }
};
int main() {
   Example e;
   e.func();
   return EXIT_SUCCESS;
}
```

this pointer can also be captured using [this], [=] or [&]. In any of these cases, class data members(including private) can be accessed as you do in a normal method.

Generic Lambda(C++14)

```
const auto l = [](auto a, auto b, auto c) {};

// is equivalent to

struct anonymous {
    template <class T0, class T1, class T2>
    auto operator()(T0 a, T1 b, T2 c) const {
    }
};
```

Generic lambda introduced in C++14 which can captures parameters with auto specifier.

Variadic Generic Lambda(C++14):

```
template <typename... Args>
void print(Args &&... args) {
    (void(cout << forward<Args>(args) << endl), ...);
}

int main() {
    auto variadic_generic_lambda = [](auto &&... param) {
        print(forward<decltype(param)>(param)...);
    };

    variadic_generic_lambda(1, "lol", 1.1);
    return EXIT_SUCCESS;
}
```

Mutable Lambda Function(C++11):

Typically, a lambda’s function call operator is const-by-value which means lambda requires mutable keyword if you are capturing anything by-value.

```
[]() mutable {}
```

```
// is equivalent to

struct anonymous {
   auto operator()() { // call operator
   }
};
```

Lambda as a Function Pointer(C++11):

```
auto funcPtr = +[] {};
static_assert(is_same<decltype(funcPtr), void (*)()>::value);
```

You can force the compiler to generate lambda as a function pointer rather than closure by adding + in front of it as above.

constexpr Lambda Expression(C++17):

Since C++17, a lambda expression can be declared as constexpr.

```
constexpr auto sum = [](const auto &a, const auto &b) { return a + b; };
/*
   is equivalent to

   constexpr struct anonymous
   {
      template <class T1, class T2>
      constexpr auto operator()(T1 a, T2 b) const
      {
         return a + b;
      }
   };
*/
static_assert(sum(10, 10) == 20);
```

Even if you don’t specify constexpr , the function call operator will be constexpr anyway, if it happens to satisfy all constexpr function requirements.

Template Lambda Expression(C++20):

As we saw above in generic lambda function, we can declare parameters as auto. That in turn templatized by compiler & deduce the appropriate template type. But there was no way to change this template parameter and use real C++ template arguments. For

example:

```
template <typename T>
void f(vector<T>&    vec){
   //. . .
}
```

How do you write the lambda for the above function which takes std::vector of type T?

This was the limitation till C++17, but with C++20 it is possible as below:

```
auto f = []<typename T>(vector<T>&  vec){
   // . . .
};
std::vector<int> v;
f(v);
```

constexpr lambda expressions in C++17:

Visual Studio 2017 version 15.3 and later (available with /std:c++17): A lambda expression may be declared as constexpr or used in a constant expression when the initialization of each data member that it captures or introduces is allowed within a constant expression.

A lambda is implicitly constexpr if its result satisfies the requirements of a constexpr function:

```
auto answer = [](int n)
   {
      return 32 + n;
   };

constexpr int response = answer(10);
```

If a lambda is implicitly or explicitly constexpr, conversion to a function pointer produces a constexpr function:

```
auto Increment = [](int n)
   {
      return n + 1;
   };

constexpr int(*inc)(int) = Increment;
```

```
struct S2 { void f(int i); };
void S2::f(int i)
{
   [=]{};          // OK: by-copy capture default
   [=, &i]{};      // OK: by-copy capture, except i is captured by reference
   [=, *this]{};   // until C++17: Error: invalid syntax
                   // since c++17: OK: captures the enclosing S2 by copy
   [=, this] {};   // until C++20: Error: this when = is the default
                   // since C++20: OK, same as [=]
}

#include <iostream>

auto make_function(int& x) {
  return [&]{ std::cout << x << '\n'; };
}

int main() {
  int i = 3;
  auto f = make_function(i); // the use of x in f binds directly to i
  i = 5;
  f(); // OK; prints 5
}
```

Lambda capture of parameter pack:

Capture parameter packs by value:

```
template <typename... Args>
auto f(Args&&... args){
    // BY VALUE:
    return [...args = std::forward<Args>(args)] {
        // ...
    };
}
```

Capture parameter packs by reference:

```
template <typename... Args>
auto f(Args&&... args){
    // BY REFERENCE:
    return [&...args = std::forward<Args>(args)] {
        // ...
    };
}
```

How can I compile c++ 2020 features with an online compiler?

Ans: https://godbolt.org/

https://wandbox.org/permlink/Brw6TgAhdy89OIyj

Important c++site?

https://codereview.stackexchange.com/

Faq site for C++

https://isocpp.org/faq

```
// lambdaCaptureThis.cpp

#include <iostream>
```

```
#include <string>

struct Lambda {
   auto foo() const {
      return [=,this] { std::cout << s << std::endl; };   // (1)  c++20
   }
   std::string s = "lambda";
    ~Lambda() {
      std::cout << "Goodbye" << std::endl;
   }
};

auto makeLambda() {
   Lambda lambda;                              // (2)
   return lambda.foo();
}                                          // (3)


int main() {

   std::cout << std::endl;

   auto lam = makeLambda();
   lam();                                     // (4)

   std::cout <<" hara"<< std::endl;

}
```

How compiler interpret an user written code with examples:

1) Suppose user has written some simple hellow World code like below:

```
#include<iostream>
using namespace std;
int main()
{
   cout<<" Hello World\n";
   return 0;
}
```

Then compiler will interpreted it as like below:

```
std::operator<<(std::cout, " Hello World \n");
```

Basically It will call global <<”” operator and pass the argument.

2) =default in class constructor.

```
class Test
{
  public:
  Test()=default;
};
```

It will read it as like   inline constexpr Test() noexcept = default;

3) [](){ return 43;};

```
Int main()
{
Auto fun = [](){ return 40;};
Fun();

Return 0;
}
```

Will be interpreted like below:

```
int main()
{

  class __lambda_7_13
  {
    public:
    inline /*constexpr */ int operator()() const
    {
      return 43;
    }

    using retType_7_13 = int (*)();
    inline /*constexpr */ operator retType_7_13 () const noexcept
    {
      return __invoke;
    };

    private:
    static inline int __invoke()
    {
      return 43;
    }
  public:
  // /*constexpr */ __lambda_7_13() = default;

 };

 __lambda_7_13 fun = __lambda_7_13{};
 std::cout.operator<<(fun.operator()());
 return 0;
}
```

Compile-time initialization of a static

If you apply constinit to staticA. constinit guarantees that staticA is initialized during compile-time.

```
// sourceSIOF3.cpp
constexpr int quad(int n) {
   return n * n;
}
constinit auto staticA  = quad(5);  // (2)

// mainSOIF3.cpp
#include <iostream>
extern constinit int staticA;     // (1)
auto staticB = staticA;
int main() {
   std::cout << std::endl;
   std::cout << "staticB: " << staticB << std::endl;
   std::cout << std::endl;
}
```

(1) declares the variable staticA. staticA (2) is initialized during compile-time. By the way,
using constexpr in (1) instead of constinit is not valid, because constexpr requires a definition and not just a declaration

Spaceship operator More explanation:

Default comparisons (since C++20)
Provides a way to request the compiler to generate consistent relational operators for a class.

In brief, a class that defines operator<=> automatically gets compiler-generated operators <, <=, >, and >=. A class can define operator<=> as defaulted, in which case the compiler will also generate the code for that operator.

```
class Point {
 int x;
 int y;
public:
 auto operator<=>(const Point&) const = default;
 // ... non-comparison functions ...
};
// compiler generates all four relational operators
Point pt1, pt2;
std::set<Point> s; // ok
s.insert(pt1); // ok
if (pt1 <= pt2) { /*...*/ } // ok, makes only a single call to <=>
```

Custom comparisons

When the default semantics are not suitable, such as when the members must be compared out of order, or must use a comparison that's different from their natural comparison, then the programmer can write operator<=> and let the compiler generate the appropriate relational operators. The kind of relational operators generated depends on the return type of the user-defined operator<=>.

There are three available return types:

Return type	Operators	Equivalent values are..	Incomparable values are..
`std::strong_ordering`	== != < > <= >=	indistinguishable	not allowed
`std::weak_ordering`	== != < > <= >=	distinguishable	not allowed
`std::partial_ordering`	== != < > <= >=	distinguishable	allowed

Strong ordering

An example of a custom operator<=> that returns std::strong_ordering is an operator that compares every member of a class, except in order that is different from the default (here: last name first)

```
class TotallyOrdered : Base {
  std::string tax_id;
  std::string first_name;
  std::string last_name;
public:
 // custom operator<=> because we want to compare last names first:
 std::strong_ordering operator<=>(const TotallyOrdered& that) const {
   if (auto cmp = (Base&)(*this) <=> (Base&)that; cmp != 0)
       return cmp;
   if (auto cmp = last_name <=> that.last_name; cmp != 0)
       return cmp;
   if (auto cmp = first_name <=> that.first_name; cmp != 0)
       return cmp;
   return tax_id <=> that.tax_id;
 }
 // ... non-comparison functions ...
};
// compiler generates all four relational operators
TotallyOrdered to1, to2;
std::set<TotallyOrdered> s; // ok
s.insert(to1); // ok
if (to1 <= to2) { /*...*/ } // ok, single call to <=>
```

Note: an operator that returns a std::strong_ordering should compare every member, because if any member is left out, substitutability can be compromised: it becomes possible to distinguish two values that compare equal.

Weak ordering

An example of a custom operator<=> that returns std::weak_ordering is an operator that compares string members of a class in case-insensitive manner: this is different from the default comparison (so a custom operator is required) and it's possible to distinguish two strings that compare equal under this comparison

```
class CaseInsensitiveString {
  std::string s;
public:
  std::weak_ordering operator<=>(const CaseInsensitiveString& b) const {
    return case_insensitive_compare(s.c_str(), b.s.c_str());
  }
  std::weak_ordering operator<=>(const char* b) const {
    return case_insensitive_compare(s.c_str(), b);
  }
  // ... non-comparison functions ...
};

// Compiler generates all four relational operators
CaseInsensitiveString cis1, cis2;
std::set<CaseInsensitiveString> s; // ok
s.insert(/*...*/); // ok
if (cis1 <= cis2) { /*...*/ } // ok, performs one comparison operation

// Compiler also generates all eight heterogeneous relational operators
if (cis1 <= "xyzzy") { /*...*/ } // ok, performs one comparison operation
if ("xyzzy" >= cis1) { /*...*/ } // ok, identical semantics
```

Note that this example demonstrates the effect a heterogeneous operator<=> has: it generates heterogeneous comparisons in both directions.

Partial ordering

Partial ordering is an ordering that allows incomparable (unordered) values, such as NaN values in floating-point ordering, or, in this example, persons that are not related:

```
class PersonInFamilyTree { // ...
public:
  std::partial_ordering operator<=>(const PersonInFamilyTree& that) const {
    if (this->is_the_same_person_as ( that)) return partial_ordering::equivalent;
    if (this->is_transitive_child_of( that)) return partial_ordering::less;
    if (that. is_transitive_child_of(*this)) return partial_ordering::greater;
    return partial_ordering::unordered;
  }
  // ... non-comparison functions ...
};
// compiler generates all four relational operators
PersonInFamilyTree per1, per2;
if (per1 < per2) { /*...*/ } // ok, per2 is an ancestor of per1
else if (per1 > per2) { /*...*/ } // ok, per1 is an ancestor of per2
else if (std::is_eq(per1 <=> per2)) { /*...*/ } // ok, per1 is per2
else { /*...*/ } // per1 and per2 are unrelated
if (per1 <= per2) { /*...*/ } // ok, per2 is per1 or an ancestor of per1
if (per1 >= per2) { /*...*/ } // ok, per1 is per2 or an ancestor of per2
if (std::is_neq(per1 <=> per2)) { /*...*/ } // ok, per1 is not per2
```

Defaulted three-way comparison

The default operator<=> performs lexicographical comparison by successively comparing the base (left-to-right depth-first) and then non-static member (in declaration order) subobjects of T to compute <=>, recursively expanding array members (in order of increasing subscript), and stopping early when a not-equal result is found, that is:

```
for /*each base or member subobject o of T*/
  if (auto cmp = lhs.o <=> rhs.o; cmp != 0) return cmp;
return strong_ordering::equal; // converts to everything
```

Advantage of spaceship operator:

It's the common generalization of all other comparison operator (for totally-ordered domains): >, >=, ==, <=, < . Using <=> (spaceship), you can implement each of these other operations in a completely generic way. For strings, it's equivalent to the good old strcmp() function from the C standard library. So - useful for lexicographic order checks, such as data in vectors or lists or other ordered containers.

For integral numbers, it's what the hardware does anyway: On x86 or x86_64 Comparing a and b (CMP RAX, RBX) is basically like subtracting (SUB RAX, RBX) except that RAX doesn't actually change, only the flags are affected, so you can use "jump on equal/not equal/greater than/lesser than/etc."

(JE/JNE/JGT/JLT etc.) as the next instruction. CMP should be thought of as a "spaceship compare".

Various C++20 Links:

https://en.cppreference.com/w/cpp/utility/compare/strong_ordering

https://www.modernescpp.com/index.php/tag/c-20?start=0

https://en.wikipedia.org/wiki/C%2B%2B20

https://hackaday.com/2019/07/30/c20-is-feature-complete-heres-what-changes-are-coming/

https://www.i-programmer.info/news/184-cc/12977-c-20-feature-list-finalized.html

https://en.cppreference.com/w/cpp/20

https://www.javatpoint.com/cpp-features

https://github.com/AnthonyCalandra/modern-cpp-features/blob/master/CPP20.md

https://stackoverflow.com/questions/47466358/what-is-the-operator-in-c

The End