Modern C++

C++17

Nicolai M. Josuttis
IT-communication.com

4/18

Nicolai M. Josuttis

- **Independent consultant**
  - continuously learning since 1962
- **Systems Architect, Technical Manager**
  - finance, manufacturing, automobile, telecommunication
- **Topics:**
  - C++
  - SOA (Service Oriented Architecture)
  - Technical Project Management
  - Privacy (contributor of Enigmail)

C++17: Disclaimer

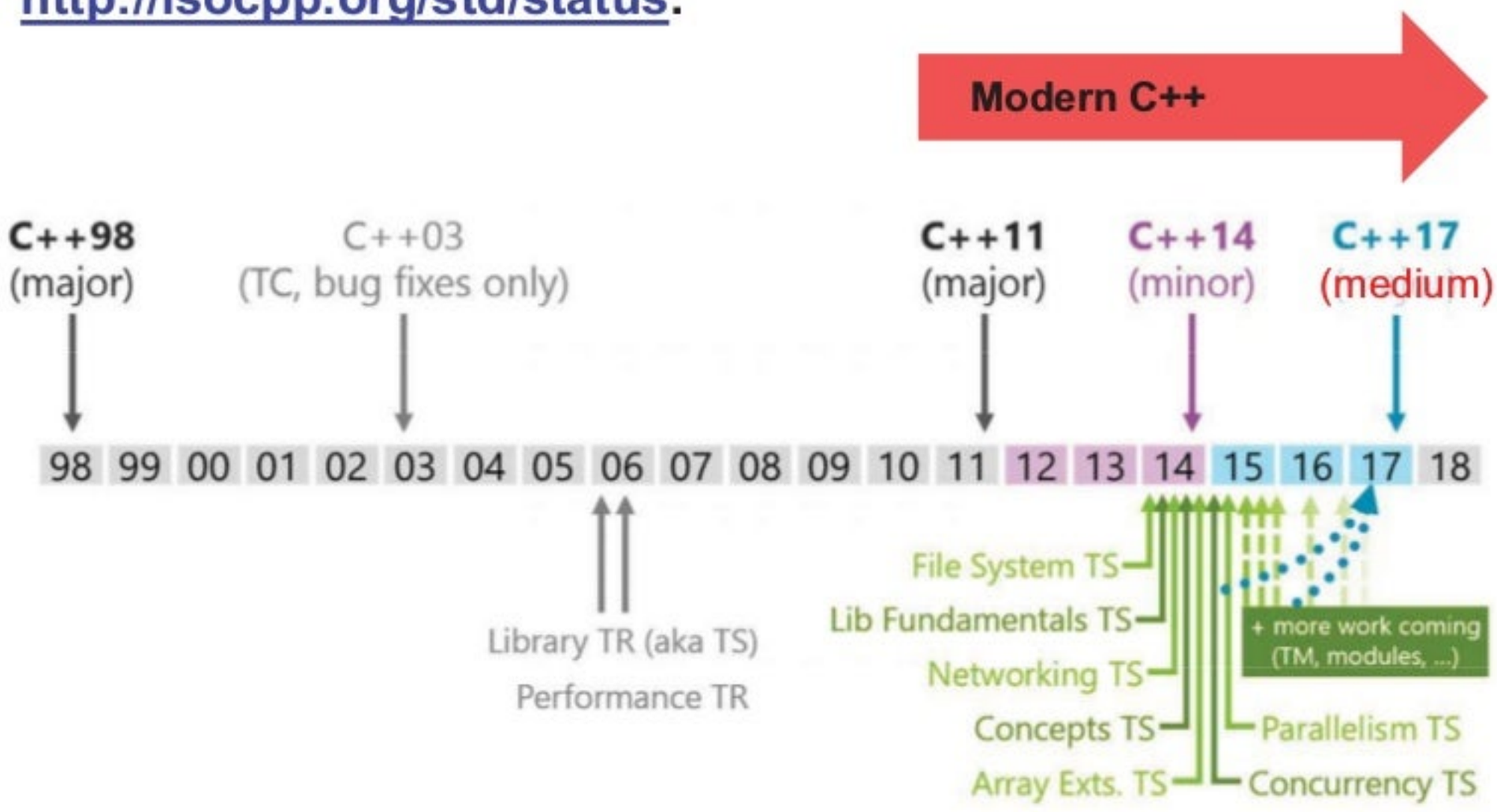
- **C++17 is brand new**
  - Limited experimental compiler support
  - We will find flaws in C++17
- **These slides are brand new**
  - You will find flaws
  - Some features are probably missing
  - Some features are not well (enough) described
  - I still learn
- **Feedback welcome!**

Modern C++

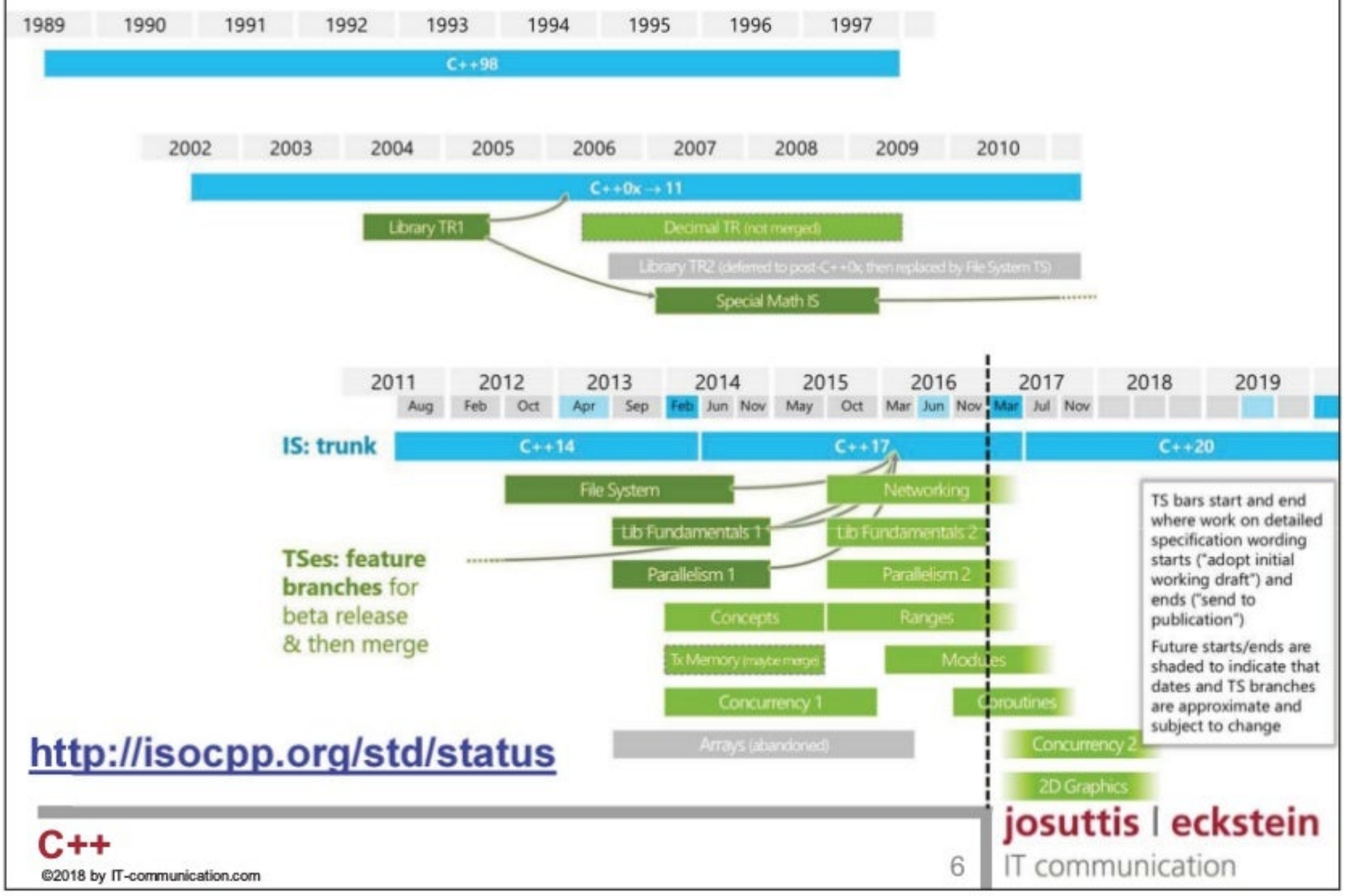
C++17

C++ Timeframe

http://isocpp.org/std/status:



C++17 Timeframe



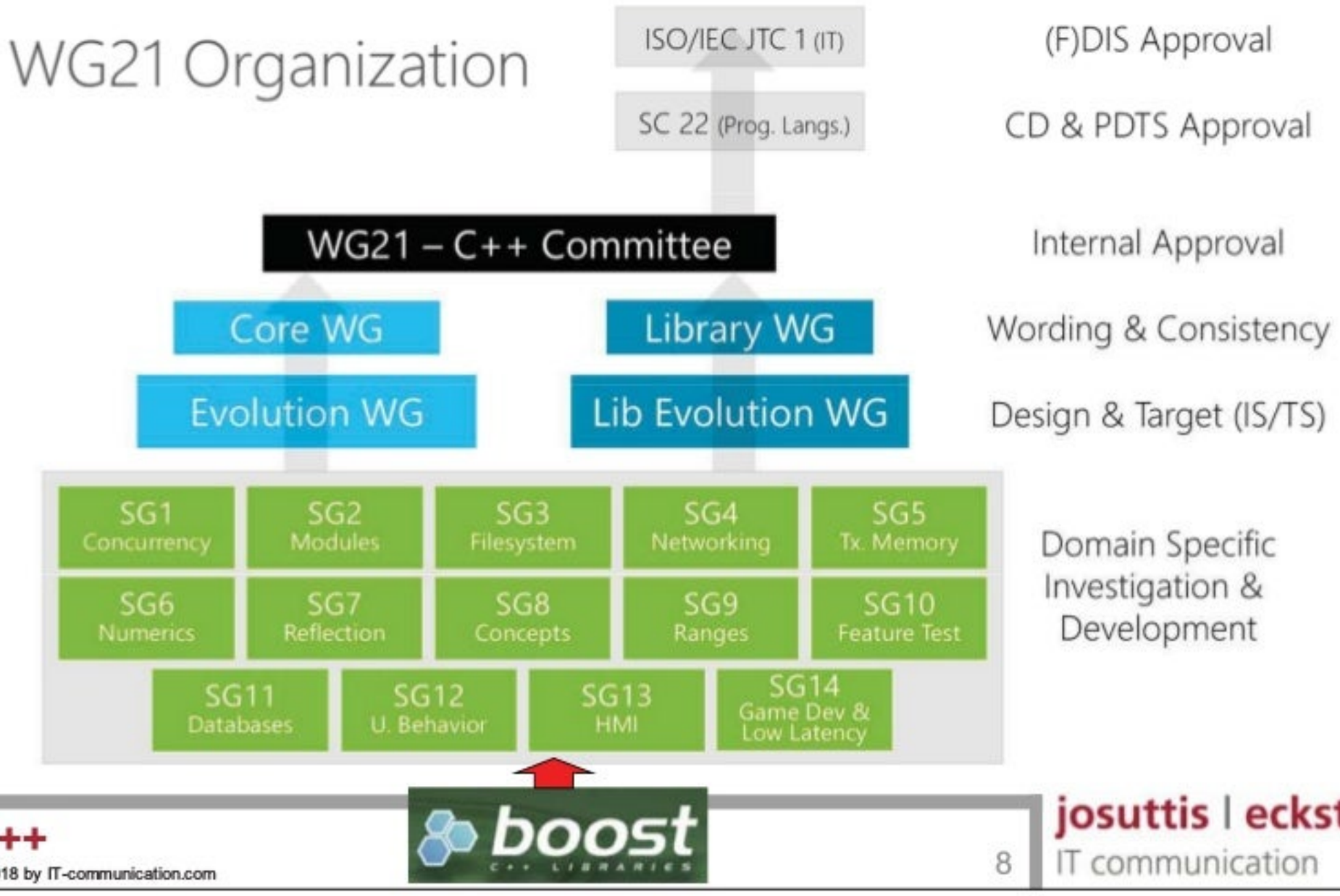
C++ Standardization

- **by ISO**
  - formal votes by national bodies, e.g.:
    - ANSI for USA
    - DIN for Germany
- **Email reflectors for different working and study groups**
  - core, library, concurrency, ...
- **2 or 3 joined meetings of ANSI and ISO per year**
  - open to the public
- **Everybody is welcome to**
  - join meetings
  - propose new features
  - discuss

- **Informal web site: http://isocpp.org/std**
- **Formal web site: http://www.open-std.org/jtc1/sc22/wg21/**

C++ Working and Study Groups

http://isocpp.org/std/the-committee:



Availability of the C++17 Standard

- **Document number:**
  - ISO/IEC **14882:2017**
- **Available at nations bodies (ANSI, DIN, ...) and INCITS**
- **Prices vary:**
  - ISO/IEC 14882:2017: $232 (members: $185.60)
  - DIN 14882:2017-12: 229,40 €
  - BS ISO/IEC 14882:2014: £630 (members: £315)
  - ...
- **Version close to final version as document N4659.pdf for free at:**
  - https://wg21.link/n4659
    - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

C++ Documents and Papers

- **All working documents are public and available**
  - https://wg21.link/
- **Different paper numbers schemes:**
  - **N***xxxx*
  - **P***yyyy***R0**, **P***yyyy***R1**, ...
- **Issue Lists:**
  - **cwg***xxx* **ewg***xxx*
  - **lwg***xxx* **lewg***xxx* **fs***xxx*
- **For example:**
  - https://wg21.link/n4659 C++17 draft standard
  - https://wg21.link/p0217r3 Proposed wording for Structured Bindings
  - https://wg21.link/lwg2911 Issue for `std::is_aggregate<>`

C++17 Support

- **gcc/g++ 7.1**
  - supports all language features
  - supports several library features
  - Option: `-std=c++17`

- **Clang 5**
  - supports all language features (clang 3.9 and 4 many)
  - Option: `-std=c++0z` or `-std:c++17`

- **Visual Studio 2017**
  - Compiler version >= 19.10 (binary compatible to VS2015)
  - supports some language features
  - frequent updates
  - Option: `/std:c++latest` or `/std:c++17`

C++17 and Visual C++

- **Visual Studio / Visual C++ adapts C++17 more and more**
  - Some C++17 features in VS2015
  - More and more C++17 features in VS2017

- **VS2017 now has compiler switch for C++ Standard version**
  - `/std:c++14`, `/std:c++17`, `/std:c++latest`

- **VS2017 is binary compatible to VS2015**
  - VS 2017 has multiple flavors (compiler version19.10.x):
    - Release channel:
      - VS2017: update 15.0, 15.1, 15.2, 15.3 (significantly more)
    - Preview channel
      - VS2017: update 15.4, preview 2 since Sep 2017

- **See:**
  - https://blogs.msdn.microsoft.com/vcblog/2017/05/10/c17-features-in-vs-2017-3/

C++17 with g++

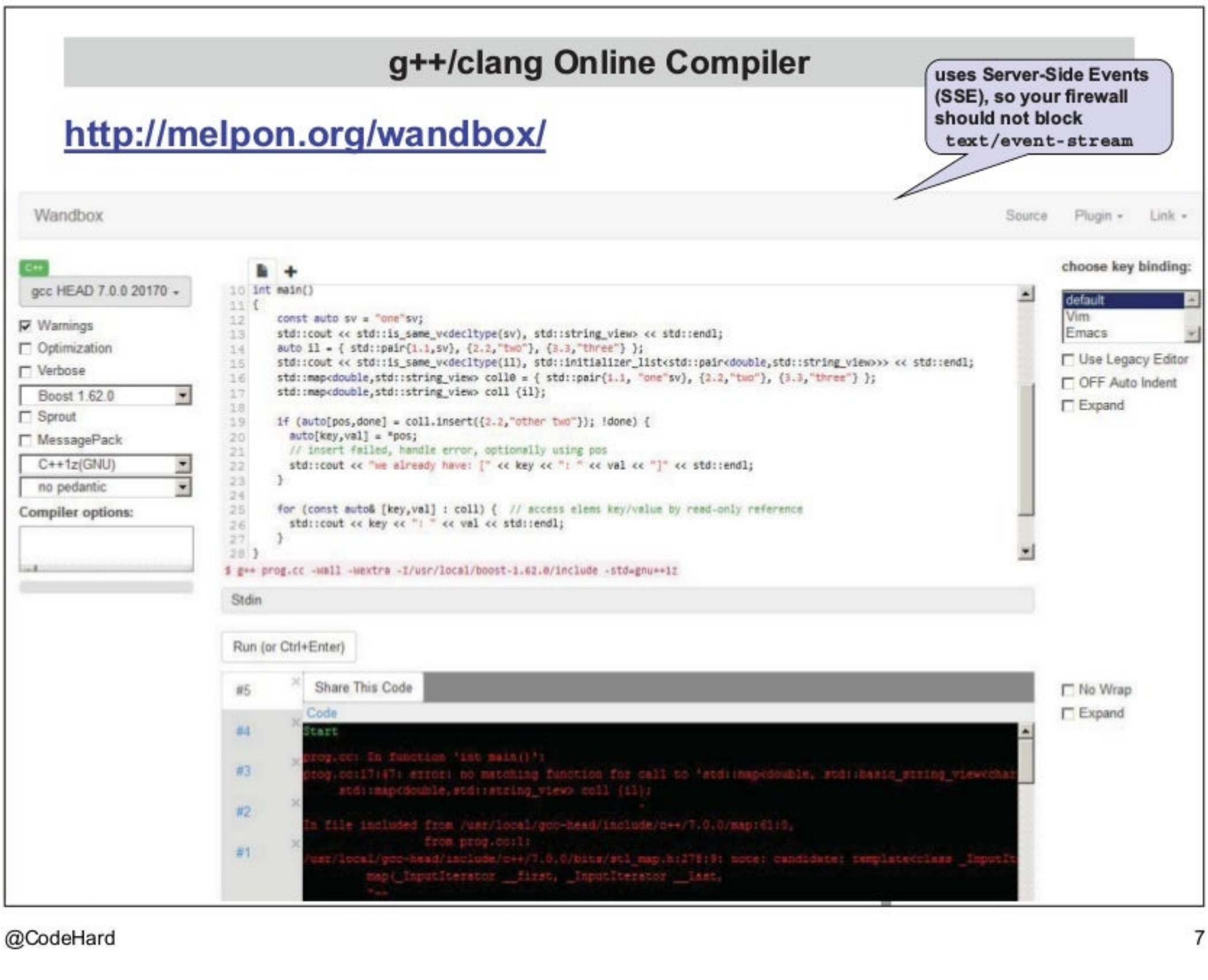
- As of 2017-04-17:

Language Feature	Proposal	Available in GCC?	SD-6 Feature Test
Removing trigraphs	N4086	5.1	
u8 character literals	N4267	6	__cpp_unicode_characters >= 201411
Folding expressions	N4295	6	__cpp_fold_expressions >= 201411
Attributes for namespaces and enumerators	N4266	4.9 (namespaces) 6 (enumerators)	__cpp_namespace_attributes >= 201411 __cpp_enumerator_attributes >= 201411
Nested namespace definitions	N4230	6	__cpp_nested_namespace_definitions >= 201411
Allow constant evaluation for all non-type template arguments	N4268	6	__cpp_nontype_template_args >= 201411
Extending static_assert	N3928	6	__cpp_static_assert >= 201411
New Rules for auto deduction from braced-init-list	N3922	5	
Allow typename in a template template parameter	N4051	5	
[[fallthrough]] attribute	P0188R1	7	__has_cpp_attribute(fallthrough)
[[nodiscard]] attribute	P0189R1	4.8 ([[gnu::warn_unused_result]]) 7 (P0189R1)	__has_cpp_attribute(nodiscard)
[[maybe_unused]] attribute	P0212R1	4.8 ([[gnu::unused]]) 7 (P0212R1)	__has_cpp_attribute(maybe_unused)
Extension to aggregate initialization	P0017R1	7	__cpp_aggregate_bases >= 201603
Wording for constexpr lambda	P0170R1	7	__cpp_constexpr >= 201603
Unary Folds and Empty Parameter Packs	P0036R0	6	__cpp_fold_expressions >= 201603
Generalizing the Range-Based For Loop	P0184R0	6	__cpp_range_based_for >= 201603
Lambda capture of *this by Value	P0018R3	7	__cpp_capture_star_this >= 201603
Construction Rules for enum class variables	P0138R2	7	
Hexadecimal floating literals for C++	P0245R1	3.0	__cpp_hex_float >= 201603
Dynamic memory allocation for over-aligned data	P0035R4	7	__cpp_aligned_new >= 201606
Guaranteed copy elision	P0135R1	7	
Refining Expression Evaluation Order for Idiomatic C++	P0145R3	7	
constexpr if	P0292R2	7	__cpp_if_constexpr >= 201606
Selection statements with initializer	P0305R1	7	
Template argument deduction for class templates	P0091R3	7	__cpp_deduction_guides >= 201606
Declaring non-type template parameters with auto	P0127R2	7	__cpp_template_auto >= 201606
Using attribute namespaces without repetition	P0028R4	7	
Ignoring unsupported non-standard attributes	P0283R2	Yes	
Structured bindings	P0217R3	7	__cpp_structured_bindings >= 201606
Remove Deprecated Use of the register Keyword	P0001R1	7	
Remove Deprecated operator++(bool)	P0002R1	7	
Make exception specifications be part of the type system	P0012R1	7	__cpp_noexcept_function_type >= 201510
__has_include for C++17	P0061R1	5	
Rewording inheriting constructors (core issue 1941 et al)	P0136R1	7	__cpp_inheriting_constructors >= 201511
Inline variables	P0386R2	7	__cpp_inline_variables >= 201606
DR 150, Matching of template template arguments	P0522R0	7	__cpp_template_template_args >= 201611
Removing dynamic exception specifications	P0003R5	7	
Pack expansions in *using-declarations*	P0195R2	7	__cpp_variadic_using >= 201611
A byte type definition	P0298R0	7	

g++/clang Online Compiler

uses Server-Side Events (SSE), so your firewall should not block `text/event-stream`

http://melpon.org/wandbox/



C++17 with VS17

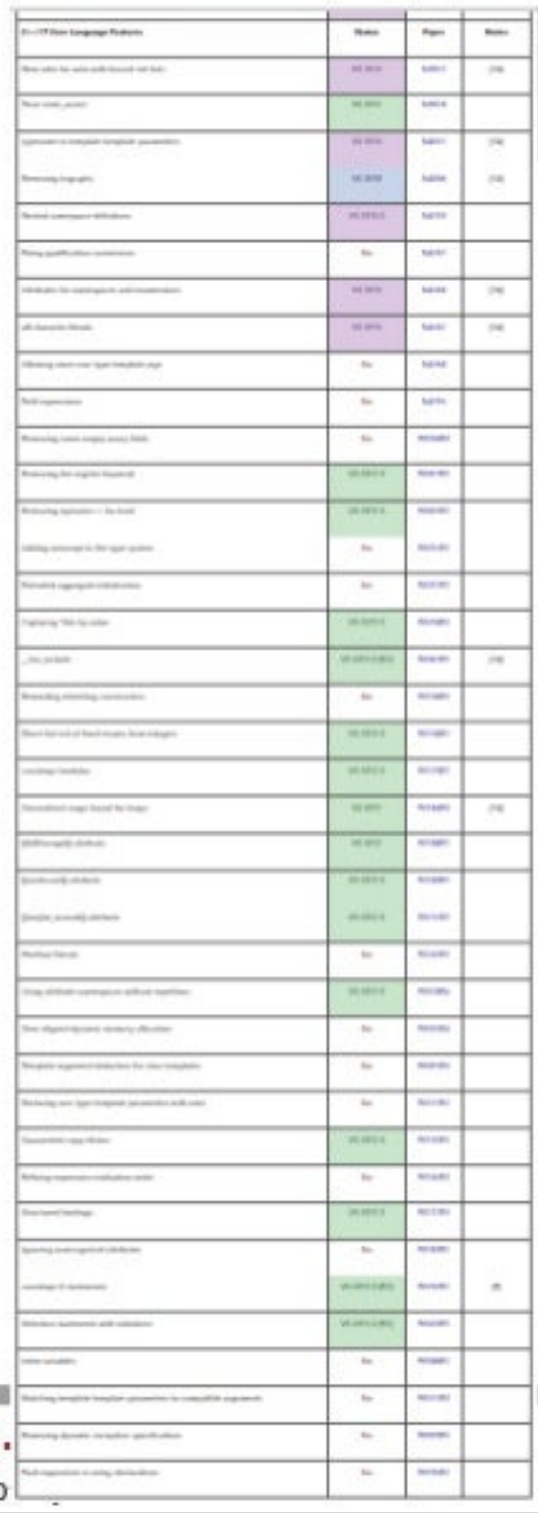
- As of 2017-05-14:

<= Core

Lib =>

VS2017

VS2015.x



C++17 Support by Visual Studio 2013 and 2015

- **Visual Studio 2013:**
  - Support for incomplete types in vector, list, and forward_list
- **Visual Studio 2015:**
  - global std::size(), std::empty(), std::data()
  - std::invoke()
  - std::uncaught_exceptions()
  - std::bool_constant, std::void_t
- **Visual Studio 2015 Update 2:**
  - global std::as_const()
  - Type traits suffix _v
  - std::scoped_lock
  - floor(), ceil(), round(), abs() for <chrono>
  - Type traits conjunction<>, disjunction<>, negation<>

C++17

Language Features

C++17: Structured Bindings

- **Initialize multiple identifiers/names from**
  - class/struct/union objects
  - std::pair<>, std::tuple<>, std::array<>
  - raw arrays
- **Types are deduced from members/elements**

see P0217R3

```
struct MyStruct {
  int i;
  std::string s;
};

MyStruct ms;
...
auto [u,v] = ms;                  // create implicit entity where u, v represent i, s
std::cout << u << ' '             // print members, assigned to [u,v]
          << v << '\n';
```

Equivalent to:

```
auto e = ms;
std::cout << e.i << ' '
          << e.s << '\n';
```

C++17: Using Structured Bindings

```
std::map<std::string, double> coll;

// C++11/C++14:
for (const auto& elem : coll) {      // elems are std::pair<const std::string, double>
  std::cout << elem.first << ": "
            << elem.second << '\n';
}

// since C++17:
for (const auto& [key,val] : coll) {  // access elems by read-only reference
  std::cout << key << ": "
            << val << '\n';
}


for (auto& [key,val] : coll) {        // access elems by reference
  if (key != "ignore") {
    val *= 2;
  }
}
```


C++17: Structured Bindings for Return Values

```
struct MyStruct {
  int i;
  std::string s;
};
MyStruct f1();
auto [u,v] = f1();          // u,v have type and value of members of returned object
auto [w] = f1();            // Error: number of elements does not fit

auto f2() -> int(&)[2];     // f2() returns reference to int array
auto [x, y] = f2();         // x and y are ints initialized by elems of returned array

std::array<int,4> f3();
auto [i,j,k,l] = f3();      // i,j,k,l name the 4 elements of the copied return value

std::tuple<char,float,std::string> f4();
auto [a,b,c] = f4();        // a,b,c have types and values of returned tuple
```

C++17: Structured Bindings and Qualifiers

- **Structured bindings declaration can have the usual qualifiers: `&`, `const`, `alignas`, ...**
  - Always apply to the implicitly created entity as a whole

```
int a[] = { 7, 11 };
…
auto& [ r, s ] = a;              // r and s are identifies to elements of a reference to a
r = 42;                          // modify first element in a
std::cout << a[0] << '\n';       // prints: 42

struct MyStruct {
  std::string name;
  int val;
};
MyStruct ms = {"Jim", 42};
auto&& [n,v] = ms;                                  // n and v are elements of reference to ms
std::cout << "name: " << ms.name << '\n';           // prints: Jim
std::string s = std::move(n);                       // move ms.name to s

MyStruct f1();
const auto& [nm,vl] = f1();                         // extends lifetime of returned temporary

alignas(16) auto[s,i] = f1();                       // entity is 16-byte aligned, not i
```


C++17: Addresses of Structured Bindings Elements

- **Structured binding declarations guarantee that the object whose members are denoted are kept "together"**

```
auto f() -> int (&)[2];
auto [x, y] = f();
assert(&x + 1 == &y);              // OK

struct S {                         // class with standard-layout
   int no;
   double price;
   std::string msg;
};
auto [n,p,s] = S{};
assert(&((S*)&n)->msg == &s);      // OK
```

Structures Bindings Might Not Decay Members

- `auto` does not decay structured bindings
  - because it applies to the object as a whole

```
struct S {
  int x[3];
  int y[3];
};

S s1{};   // initialize all member arrays with elements being 0
...
auto [a, b] = s1;                                  // a and b have same member types

std::is_same<decltype(a), int[3]>::value           // yields true
for (int elem : a) {                               // OK: a is still array
  std::cout << elem << '\n';
}

auto p = a;
std::is_same<decltype(p), int*>::value             // yields true
for (int elem : p) {                               // ERROR: p is no array
  ...
}
```

Equivalent to:

```
auto  e = s1;
auto& a = e.x;
auto& b = e.y;
```


C++17: Structured Bindings via Tuple-like APIs

```
class Customer {
 ...
 public:
  std::string getFirst() const { return first; }
  std::string getLast() const { return last; }
  long getValue() const { return val; }
};

#include <utility>   // for tuple-like API

// provide a tuple-like API for class Customer for structured bindings:
template<>
struct std::tuple_size<Customer> {
  static constexpr int value = 3;    // we have 3 attributes
};

template<>
struct std::tuple_element<2, Customer> {
  using type = long;                 // last attribute is a long
};
template<std::size_t Idx>
struct std::tuple_element<Idx, Customer> {
  using type = std::string;          // the other attributes are strings
};

// define specific getters:
template<int> auto get(const Customer& c);
template<> auto get<0>(const Customer& c) { return c.getFirst(); }
template<> auto get<1>(const Customer& c) { return c.getLast(); }
template<> auto get<2>(const Customer& c) { return c.getValue(); }
```

```
Customer c("Tim", "Lee", 42);

auto [f,l,v] = c;

// prints "Tim Lee: 42":
std::cout << f << ' '
          << l << ": "
          << v << '\n';
```

All full specializations of function templates have to use the same signature (including the exact same return type `auto`).

C++17: `if` with Initializers Need Names

```
{
  std::lock_guard<std::mutex> lg(mv);
  if (!v.empty()) {
    std::cout << v.front() << '\n';
  }
}
```

```
if (std::lock_guard<std::mutex> lg(mx); !v.empty()) {   // OK
  std::cout << v.front() << '\n';
}

if (std::lock_guard lg{mx}; !v.empty()) {   // OK, due to class template arg. deduction
  std::cout << v.front() << '\n';
}

if (std::lock_guard<std::mutex>(mx); !v.empty()) {   // ERROR: lock ends before ';'
  std::cout << v.front() << '\n';
}

if (std::lock_guard{mx}; !v.empty()) {               // ERROR: lock ends before ';'
  std::cout << v.front() << '\n';
}

if (std::lock_guard<std::mutex> _(mx); !v.empty()) {    // OK, but...
  std::cout << v.front() << '\n';
}
```


C++17: `if` and `switch` with Initializers

- **New additional syntax for `if` and `switch`:**

  `if` (*init*; *condition*)
  `switch` (*init*; *condition*)

```
{
  status s = check();
  if (s != status::ok) {
    return s;
  }
}
```

```
if (status s = check(); s != status::ok) {
  return s;
}
```

```
{
  std::lock_guard<std::mutex> lg(mv);
  if (!v.empty()) {
    std::cout << v.front() << '\n';
  }
}
```

```
if (std::lock_guard<std::mutex> lg(mx);
    !v.empty()) {
  std::cout << v.front() << '\n';
}
```

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

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

C++17: Structured Bindings and `if` with Initializer

- **Combining**
  - Initialize multiple objects by multiple return values
  - using the supplementary initialization syntax for `if`
- **For**

```
std::map<std::string, int> coll;
```

you can replace:

```
auto ret = coll.insert({"new",42});
if (!ret.second){
  // if insert failed, print why using iterator ret.first (for elem already there)
  const auto& elem = *(ret.first);
  std::cout << '"' << elem.first << "\" exists with key: "
            << elem.second << '\n';
}
```

Or in one expression:

```
ret.first->second
```

- **by:**

```
if (auto [pos,ok] = coll.insert({"new",42}); !ok) {
  // if insert failed, handle error using iterator pos (for elem already there)
  const auto& [key,val] = *pos;
  std::cout << '"' << key << "\" exists with key: " << val << '\n';
}
```


C++17: `if` with Initializers Need Names

- **Any temporary without a name in the initialization only exists there, not in the whole statement**
  - Same as with `for` loops

```
if (std::lock_guard<std::mutex> lg(mx); !v.empty()) {   // OK
  std::cout << v.front() << '\n';
}

if (std::lock_guard<std::mutex> _(mx); !v.empty()) {    // OK, but...
  std::cout << v.front() << '\n';
}

if (std::lock_guard<std::mutex>(mx); !v.empty()) {   // ERROR: lock ends before ';'
  std::cout << v.front() << '\n';
}

if (std::lock_guard lg(mx); !v.empty()) {   // OK, due to class template arg. deduction
  std::cout << v.front() << '\n';
}
```

C++17: `if` with Initializers Makes Hidden Checks Visible

- **Be careful when converting existing `if` with initialization:**

```
void f(const Expression* expr)
{
  if (const auto* unaryExpr = expr->AsUnaryExpression()) {
    if (unaryExpr->Kind() == TokenKind::Plus) {
      …
    }
  }
}
```

- **Naive modifications to use new feature:**

```
void f(const Expression* expr)
{
  if (const auto* unaryExpr = expr->AsUnaryExpression();
      unaryExpr->Kind() == TokenKind::Plus) {
    …
  }
}
```

missing implicit:
`if (unaryExpr)`

Thanks to Jonathan Caves for this example

C++17: Inline Variables

- **Static variables marked with `inline` count as definitions**
  - Guaranteed to exist only once in the program
  - No need for CPP files to define static/global objects

monitor.hpp:

```
class Monitor {
  public:
    Monitor() { … }
    void log(const std::string& msg) { … }
};

// Declare THE global monitor in the header file
// - might be included by multiple translation units
inline Monitor progMonitor;
```

```
#include "monitor.hpp"
…
progMonitor.log("main()");
```

```
#include "monitor.hpp"
…
progMonitor.log("init()");
```

…

C++17: Inline Variables

- **Static variables marked with `inline` count as definitions**
  - Guaranteed to exist only once in the program
    - even if included multiple times in multiple translation units
  - No need for CPP files to define static/global objects

c.hpp:

```
class C
{
  private:
    static inline bool lg = false;
  public:
    static void log (bool b) {
        lg = b;
    }

    C() {
      if (lg) std::cout << "C::C()\n";
    }
    …
};
```

```
#include "c.hpp"

int main()
{
  C::log(true);
  C c1;
  …
}
```

```
#include "c.hpp"

void foo()
{
  C c2;

  …
}
```


C++17 A Header-Only ::new Tracker

```
class TrackNew
{
 private:
  inline static int  numMalloc = 0;    // number of ::new or ::new[] calls
  inline static long sumSize = 0;      // bytes allocated so far
  inline static bool doTrace = false; // tracing enabled
 public:
  static void trace(bool b) {   // enable/disable tracing
    doTrace = b;
  }
  static void status() {        // print current state
    std::cerr << numMalloc << " mallocs for " << sumSize << " Bytes" << '\n';
  }
  static void* allocate(std::size_t size, const char* call) { // implementation of tracked allocation
    ++numMalloc;
    sumSize += size
    if (doTrace) …
    return std::malloc(size);
  }
};

inline void* operator new (std::size_t size) {
  return TrackNew::allocate(size, "::new");
}

inline void* operator new[] (std::size_t size) {
  return TrackNew::allocate(size, "::new[]");
}
```

```
// including the header is enough:
#include "tracknew.hpp"

int main()
{
  // optionally:
  TrackNew::trace(true);
  …
  TrackNew::status();
}
```

C++17: Inline Variables

- might be initialized before main() or before first use
- can be `thread_local`
- `constexpr` implies `inline`

monitor.hpp:

```
class Monitor {
  public:
    Monitor() { … }
    void log(const std::string& msg) { … }
};

// Declare THE global monitor in the header file
// - might be included by multiple translation units
inline thread_local Monitor threadLocalMonitor;
```


C++17: `static constexpr` Data Members are `inline`

- Static `constexpr` data members are `inline` since C++17
  - Declarations are definitions
  - Before C++17, the definitions often were not necessary
    - Optimized away
    - Definition not needed (e.g. passed by value)

```
struct A {
  static constexpr int n = 5; // C++11/C++14: declaration

};

std::cout << A::n;          // OK (ostream::operator<<(int) gets A::n by value)

int inc(const int& i);      // note: takes argument by reference
std::cout << inc(A::n);     // may fail without definition (OK with gcc –O3, VisualStudio 2015)
const int* p = &n;          // probably fails without definition (OK with VisualStudio2015)

// possible definition (in one translation unit):
constexpr int A::n;
```

C++17: `static constexpr` Data Members are `inline`

- **Static `constexpr` data members are `inline` since C++17**
  - Declarations are definitions
  - Before C++17, the definitions often were not necessary
    - Optimized away
    - Definition not needed (e.g. passed by value)

```
struct A {
  static constexpr int n = 5; // C++11/C++14: declaration
                              // since C++17: inline definition
};

std::cout << A::n;          // OK (ostream::operator<<(int) gets A::n by value)

int inc(const int& i);      // note: takes argument by reference
std::cout << inc(A::n);     // ...
const int* p = &n;          // pr...

// possible definition (in one translation unit):
constexpr int A::n;         // C++11/C++14: definition
                            // since C++17: redundant declaration (deprecated)
```

- Always OK since C++17

C++17: Aggregates with Base Classes

- **Aggregates now can have public base classes**
- **Initialization with nested {} possible**
- **New type trait: std::is_aggregate<>**

see P0017R1

Before C++17:

```
struct Dv : Data {
  bool used;
  Dv (string s, double d, bool b) {
    : Data{s,d}, used{b} {
  }
  ...
};
```

necessary to enable:

```
Dv x{"hi", 6.7, false};
```

```
struct Data {
  std::string name;
  double value;
};

struct Dv : Data {
  bool used;
  void print() const;
};
```

Since C++17, this is a class that is an aggregate

```
Dv u;                                  // OK: but value/done with undefined values
Dv z{};                                // OK: value/done with 0/false
bool b = std::is_aggregate<Aggr>::value;   // true
Dv x{{"item", 6.7}, false};            // OK since C++17
Dv y{"item", 6.7, false};              // OK since C++17 unless empty subobject/base
```

C++17: Aggregate Initialization

- **List initialization is zero initialization if no value is passed:**
  - Default constructor for class types
  - `0`, `false`, `nullptr` for fundamental data types (FDT)

```
struct Data {
  std::string name;
  double value;
};

struct PData : Data {
  bool flag;
};

PData a{{"test1", 1.1}, false};   // initialize all elements
PData b{"test2", 2.2, false};     // initialize all elements
PData c{};                        // same as: {{"",0.0},false}
PData d{{"msg"}};                 // same as: {{"msg",0.0},false}
PData e{{}, true};                // same as: {{"",0.0},true}
PData f;                          // numeric elements have unspecified value
```


C++17: Aggregates Incompatibilities

- **Aggregates are arrays or class types with**
  - no user-declared or explicit constructor
  - no constructor inherited by a using declaration
  - no private or protected non-static data members
  - no virtual functions
  - no virtual, private, or protected base classes
- **Aggregate initialization must not use private or protected base class members/constructors**

```
struct Derived;

struct Base {
 friend struct Derived;
 private:
  Base() {            // private default constructor
  }
};

struct Derived : Base {
};

Derived d1;           // still OK
Derived d2{};         // ERROR since C++17
```

C++14: No aggregate
- Calls implicit default constructor

C++17: Aggregate
- Call of private base constructor is not allowed

C++17: Aggregates Incompatibilities

- **Aggregates are arrays or class types with**
  - no user-declared or explicit constructor
  - no constructor inherited by a using declaration
  - no private or protected non-static data members
  - no virtual functions
  - no virtual, private, or protected base classes
- **Note: A deleted default constructor doesn't disable aggregate initialization (https://wg21.link/cwg1578)**

```
struct CData {
  int val;
  int elems[3];
};

class CppData : public CData {
  public:
    CppData() = delete;
    void print() const;
};

CppData cd1;                  // ERROR
CppData cd2{};                // ERROR with C++14, OK since C++17 (!!!)
CppData cd3{3, 0, 8, 15};     // ERROR with C++14, OK since C++17 (!!!)
```

C++14: No aggregate
- Constructor disables any initialization

C++17: Aggregate
- Aggregate initialization still OK

C++17: Mandatory RVO and Copy Elision

- **Copy elision for initialization from temporaries (prvalues) is required since C++17**
- **Callable copy/move constructor no longer required**

see P0135

```
class NoCpMv
{
 public:
  NoCpMv() = default;
  // no copy/move constructor:
  NoCpMv(const NoCpMv&) = delete;
  NoCpMv(NoCpMv&&) = delete;
  ...
};
```

```
void call(NoCpMv obj)
{
}

call(NoCpMv());       // OK since C++17
                      // (no copy/move)

NoCpMv ret()
{
  return NoCpMv();    // OK since C++17
}                     // (no copy/move)

NoCpMv x = ret();     // OK since C++17
call(ret());          // OK since C++17
call(x);              // still ERROR
```

C++17: Benefits from Mandatory Copy Elision

- **Required copy elision helps to**
  - provide factories for any type
  - dealing with types with deleted move

```
#include <utility>

// generic factory:
template <typename T, typename... Args>
T create(Args&&... args)
{
  ...
  return T{std::forward<Args>(args)...};
}
```

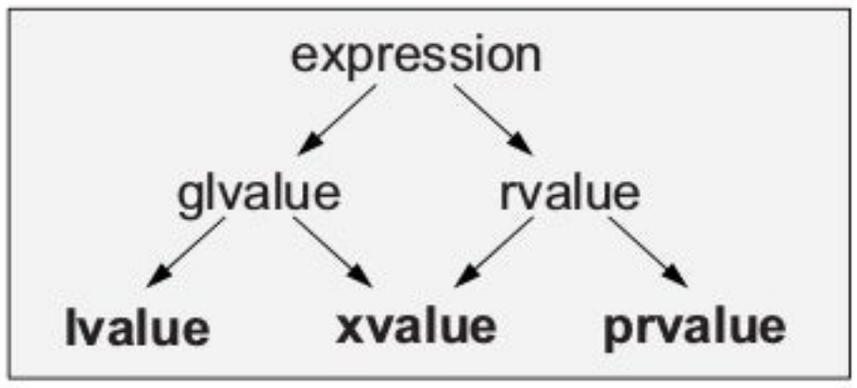
```
#include <atomic>

// OK since C++17:
// (Note: can't copy/move std::atomic<> objects)
auto ai = create<std::atomic<int>>(42);
```

```
class CopyOnly {
 public:
   CoypeOnly() {
   }
   CopyOnly(int) {
   }
   CopyOnly(const CopyOnly&) = default;
   CopyOnly(CopyOnly&&) = delete;
};

CopyOnly ret() {
  return CopyOnly{};   // OK since C++17
}

CopyOnly x = 42;       // OK since C++17
```

Remember:
Copy constructor is no fallback, if move constructor is explicitly deleted

Value Categories since C++11

C++11:



Every expression is one of:

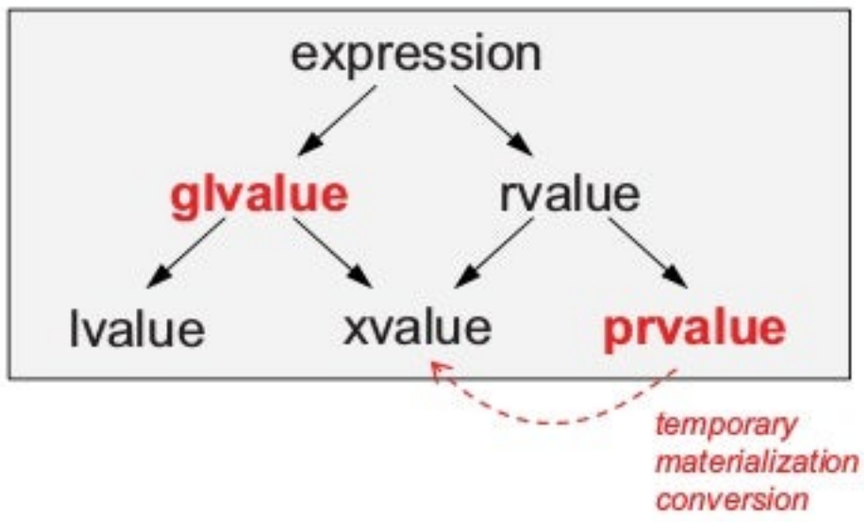
- **LValue: Localizable value**
  - Variable, data member, function, string literal, returned lvalue reference
  - Can be on the left side of an assignment only if it's modifiable
- **PRValue: Pure RValue** (former RValue)
  - All Literals except string literals (`42`, `true`, `nullptr`,...), `this`, lambda, returned non-reference, result of constructor call (`T(...)`)
- **XValue: eXpiring value**
  - Returned rvalue reference (e.g. by **std::move()**), cast to rvalue reference

General categories: GLValue: Generalized LValue, RValue: generalized RValue

Value Categories since C++17

C++17:

- **prvalues perform initialization**
  - no temporary object yet
- **glvalues produce locations**
- ***Materialization* as a temporary object:**
  - prvalue-to-xvalue conversion
    - when bound to a reference
    - with member access
    - with conversion to base




```
MyClass ret() {
  return MyClass();   // returns prvalue (no temporary object yet)
}
MyClass x = ret();    // uses prvalue for initialization

void callV(MyClass obj);         // accepts any value category
void callR(const MyClass& r);    // requires glvalue
void callM(MyClass&& r);         // requires xvalue  (may be materialized from prvalue)

callV(ret());                    // passes prvalue and uses it for initialization of obj
callR(ret());                    // passes prvalue (materialized as xvalue)  to r
callM(ret());                    // passes prvalue (materialized as xvalue)  to r
```


C++17: constexpr Lambdas

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

see P0170R1

```
auto squared = [](auto val) {        // implicitly constexpr since C++17
  return val*val;
};

std::array<int,squared(5)> a;        // OK since C++17  =>  std::array<int,25>


auto squared2 = [](auto val) constexpr {   // OK since C++17
  return val*val;
};


auto squared3 = [](auto val) constexpr {   // ERROR:
  static int calls=0;                      // can't be constexpr due to static
  ...
  return val*val;
};
```

Lambdas: Capturing `this`

- **In member functions, can also pass `this` as capture**
  - implicitly done with `[=]` and `[&]`
- **Since C++17, `*this` allows to pass `*this` by value**

```
class C
{
  private:
    std::string name;
  public:
    …
    void foo() {

      …  []  { std::cout << name << '\n'; }  …      // Error

      …  [&] { std::cout << name << '\n'; }  …      // OK
      …  [=] { std::cout << name << '\n'; }  …      // OK
      …  [this] { std::cout << name << '\n'; }  …   // OK

      …  [*this] {std::cout << name << '\n'; }  …   // OK since C++17:
                                                    // local copy of *this

    }
};
```


C++17: Attributes

- **New Attributes (formal annotations):**

```
[[ nodiscard ]]
[[ maybe_unused ]]
[[ fallthrough ]]
```

see P0189R1 P0212R1 P0188R1

- **Attributes for namespaces and enumerations:**

```
enum E {
  foobar = 0,
  foobat [[deprecated]] = foobar
};

namespace [[deprecated]] xyz {
  …
}
```

see N4266

- **Using declarations for attributes:**

```
[[ using xyz : … ]]
```

see P0028R4

C++17: New Attributes

- **C++17 defines new Attributes (formal annotations):**

  `[[nodiscard]]`
  - force warning if return values are not used
  - portable `[[gnu::warn_unused_result]]`
  - Possible example:

```
template <class F, class... Args>
[[nodiscard]] future<…> async(F&& f, Args&&... args);
```

But, only since C++20 used in the library

  `[[maybe_unused]]`
  - disabled warnings about names and entities that are intentionally not used
  - For example:

```
[[maybe_unused]] int y = foo();
```

  `[[fallthrough]]`
  - for intentional switch `cases` having statements but no `break;`

C++17: Nested Namespace Definitions

- **New syntax for nested namespace definitions:**
  - Instead of:

```
namespace A {
  namespace B {
    namespace C {
      …
    }
  }
}
```

see N4230

you can write since C++17:

```
namespace A::B::C {
  …
}
```

First proposed in 2003

- **No support for nested inline namespaces**

C++17: Heap Allocation with Alignment

- **Since C++17 operator new provides an interface to deal with aligned datatypes**
  - uses in `std`: `enum class align_val_t : size_t {};`
- **Several new overloads for operators**
  - new, new[]
  - delete, delete[]

```
new T
  // results into calling one of (whichever is provided):
  operator new(sizeof(T))
  operator new(sizeof(T), std::align_val_t(alignof(T)))

new(adr) std::string
  // results into calling one of (whichever is provided):
  operator new(sizeof(std::string), adr)
  operator new(sizeof(std::string),
               std::align_val_t(alignof(std::string)), adr)
```


C++17: Example for Heap Allocation with Alignment

```
#include <new>    // for align_val_t

struct alignas(512) Aligned {
  int i;
  alignas(128) double d;
  alignas(512) char c;

  // allocate memory for single object with default alignment:
  static void* operator new (std::size_t size) {
    std::cout << "Aligned::new called with size: " << size << "\n";
    return ::new char[size];
  }

  // allocate memory for single object with extended alignment:
  static void* operator new (std::size_t size, std::align_val_t align) {
    std::cout << "Aligned::new called with size: " << size
              << " and align: " << static_cast<int>(align) << "\n";
    return ::new char[size];
  }
};

Aligned a;                          // aligned for 512 address (since C++11)
Aligned* ap = new Aligned{};        // requests heap alignment for 512 address since C++17
```

max inner alignment forces outer alignment

used as fallback

size/align: 1024/512 (instead of e.g. 24/64)

C++17: __has_include

__has_include:

- Test for existence of a header
- Helps to write portable code

- For example:

```
#if __has_include(<optional>)
#  include <optional>
#  define HAS_OPTIONAL 1
#elif __has_include(<experimental/optional>)
#  include <experimental/optional>
#  define HAS_OPTIONAL 1
#  define OPTIONAL_IS_EXPERIMENTAL 1
#else
#  define HAS_OPTIONAL 0
#endif
```


C++17: UTF-8 Character Literals

- Encoding prefix u8 yields `char` with UTF-8 value

see N4267

  - since C++98:
    - `L'6'` for wchar_t literals (two or four bytes, no specific character set)
  - since C++11:
    - `u'6'` for char16_t literals (two-byte UTF-16)
    - `U'6'` for char32_t literals (four-byte UTF-32)
  - since C++17:
    - `u8'6'` for UTF-8 character literals
      - Already for string literals since C++11
    - provided:
      - the UTF-8 literal only has a single character
      - its value is representable with a single UTF-8 code unit (i.e. is a US-ASCII character)

```
u8'ö'   // ERROR: two bytes/code-units in UTF 8 (hex: C3 B6)
```

Only necessary, when the source/execution character set is not US-ASCII (e.g. compiling under an EBCDIC character set)

C++17: Hexadecimal Floating-Point Literals

see P0245R1

- `0x` **also for floating-point literals**
  - significand is given in hexadecimal
  - exponent is given in decimal and interpreted with respect to base 2
- **As already in C99**
- **As the `std::hexfloat` iostream format** (since C++11)

```
std::initializer_list<double> values {
                    0x1p4,          // 16
                    0xA,            // 10
                    0xAp2,          // 40
                    5e0,            // 5
                    0x1.4p+2,       // 5
                    1e5,            // 100000
                    0x1.86Ap+16,    // 100000
                    0xC.68p+2,      // 49.625
};
for (double d : values) {
  std::cout << "dec: " << std::setw(6) << std::defaultfloat << d
            << "  hex: " << std::hexfloat << d << '\n';
}
```

Output:

```
dec:     16  hex: 0x1p+4
dec:     10  hex: 0x1.4p+3
dec:     40  hex: 0x1.4p+5
dec:      5  hex: 0x1.4p+2
dec:      5  hex: 0x1.4p+2
dec: 100000  hex: 0x1.86ap+16
dec: 100000  hex: 0x1.86ap+16
dec: 49.625  hex: 0x1.8dp+5
```


C++17: Static_assert() without Messages

- **static_assert() no longer requires to pass a text message**
  - Printing a default text message if assertion is violated

see N3928

```
static_assert(sizeof(int)>=4, "integers are too small");   // OK since C++11
static_assert(sizeof(int)>=4);                             // OK since C++17

template <typename T>
class C
{
  static_assert(std::is_default_constructible<T>::value,
                "class C: element type T must be default-constructible");
  static_assert(std::is_default_constructible<T>::value); // OK since C++17
  ...
};
```

C++17

Template Features

C++17: More Convenience for Type Traits returning Values

- **Since C++17, shortcuts will be provided for type traits returning values**
  - Instead of:
    ```
    std::is_const<T>::value
    ```
    you can use:
    ```
    std::is_const_v<T>
    ```

- **This is done via variable templates**
  - available since C++14:

```
namespace std {

  template <typename T>
  constexpr bool is_const_v = is_const<T>::value;

}
```

C++17: Compile-Time `if`

- **C++17 provides a compile-time if**
  - "constexpr if"
  - The *then* or *else* part may become a *discarded statement*

```
template <typename T>
std::string asString(T x)
{
  if constexpr(std::is_arithmetic_v<T>) {
    return std::to_string(x);
  }
  else if constexpr(std::is_same_v<T, std::string>) {
    return x;
  }
  else {
    return std::string(x);
  }
}

std::cout << asString(42) << '\n';
std::cout << asString(std::string("hello")) << '\n';
std::cout << asString("hello") << '\n';
```

All example calls would be invalid when using run-time if

C++17: Compile-Time `if` with Initializers

```
template <typename Coll, typename Mutex>
void process(const Coll& coll, Mutex& m)
{
  …
  if constexpr (std::lock_guard lg{m};
                std::is_pointer_v<typename Coll::value_type>) {
    …  // implement processing for pointer elements
  }
  else {
    …  // implement processing for other elements
  }
  …
}

std::vector<int> vi;
std::mutex viMutex;
…
process(vi, viMutex);

std::vector<int*> vp;
std::mutex vpMutex;
…
process(vp, vpMutex);
```

Initialization might be evaluated at runtime

C++17: Compile-Time `if` Outside Templates

- `if constexpr` also can be used in non-template code
- All code in discarded statements must be valid

```
template <typename T>
void foo(T t);

int main()
{
 if constexpr(std::numeric_limits<char>::is_signed) {
   foo(42);                  // OK
   static_assert(std::numeric_limits<char>::is_signed,
                 "char is unsigned?"); // always fails if char is unsigned
 }
 else {
   undeclared(42);   // always error if undeclared() not declared
   static_assert(!std::numeric_limits<char>::is_signed,
                 "char is signed?");    // always fails if char is signed
 }
}
```

This program **never** successfully compiles

C++17: Compile-Time `if`

- The *discarded statement* must still be valid
  - Performs the first phase of template translation (definition phase)
  - Calls not depending on template parameters are instantiated

```
template <typename T>
void foo(T t)
{
  if constexpr(std::is_integral_v<T>) {
    if (t > 0) {
      foo(t-1);          // recursive call OK
    }
  }
  else {
    undeclared();        // always ERROR if not declared (even if discarded)
    undeclared(t);       // ERROR if not declared and not discarded (i.e. T is not integral)
    static_assert(sizeof(int)>4, "small int"); // may assert even if discarded
    static_assert(sizeof(T)>4, "small T");     // may assert only if not discarded
  }
}
```

- Compile-time error even if `foo<>()` never called
  - Except with VC++

Templates: Two-Phase Translation

1. Without instantiation at *definition time*:

- Syntax errors are discovered (such as missing semicolons)
- For code that does not depend on template parameters:
  - use of unknown names (type names, function names, ...) is discovered
  - static assertions are checked

2. At *instantiation time*:

- All code depending on template parameters is double-checked

```
template<typename T>
void foo(T t)
{
  undeclared();      // 1st phase error  if undeclared() unknown
  undeclared(t);     // 2nd phase error  if undeclared(T) unknown

  static_assert(sizeof(int)>4, "small int"); // 1st phase error  if sizeof(int) > 4
  static_assert(sizeof(T)>4, "small T");     // 2nd phase error if sizeof(T) > 4
  static_assert(false, "oops");  // always fails when template is compiled (even if not called)
}
```

- 3 compile-time errors even if `foo<>()` never called
  - Except with VC++

Templates: Two-Phase Translation and Visual C++

- **Visual C++** (by default) **does no first-phase lookup at all**
- **Visual Stuio 2017** (since 15.3) **allows first-phase lookup with `/permissive-`**
  - https://blogs.msdn.microsoft.com/vcblog/2017/09/11/two-phase-name-lookup-support-comes-to-msvc
  - Undeclared functions and `static_assert()` still always ignored

```
void func(void*) {
  std::cout << "calls func(void*) (first-phase lookup)\n";
}

template<typename T>
void callfunc(T) {
  func(0);    // first-phase lookup should find only func(void*) here
}

void func(int) {
  std::cout << "calls func(int) (second-phase lookup)\n";
}

int main()
{
  callfunc(42);   // calls func(void*) with correct two-phase lookup
}
```

- gcc, clang, and VS2017 with `/permissive-`:
  - calls `func(void*)` (1st phase lookup)
- VS2013, VS2015, VS2017 (without `/permissive-`):
  - calls `func(int)` (2nd phase lookup)

C++17: Compile-Time if and Variadic Templates

```
template <typename T, typename... Types>
void print (T firstArg, Types... args)
{
  std::cout << firstArg << '\n';
  if (sizeof...(args) > 0) {
    print(args...);   // Error: needs print() if sizeof...(args)==0
  }
}
```

```
template <typename T, typename... Types>
void print (T firstArg, Types... args)
{
  std::cout << firstArg << '\n';
  if constexpr(sizeof...(args) > 0) {   // since C++17
    print(args...);   // OK: not instantiated if sizeof...(args)==0
  }
}
```


C++17: Fold Expressions

- **Apply binary operators to all elements of a parameter pack**

```
template <typename... T>
auto foldSum1 (T... s){
  return (... + s);        // s1 + s2 + s3 ...
}

template <typename... T>
auto foldSum2 (T... s){
  return (0 + ... + s);    // 0 + s1 + s2 + s3 ...
}
```

```
auto i = foldSum1(17, 4);                        // 21
auto j = foldSum1(i, 1000, i);                   // 1042
foldSum1("hi", "hi", "hi");                      // ERROR
foldSum1("hi", "hi", std::string("hi"));         // ERROR
foldSum1(std::string("hi"), "hi", "hi");         // OK: hihihi
foldSum1();                                      // ERROR

auto i = foldSum2(17, 4);                        // 21
auto j = foldSum2(i, 1000, i);                   // 1042
auto k = foldSum2();                             // 0
foldSum2(std::string("hi"), "hi", "hi");         // ERROR
```

C++17: Fold Expressions

- **Apply binary operators to all elements of a parameter pack**
- **Supported syntax:**

  (... *OP* *pack*)
  (*init* *OP* ... *OP* *pack*)
  (*pack* *OP* ...)
  (*pack* *OP* ... *OP* *init*)

((*pack1* *OP* *pack2*) *OP* *pack3*)
(((*init* *OP* *pack1*) *OP* *pack2*) *OP* *pack3*)
} same from right to left

- **For operators `&&`, `||`, and `,` the pack might even be empty**
  - In that case, yields `true`, `false`, `void()`

```
template <typename... T>
auto foldSum (T... s){
  return (... + s);          // s1 + s2 + s3 ...
}

template <typename... T>
auto foldSum (T... s){
  return (0 + ... + s);      // even works if sizeof...(s)==0
}
```


C++17: Using Fold Expressions

```
template<typename T>
void print (T arg)
{
   std::cout << arg << ' ';
}

template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
   print(firstArg);
   print(args...);
}
```

```
std::string str = "world";
print(7.5, "hi", str);      // 7.5 hi world

  => print(7.5);
     print("hi", str);

     => print(7.5);
        print("hi")
        print(str);
```

Code effectively compiled:

```
std::string str = "world";
std::cout << "hi" << ' ';
std::cout << 7.5 << ' ';
std::cout << str << ' ';
```

```
template <typename... Args>
void printAll (Args... args) {
  (std::cout << ... << args) << ' ';
}
```

```
std::string str = "world";
printAll(7.5, "hi", str);       // 7.5hiworld
```

C++17: Using Fold Expressions

```
template <typename... Args>
void printAll (Args... args) {
  (std::cout << ... << args) << ' ';
}
```

```
std::string str = "world";
printAll(7.5, "hi", str);        // 7.5hiworld
```

```
template<typename T>
const T& spaceBefore(T arg) {
  std::cout << ' ';
  return arg;
}

template <typename First, typename... Args>
void printSpaced(First arg0, Args... args) {
  std::cout << arg0;
  (std::cout << ... << spaceBefore(args)) << ' ';
}
```

```
std::string str = "world";
printSpaced(7.5, "hi", str);    // 7.5 hi world
```


C++17: Fold Expressions for Function Calls

```
#include <iostream>

struct A {
  void print() {
    std::cout << "A\n";
  }
};

struct B {
  void print() {
    std::cout << "B\n";
  }
};

struct C {
  void print() {
    std::cout << "C\n";
  }
};
```

```
template<typename... Bases>
struct MultiBase : private Bases...
{
  void print() {
    (... , Bases::print());
  }
};
```

expands to:

```
A::print(), B::print(), C::print();
```

```
int main()
{
    MultiBase<A,B,C> mb;
    mb.print();
}
```

C++17: Fold Expressions for Member Pointers

```
// define binary tree structure:
struct Node {
   int value;
   Node* left;
   Node* right;
   …
};
auto left = &Node::left;
auto right = &Node::right;

// traverse tree, using fold expression:
template <typename T, typename... TP>
X* traverse (T np, TP... paths){
  return (np ->* ... ->* paths);     // np ->* paths1 ->* paths2 ...
}

// init and use binary tree structure:
X* root = new Node{0};
root->left = new Node{1};
root->left->right = new Node{2};
…
auto p = traverse(root, left, right);
```


Fold Expressions for Types

- Fold expressions can also apply to types

```
template<typename T1, typename... TN>
struct IsHomogeneous {
  static constexpr bool value
    = (std::is_same<T1,TN>::value && ...);
};

using Size = int;
…
IsHomogeneous<int, Size, decltype(42)>::value
```

expands to:
```
std::is_same<int,Size>::value &&
std::is_same<int,decltype(42)>::value
```

```
template<typename T1, typename... TN>
constexpr bool isHomogeneous(T1, TN...)
{
  return (std::is_same<T1,TN>::value && ...);
}
isHomogeneous(43, -1, "hello", nullptr)
```

expands to:
```
std::is_same<int,int>::value &&
std::is_same<int,const char*>::value &&
std::is_same<int,std::nullptr_t>::value
```

C++17: Class Template Argument Deduction

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

```
template<typename T>
class complex;
...
std::cout << std::complex<int>{5,3};  // OK, all C++ versions

std::cout << std::complex{5,3};     // OK since C++17, deduces std::complex<int>
std::cout << std::complex(5,3);     // OK since C++17, deduces std::complex<int>

std::cout << std::complex(5,3.3); // ERROR: args do not have the same type T


std::mutex mx;
std::lock_guard lg(mx);   // OK since C++17
                          // deduces: std::lock_guard<std::mutex>
```


C++17: No Partial Argument Deduction for Class Templates

- **Partial class template argument deduction is not possible**
  - Specify all **or** deduce all template parameters (without defaults)

```
template <typename T1, typename T2, typename T3 = T2>
class C
{
  public:
    C (T1 x = T1{}, T2 y = T2{}, T3 z = T3{}); // constructor for 0, 1, 2, or 3 arguments
    ...
};

C<string,string,int> c0;                  // OK, T1,T2 are string, T3 is int
C<string,string,int> c1("hi","guy",42);   // OK, T1,T2 are string, T3 is int
C<int,string> c2(52,"my");                // OK, T1 is int,T2 and T3 are strings
C<string,string> c3("hi","my","guy");     // OK, T1,T2,T3 are strings
C<string,string> c4("hi","my",42);        // Error: 42 is not a string

C c5;                          // Error: T1 and T2 undefined
C c6("hi");                    // Error: T2 undefined
C c7(22,44.3);                 // OK since C++17, T1 is int, T2 and T3 are double
C c8(22,44.3,"hi");            // OK since C++17, T1 is int, T2 is double, T3 is const char*
C c9("hi","guy");              // OK since C++17, T1, T2, and T3 are const char*

C<string> c10("hi","my");      // Error: only T1 explicitly defined
C<> c11(22,44.3);              // Error: neither T1 nor T2 explicitly defined
C<> c12(22,44.3,42);           // Error: neither T1 nor T2 explicitly defined
```

C++17: Applying Argument Deduction for Class Templates

```
std::cout << std::complex(5,3);      // OK since C++17, deduces a std::complex<int>
std::cout << std::complex(5,3.3);    // Error: args do not have the same type


std::mutex mx;
std::lock_guard lg(mx);   // OK since C++17, deduces: std::lock_guard<std::mutex>



std::tuple<int> t(42, 43);   // still error (good!)
```

Motivating example against allowing partial deduction

```
// creating set with specific sorting criterion:
std::set<Cust> coll([](const Cust& x, const Cust& y) {   // still error (too bad)
                       return x.name() > y.name();
                    });

auto sortcrit = [](const Cust& x, const Cust& y) {
                     return x.name() > y.name();
                  };
std::set<Cust, decltype(sortcrit)> coll(sortcrit);   // OK
```


C++17 Deducing Type of Lambdas in Class Templates

```
#include <utility>      // for std::forward()

template<typename CB>
class CountCalls
{
  private:
    CB callback;        // callback to call
    long calls = 0;     // counter for calls
  public:
    CountCalls(CB cb) : callback(cb) {
    }
    template<typename... Args>
    auto operator() (Args&&... args) {
      ++calls;
      return callback(std::forward<Args>(args)...);
    }
    long count() const {
      return calls;
    }
};

CountCalls sc([](auto x, auto y) {
                  return x > y;
              });
std::sort(v.begin(), v.end(),
          std::ref(sc));
std::cout << "sorted with " << sc.count() << " calls\n";
```

enables to deduce the type CB of a passed lambda/functor

deduces CountCalls<*TypeOfLambda*>

C++17: Class Template Argument Deduction "Copies by Default"

- **Class Argument Template Deduction "Copies by Default"**
  - After

    `C<T> x;`

    `C{x}` deduces to: `C<T>{x}` instead of: `C<C<T>>{x}`

```
std::vector v{1, 2, 3};     // vector<int>
std::vector a{v};           // vector<int>
std::vector b(v);           // vector<int>
std::vector c = {v};        // vector<int>
auto d = std::vector{v};    // vector<int>

std::vector v2{v, v};       // vector<vector<int>>
```

instead of: vector<vector<int>>

C++17: Deduction Guides

- **Deduction guides can add/fix deductions**
  - Must be in the same scope (e.g. namespace std)

```
template<typename T> class complex;     // as in <complex>
...
std::cout << std::complex<int>{5,3};    // OK, all versions
std::cout << std::complex{5,3};         // OK since C++17, deduces a std::complex<int>
std::cout << std::complex(5,3);         // OK since C++17, deduces a std::complex<int>
std::cout << std::complex(5,3.3);       // Error: args do not have the same type T
```

A **user-defined** deduction guide for any standard library class template result into undefined behavior.

```
// If we'd have:
namespace std {
  template<typename T1, typename T2>
   complex(T1,T2) -> complex<common_type_t<T1,T2>>;
}

// then:
std::cout << std::complex(5,3.3);   // would be OK and deduce to std::complex<double>
```

Using Deduction Guides in Practice

Constructors with references:

- **Use deduction guides to decay template parameter types**
  - String literals deduce to pointer types
- **Parameters are still passed by reference**

```
template<typename T>
class MyType {
 private:
   T value;
 public:
   MyType(const T& v) : value{v} {
   }
   MyType(T&& v) : value{std::move(v)} {
   }
   ...
};

// deduction guide for the constructors:
template<typename T>
MyType(T) -> MyType<T>;
```

```
MyType x{"hi"}
// equivalent to:
MyType<const char*> x{"hi"}
```

- Error without deduction guide
  => `const char value[3];`
- Error with `std::decay_t<T> value;`
  => `char* value;`

Constructors should move initialize members from by-value parameters:

```
MyType(T v) : value{v} {
}
```


C++17: Deduction Guides

- **Deduction guides**
  - may not be templates
  - may not call a constructor (e.g., for aggregates)

```
template<typename T>
struct S
{
  T val;
};
S(const char*) -> S<std::string>;   // map S<> for string literals to S<std::string>


S s1{"hello"};                 // OK, same as: S<std::string> s1{"hello"};
S s2 = {"hello"};              // OK, same as: S<std::string> s2 = {"hello"};
S s3 = S{"hello"};             // OK
S s4 = "hello";                // ERROR (no aggregate initialization)

void foo(S<std::string>);
foo(S{"hello"});               // OK, same as: foo(S<std::string>{"hello"});

foo({"hello"});                // OK, does not need a deduction guide
foo("hello");                  // ERROR (no aggregate initialization)
```

C++17: Explicit Deduction Guides

- **Deduction Guides can be explicit**
  - Affect only explicit conversions
    - No implicit conversions

```
template<typename T>
struct S
{
  T val;
};

explicit S(const char*) -> S<std::string>; // map S<> for string literals to S<std::string>
                                           // (not for implicit construction)

S s1{"hello"};              // OK, same as: S<std::string> s1{"hello"};
S s2 = {"hello"};           // ERROR (implicit conversion)
S s3 = S{"hello"};          // OK
S s4 = {S{"hello"}};        // OK

void foo(S<std::string>);
foo(S{"hello"});            // OK, same as: foo(S<std::string>{"hello"});

foo({"hello"});             // OK, does not need a deduction guide
foo("hello");               // ERROR (no aggregate initialization)
```


C++17: Aggregates/Classes and Argument Deduction

```
template <typename T>
struct A
{
  T val;
};

A i1{42};          // ERROR
A<int> i2{42};     // OK
A s1("hi");        // ERROR
A s2{"hi"};        // ERROR
A s3 = "hi";       // ERROR
A s4 = {"hi"};     // ERROR
A<std::string> s5 = {"hi"};

A(const char*)
  -> A<std::string>;

A s1("hi");        // ERROR (1)
A s2{"hi"};        // OK, string
A s3 = "hi";       // ERROR (1)
A s4 = {"hi"};     // OK, string
A<std::string> s5 = {"hi"};
```

```
template <typename T>
struct AV
{
  T val;
  AV(T s) : val(s) {
  }
};

AV s1("hi");       // OK, const char*
AV s2{"hi"};       // OK, const char*
AV s3 = "hi";      // OK, const char*
AV s4 = {"hi"};    // OK, const char*
AV<std::string> s5 = {"hi"};

AV(const char*)
  -> AV<std::string>;

AV s1("hi");       // OK, string
AV s2{"hi"};       // OK, string
AV s3 = "hi";      // ERROR (2)
AV s4 = {"hi"};    // OK, string
AV<std::string> s5 = {"hi"};
```

```
template <typename T>
struct AR
{
  T val;
  AR(const T& s) : val(s) {
  }
};

AR s1("hi");       // ERROR (3)
AR s2{"hi"};       // ERROR (3)
AR s3 = "hi";      // ERROR (3)
AR s4 = {"hi"};    // ERROR (3)
AR<std::string> s5 = {"hi"};

AR(const char*)
  -> AR<std::string>;

AR s1("hi");       // OK, string
AR s2{"hi"};       // OK, string
AR s3 = "hi";      // ERROR (2)
AR s4 = {"hi"};    // OK, string
AR<std::string> s5 = {"hi"};
```

Error (1): No valid aggregate initialization
Error (2): Invalid nested implicit user-defined conversion
Error (3): Can't use array as initializer

```
struct S {
  S(std::string s);
};
S x = "hi";   // Error
```

C++17: Deduction Guides in the Library

- **To decay and support member templates**
  - `pair<>`, `tuple<>`, `optional<>`

see P0433R2

- **Disabled for pointers (due to array type clash):**
  - `unique_ptr<>`, `shared_ptr<>`
    - conversions from `weak_ptr<>`/`unique_ptr<>` to `shared_ptr<>`

- **Initialize from range**
  - containers (except `array<>`), strings, regex

- **Other:**
  - `std::array a {42,45,77} => std::array<int,3>`
  - Container adapters deduce passed container types
  - Copying for all lock guard classes

C++17: Deduction Guides in the Library

- **Deduction guides can add/fix deductions**
- **Some are pre-defined**
- **For example:**

```
// let std::vector<> deduce element type from initializing iterators:
namespace std {
  template<typename Iterator>
  vector(Iterator, Iterator)
    -> vector<typename iterator_traits<Iterator>::value_type>;
}

std::set<float> s;
std::vector(s.begin(), s.end());    // OK, deduces: std::vector<float>
```

C++17: Deduction Guides in the Library

- **Deduction guides can add/fix deductions**
- **Some are pre-defined**
- **For example:**

```
// let std::array<> deduce their number of elements (must have same type):
namespace std {
  template<typename T, typename... U>
  array(T, U...)
    -> array<enable_if_t<(is_same_v<T,U> && ...), T>,
             (1 + sizeof...(U))>;
}
```

Fold Expression

```
std::array a{42,45,77};      // OK, deduces: std::array<int,3>
std::array a{42,45,77.7};    // Error: types differ
```


C++17: Deduction Guides for Smart Pointers

- **`make_shared()` and `make_unique()` are still useful**
  - class template argument deduction is not enabled there

```
namespace std {
  template<typename T> class shared_ptr {
    public:
      constexpr shared_ptr() noexcept;
      template<typename Y> explicit shared_ptr(Y* p);
      …
  };
}
```

```
// IF WE WOULD HAVE: deduction guide for shared pointer initialization:
namespace std{
  template<typename Y> shared_ptr(Y*) -> shared_ptr<Y>;
}

std::shared_ptr<int> sp{new int(7)}; // OK
std::shared_ptr sp{new int(7)};      // OK: deduces: shared_ptr<int>
std::shared_ptr sp{new int[10]};     // OOPS: deduces: shared_ptr<int>
```

C++17: Deduction Guides Overloading

- **Deduction guide define template parameters**
- **You can overload deduction guides**

```
template<typename T>
struct C {
  C(T) {
  }
};

template<typename T> C(const T&) -> C<double>;
template<typename T> C(T&)       -> C<long>;
template<typename T> C(T&&)      -> C<int>;

C c1(42);              // deduces C<int>
int x = 42;
C c2(x);               // deduces C<long>
C c3(std::move(x));    // deduces C<int>
const int c = 42;
C c4(c);               // deduces C<double>
C c5("oops");          // ERROR: can't convert const char[5] to double
```


C++17: Using Declarations are Lists now

- **Using declarations can be lists now**
  - Side effect of pack expansions in using declarations

see P0195R2

```
class Base {
  public:
    void a();
    void b();
    void c();
};

class Derived : private Base {
  public:
    using Base::a, Base::b, Base::c;
};

Derived d;
d.a();
d.b();
d.c();
```

C++17: Pack Expansion in `using` Declarations

see P0195R2

```
#include <string>
#include <unordered_set>

class Customer {
    std::string name;
  public:
    Customer(const std::string& n) : name(n) { }
    std::string getName() const { return name; }
};

struct CustomerEq {
    bool operator() (const Customer& c1, const Customer& c2) const {
      return c1.getName() == c2.getName();
    }
};

struct CustomerHash {
    std::size_t operator() (const Customer& c) const {
      return std::hash<std::string>()(c.getName());
    }
};

template <typename... Bases>
struct Overloader : Bases... {
    using Bases::operator()...;   // OK since C++17
};

using CustomerOP = Overloader<CustomerHash,CustomerEq>;   // combine hasher and == in one type

std::unordered_set<Customer,CustomerHash,CustomerEq> coll1;
std::unordered_set<Customer,CustomerOP,CustomerOP> coll2;
```

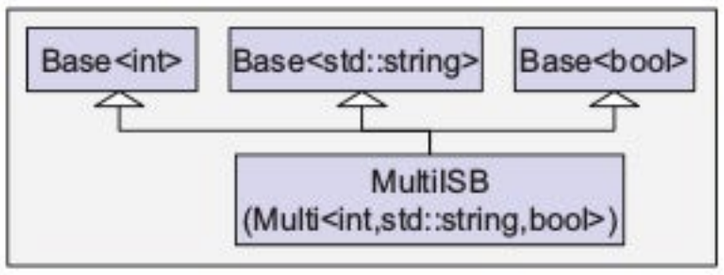

Clarified Inheriting Constructors

- **Several clarifications on inheriting constructors**
  - `using Base::Base`
    - adopts all constructors from the base classes
    - Constructors for T call
      - `Base::Base(T)` for matching type
      - `Base::Base()` for other types

see N4429

```
// wrap any class by adopting all of its constructors,
// while writing a message to the standard log whenever an object is destroyed.
template<typename T>
struct Log : public T {
  using T::T;     // inherit all constructors from class T
  ~Log() {
    std::clog << "Destroying wrapper" << '\n';
  }
};
```

C++17: Variadic `using` Declarations and Inherited Constructors




```
template<typename T>
class Base {
    T value{};
  public:
    Base() { }
    Base(const T& v) : value{v} { }
    Base& operator= (const T& v) { value = v; return *this; }
};

template<typename... Types>
class Multi : private Base<Types>...
{
  public:
    // derive all constructors:
    using Base<Types>::Base...;

    // derive all assignment operators:
    using Base<Types>::operator=...;
};

using MultiISP = Multi<int,std::string,void*>;
std::cout << std::is_base_of_v<Base<int>, MultiISP>;   // true

MultiISP m1 = 42;
MultiISP m2{"hello"};   // OK (error with MultiISB v2="hello" due to two user-defined conversions)
MultiISP m3 = nullptr;

m3 = 42;                // OK (would also work without using operator=...)
m2 = "world";           // OK due to using operator=...
```

see N4429

Calls
- `Base::Base(T)` for matching type
- `Base::Base()` for other types

Strings as Template Arguments

- **Since C++17, non-type template parameters can be instantiated with strings with no linkage**

see N4268

```
template <const char* str>
class Message
{
  ...
};

extern const char hello[] = "Hello World!";       // external linkage
const char hello11[] = "Hello World!";            // internal linkage

void foo()
{
  Message<hello>   msg;          // OK, all versions
  Message<hello11> msg11;        // OK since C++11

  static const char hello17[] = "Hello World!";   // no linkage
  Message<hello17> msg17;        // OK since C++17
}
```

C++17: Non-Type Template Parameters with `auto`

- **Since C++17 you can declare non-type template parameters with *placeholder types*: `auto` and `decltype(auto)`**

```
template <auto N> class S {
  ...
};
S<42>   s1;   // OK: type of N in S is int
S<'a'>  s2;   // OK: type of N in S is char
S<2.5>  s3;   // Error: template parameter type still cannot be double

// partial specialization:
template <int N> class S<N> {
  ...
};

// template where P must be a pointer to const something:
template <const auto* P> struct S;

// List of heterogeneous constant template arguments
template <auto... VS> struct value_list { };

// List of homogeneous constant template arguments:
template <auto V1, decltype(V1)... VS> struct typed_value_list { };
```


C++17: `auto` and Strings for Non-Type Template Parameters

```
template <auto Msg>
class Message {
  public:
    void print() {
      std::cout << Msg << '\n';
    }
};

int main()
{
  Message<'x'> m1;
  m1.print();          // outputs: x

  Message<42> m2;
  m2.print();          // outputs: 42

  static const char s[] = "hello";
  Message<s> m3;       // OK since C++17
  m3.print();          // outputs: hello
}
```

C++17: Using Fold Expressions with `auto` Template Parameters

```
template<auto sep = ' ',
         typename First, typename... Args>
void print(const First& arg0, const Args&... args)
{
  std::cout << arg0;
  auto coutSepAndArg = [](const auto& arg) {
                         std::cout << sep << arg;
                       };
  (... , coutSepAndArg(args));
  std::cout << '\n';
}
```

```
std::string str = "world";
print(7.5, "hi", str);          // 7.5 hi world

print<'-'>(7.5, "hi", str);     // 7.5-hi-world

static const char sep[] = ", ";
print<sep>(7.5, "hi", str);     // 7.5, hi, world
```


C++17: Variable Templates with `auto`

```
#include <array>

template<typename T, auto N> std::array<T,N> arr;   // OK since C++17
template<auto N> constexpr decltype(N) val = N;     // OK since C++17
```

two different global objects `arr<...>` used in both translation units

```
#include "variable.hpp"

void print();

int main()
{
  arr<double,100u>[0] = 17;
  arr<int,10>[0] = 42;

  print();

  std::cout << val<'c'> << '\n';
}
```

```
#include "variable.hpp"
#include <iostream>

void print()
{
  std::cout << arr<double,100u> << '\n';
  std::cout << arr<int,10> << '\n';

  for (unsigned i=0; i<arr<int,10>.size(); ++i) {
    std::cout << arr<int,10>[i] << '\n';
  }
}
```

C++17: Non-Type Template Parameters with `decltype(auto)`

```
#include <iostream>
#include <type_traits>

template<decltype(auto) N>
struct S {
  S() {
    if (std::is_same_v<decltype(N),int&>) std::cout << "ref ";
    std::cout << "N has value " << N << '\n';
  }
  void print() {
    std::cout << "value: " << N << '\n';
  }
};

constexpr int x = 42;
int y = 0;          // lvalue

int main()
{
    S<x> s0;        // N is int   => prints: N has value 42
    S<(y)> s1;      // N is int&  => prints: ref N has value 0
    y = 77;
    S<(y)> s2;      // N is int&  => prints: ref N has value 77
    y = 88;
    s1.print();     // prints: value: 88
    s2.print();     // prints: value: 88
}
```

Remember: `decltype(auto)` **uses the rules of** `decltype`**:**

- **type of entity e**
- **value category of expression e:**
  - for prvalues: *type*
  - for lvalues: *type*&
  - for xvalues: *type*&&

C++17

Core Fixes and Improvements

C++17: Defined Expression Evaluation Order

- **Most operators have no defined evaluation order since C:**

```
std::string s = "I heard it even works if you don't believe";
s.replace(0,8,"").replace(s.find("even"),4,"sometimes")
                 .replace(s.find("you don't"),9,"I");

it sometimes works if I believe   // guaranteed since C++17

it sometimes workIdon't believe   // possible before C++17
it even worsometiIdon't believe   // possible before C++17
it even worsometimesf youIlieve   // possible before C++17

// also undefined behavior if f(), g(), and/or h() depend on each other:
std::cout << f() << g() << h();   // calls: cout.operator<<(f())
                                  .operator<<(g()).operator<<(h());
```

- **C++17 defines evaluation order for:**

`.` `->` `.*` `->*` `= += -=...` `[]` `<<` `>>` *expr*(*a*,*b*,*c*)

– *expr* before *a* and *b* and *c*, but *a*,*b*,*c* still in any order

C++17: Defined Expression Evaluation Order

- **C++17 defines evaluation order of:**

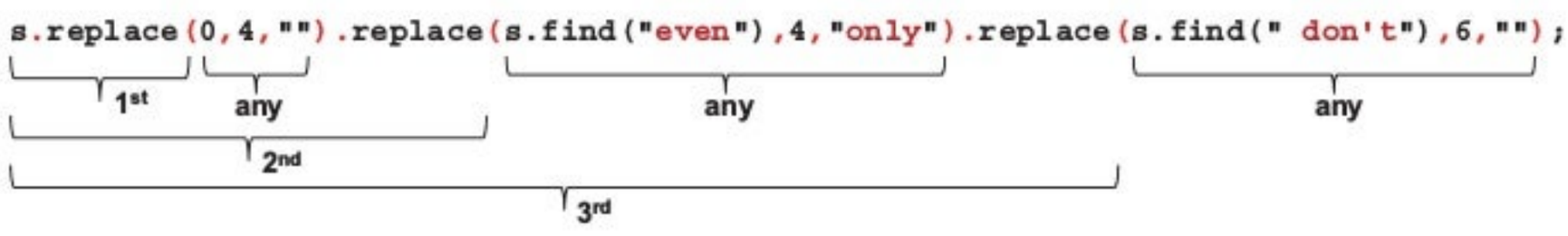
`.` `->` `.*` `->*` `= += -=...` `[]` `<<` `>>` *f*(*a*,*b*,*c*)

– For *f*(*a*,*b*,*c*):

f before *a*/*b*/*c*, but *a*,*b*,*c* still in any order

```
std::string s = "but I have heard it works even if you don’t believe";
s.replace(0,4,"").replace(s.find("even"),4,"only")
                 .replace(s.find(" don't"),6,"");
```

is now evaluated as follows:



C++17: Defined Expression Evaluation Order Consequences

```
void printElem(const std::vector<int>& v, int idx,
               const std::string& pre = "")
{
  std::cout << pre << v.at(idx) << '\n';   // might throw std::out_of_range
}
```

- Since C++17 expressions for operator << are guaranteed to evaluate from left to right

```
try {
  std::vector<int> coll{7, 14, 21, 28};
  …
  for (int i=0; i<=coll.size(); ++i) {
    printElem(coll, i, "elem: ");
  }
  …
}
catch (const std::exception& e) {
  std::cerr << "EXCEPTION: "
            << e.what() << '\n';
}
catch (...) {
  std::cerr << "EXCEPTION of unknown type\n";
}
```

Might output before C++17:

```
elem: 7
elem: 14
elem: 21
elem: 28
EXCEPTION: …
```

Will output since C++17:

```
elem: 7
elem: 14
elem: 21
elem: 28
elem: EXCEPTION: …
```


C++17: `noexcept` Part of the Function Type

- **C++17 type system differentiates between**
  - noexcept functions
  - functions that don't guarantee not to throw
- **A function that might throw can't be used as noexcept function**
  - The opposite is fine

see P0012R1

```
void f1();
void f2() noexcept;                          // different type than f1

void f3() noexcept(sizeof(int)<4);           // same type as either f1 or f2
void f4() noexcept(sizeof(int)>=4);          // different type than f3
```

C++17: `noexcept` Part of the Function Type

- **C++17 type system differentiates between**
  - noexcept functions
  - functions that don't guarantee not to throw
- **A function that might throw can't be used as noexcept function (the opposite is fine)**
- **Type of functions with conditional noexcept specifications depends on what the condition yields**
- **Dynamic exception specifications were removed**

see P0012R1

```
void f1();
void f2() noexcept;                          // different type than f1

void f3() noexcept(sizeof(int)<4);           // same type as either f1 or f2
void f4() noexcept(sizeof(int)>=4);          // different type than f3
                                             // same type as f2 if f3 has same type as f1

void fnothrow() throw();                     // OK, but may not unwind stack if violated
                                             // same type as f2
void fthrow() throw(std::bad_alloc);         // Error, invalid since C++17
```


C++17: `noexcept` Part of the Function Type

```
void f1();
void f2() noexcept;          // different type than f1

void (*fp)() noexcept;       // requires noexcept function
fp = f2;                     // OK
fp = f1;                     // Error since C++17

void (*fp2)();               // doesn't require noexcept function
fp2 = f2;                    // OK
fp2 = f1;                    // OK (can also use noexcept function)
```

```
template<typename T>
void call(T op1, T op2);

call(f1, f2);                // Error since C++17 (f1 and f2 now have different types)


template<typename T1, typename T2>
void call2(T1 op1, T2 op2);

call2(f1, f2);               // OK
```

C++17: `noexcept` Part of the Function Type

```
// primary template (in general type T is no function):
template<typename T> struct is_function : std::false_type { };

// partial specializations for all function types:
template<typename Ret, typename... Params>
struct is_function<Ret (Params...)> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) &> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const &> : std::true_type { };
...

// partial specializations for all function types with noexcept:
template<typename Ret, typename... Params>
struct is_function<Ret (Params...) noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) & noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const& noexcept> : std::true_type { };
...
```

now **48** instead of **24** partial specializations

C++17: Relaxed Enumeration Initialization

- **For enumerations with a fixed underlying type, you can use an integral value of that type for direct list initializations**
  - applies to both, `enum` and `enum class`

see P0138R2

```
enum class Salutation { mr, mrs };
Salutation s4 = 0;   // Error (all versions)
Salutation s5(0);    // Error (all versions)
Salutation s6{0};    // OK since C++17  (error before C++17)

enum class Salut : char { mr, mrs };
Salut s7 = 0;        // Error (all versions)
Salut s8(0);         // Error (all versions)
Salut s9{0};         // OK since C++17  (error before C++17)

enum Flag { bit1=1, bit2=2, bit3=4 };
Flag f3{0};          // still Error (all versions)

enum Byte : unsigned char { };
Byte b1 = 42;        // Error (all versions)
Byte b2(42);         // Error (all versions)
Byte b3{42};         // OK since C++17  (error before C++17)
Byte b4 = {42};      // Error (all versions)
```

This is, why it was introduced:
- Initialization of new integral types as enum
- See `std::byte`

Initializations with `auto` and {}

- **C++11/C++14:**

```
auto a1(42);      // direct initialization, initializes an int
auto a2{42};      // direct-list-initialization, initializes std::initializer_list<int>
auto a3{1,2};     // direct-list-initialization, initializes std::initializer_list<int>
auto a4 = 42;     // copy initialization, initializes an int
auto a5 = {42};   // copy-list-initialization, initializes an initializer_list<int>
auto a6 = {1,2};  // copy-list-initialization, initializes an initializer_list<int>
```

- **Since C++17 (but some compilers support it before):**

```
auto x1(42);      // direct initialization, initializes an int
auto x2{42};      // direct-list-initialization, initializes an int
auto x3{1,2};     // Error
auto x4 = 42;     // copy initialization, initializes an int
auto x5 = {42};   // copy-list-initialization, still initializes std::initializer_list<int>
auto x6 = {1,2};  // copy-list-initialization, still initializes std::initializer_list<int>
```


C++17

New Library Components

New Basic Data Structures

- **After pair<>, tuple<>, ...
  C++17 adds some smart data types to hold a value:**
  - **std::optional<>**
    - optionally holds a value
    - either empty or has a value of a specific type
    - adapted from boost:optional<>
  - **std::variant<>**
    - holds a value that might have one of multiple pre-defined types
    - usually never empty (only due to some exceptions)
    - differs from boost:variant<>
  - **std::any<>**
    - holds a value that might have any possible type
    - may be empty
    - adapted from boost::any<>

C++17: `std::optional<>`

- **Structure to optionally hold a value of a certain type**
  - either empty or has a value
  - no need to deal with special values such as NULL or -1
- **Adapted from boost::optional**
  - `std::nullopt` instead of `boost::none`

```
std::optional<std::string> noString;
std::optional<std::string> optStr("hello");
optStr = "new value";
*optStr = "new value";
...
if (optStr) {  // equivalent to: if (optStr.has_value())
  std::string s = *optStr;          // undefined behavior if no value
}

try {
  std::string s = optStr.value();   // exception if no value
}
catch (const std::bad_optional_access& e) {
  optStr.reset();    // makes it empty
}
```

C++17: Example for `std::optional<>`

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

std::optional<int> asInt(const std::string& s)  // returns int or "no int"
{
  try {
    return std::stoi(s);
  }
  catch (...) {
    return std::nullopt;
  }
}

int main()
{
  for (auto s : {"42", "  077", "hello", "0x33"} ) {
    std::optional<int> oi = asInt(s);
    if (oi) {
      std::cout << "convert '" << s << "' to int: " << *oi << "\n";   // int
    }
    else {
      std::cout << "can't convert '" << s << "' to int\n";            // "no int"
    }
  }
}
```

Output:

```
convert '42' to int: 42
convert '  077' to int: 77
can't convert 'hello' to int
convert '0x33' to int: 0
```


C++17: `std::optional<>` Operations

Operation	Effect
constructors	Create an optional object (might call constructor for underlying type)
`make_optional<>()`	Create an optional object (passing value to initialize it)
destructor	Destroys an optional object
`=`	Assign a new value
`emplace()`	Assign a new value to the underlying type
`reset()`	Destroys any value
`has_value()`	Returns whether the object has a value
conversion to `bool`	Returns whether the object has a value
`*`	Value access (undefined behavior if no value)
`->`	Access to member of the value (undefined behavior if no value)
`value()`	Value access (exception if no value)
`value_or()`	Value access (fallback argument if no value)
`swap()`	Swaps values between two optional objects
`==, !=, <, <=, >, >=`	Compare optional objects
`hash<>`	Function object type to compute hash values

```
// move semantics is supported:
std::optional<std::string> os;
std::string s = "hello";
os = std::move(s);
std::string s2 = *os;
std::string s3 = std::move(*os);

// Comparison work on the values
// - "no value" < any other value
std::optional<bool> ob0;
if (ob < false) ...      // true

std::optional<unsigned> uo;
if (uo < 0) ...          // true

// - different than using as bool
std::optional<bool> ob{false};
if (!ob) ...             // false
if (ob==false) ...       // true

std::optional<int*> op{nullptr};
if (!op) ...             // false
if (op==nullptr) ...     // true
```

C++17: `std::optional<>` Considered Harmful

- **Beware of accessing the contained object for temporaries**
  - Lifetime issues with direct access
  - Don't replace objects by optional objects blindly

```
std::optional<std::string> getString();
...
auto a = getString().value();                  // OK: copy of contained object
auto b = *getString();                         // ERROR: undefined behavior if std::nullopt
const auto& r1 = getString().value();          // ERROR: reference to deleted contained object
auto&& r2 = getString().value();               // ERROR: reference to deleted contained object


std::vector<int> getVector();
...
for (int i : getVector()) {                    // OK
  std::cout << i << '\n';
}

std::optional<std::vector<int>> getOptVector();
...
for (int i : getOptVector().value()) {         // ERROR: iterate over deleted vector
  std::cout << i << '\n';
}
```


Declaration of std::optional<>

```
template <typename T>
class optional {
  public:
   using value_type = T;
   constexpr optional() noexcept;
   ...
   // observers:
   constexpr const T& operator*() const&;
   constexpr T& operator*() &;
   constexpr T&& operator*() &&;
   constexpr const T&& operator*() const&&;

   constexpr const T& value() const&;
   constexpr T& value() &;
   constexpr T&& value() &&;
   constexpr const T&& value() const&&;
   ...
};
```

See http://wg21.link/p0936 for our proposal to declare that the return value depends on the lifetime of ***this**

C++17: `std::variant<>`

- **Close discriminated union**
  - Structure to hold a value of one of the specified "*alternatives*"
  - Unlike `union`:
    - The type of the current value is always known
    - Can have any member of any type
    - Can be base class
  - Multiple occurrences of same type are supported
    - Semantic union (e.g. different strings for different database columns)
    - Interface with index like tuple
  - No empty state
    - Can use `std::monostate` to simulate it
  - Default-Constructible if first "alternative" is default constructible
    - Can use `std::monostate` for it
  - Differs from boost::variant
    - No heap allocation

C++17: Example for `std::variant<>`

```
#include <variant>

std::variant<int, int, std::string> var; // sets first int to 0, index()==0
var = "hello";                           // sets string, index()==2
var.emplace<1>(42);                      // sets second int, index()==1

try {
  auto s = std::get<std::string>(var);   // throws exception (second int currently set)
  auto a = std::get<double>(var);        // compile-time error: no double
  auto b = std::get<3>(var);             // compile-time error: no 4th alternative
  auto c = std::get<int>(var);           // compile-time error: int twice
  auto i = std::get<1>(var);             // OK, i1==42
  auto j = std::get<0>(var);             // throws exception (other int currently set)

  std::get<1>(var) = 77;                 // OK, because second int already set
  std::get<0>(var) = 99;                 // throws exception (other int currently set)
}
catch (const std::bad_variant_access& e) {      // if runtime error
  std::cout << "Exception: " << e.what() << '\n';
}
```

C++17: `std::variant<>` Operations

Operation	Effect
constructors	Create a variant object (might call constructor for underlying type)
destructor	Destroys an variant object
`=`	Assign a new value
`emplace()`	Assign a new value to the underlying actual type
`valueless_by_exception()`	Returns whether the variant has no value due to an exception
`index()`	Returns the index of the current alternative
`swap()`	Swaps values between two variant objects
`==, !=, <, <=, >, >=`	Compare variant objects
`hash<>`	Function object type to compute hash values
`holds_alternative<T>()`	Returns whether there is a value for type T
`get<T>()`	Returns the value for the alternative with type T
`get<Idx>()`	Returns the value for the alternative with index Idx
`get_if<T>()`	Returns a pointer to the value for the alternative with type T or nullptr
`get_if<Idx>()`	Returns a pointer to the value for the alternative with index Idx or nullptr
`visit()`	Perform operation for the current alternative

C++17: Tricky Usage for `std::variant<>`

```
#include <variant>

// explicitly choose the alternative:
std::variant<int, long> var{std::in_place_index<1>, 11};
                                        // inits long, although int passed, index()==1
var = 22;                               // sets int, index()==0
var.emplace<1>(32);                     // sets long although int passed, index()==1


// initialize variant with a set with lambda as sorting criterion:
auto sc = [] (int x, int y) {
             return std::abs(x) < std::abs(y);
           };
std::variant<std::vector<int>,
             std::set<int,decltype(sc)>> v8{std::in_place_index<1>,
                                            {4, 8, -7, -2, 0, 5},
                                            sc};


if (auto ip = std::get_if<1>(&var); ip) {   // note: pass the address!
  std::cout << "*ip: " << *ip << '\n';
}
```

new `if` with initialization

C++17: std::variant<> and std::monostate

```
struct NoDefConstr {
  NoDefConstr(int i) {
  }
};

std::variant<NoDefConstr, int> v1;     // ERROR: can't default construct first type

std::variant<std::monostate, NoDefConstr> v2;   // OK, value is monostate{}
std::cout << "index: " << v2.index() << '\n';   // print: 0

// check for monostate:
if (v2.index() == 0) { ... }
if (!v2.index()) { ... }   // monostate must be first alternative
if (std::holds_alternative<std::monostate>(v2)) { ... }
if (std::get_if<0>(&v2)) { ... }
if (std::get_if<std::monostate>(&v2)) { ... }

v2 = 42;
v2 = std::monostate{};      // monostate again
```

a way to signal emptiness

C++17: std::variant<> Becoming Empty

- **variant<> might become empty only by unhandled exception**
  - Only if using the variant after exception without re-initialization
- **Then:**

```
v.valueless_by_exception() == true
v.index() == std::variant_npos
```

```
struct S {
  operator int() { throw "EXCEPTION"; }    // any conversion to int throws
};

std::variant<float,int> var{12.2};
try {
  var.emplace<1>(S());                     // oops, throws while being set
}
catch (...) {
  // oops, really don't set var to a new defined state?
}
// var.valueless_by_exception() == true
// var.index() == std::variant_npos
```

C++17: Deriving from std::variant<>

```
class Derived : public std::variant<int, std::string> {
};

Derived d = {{"hello"}};                  // aggregate initialization

std::cout << d.index() << '\n';           // prints: 1
std::cout << std::get<1>(d) << '\n'; // prints: hello

d.emplace<0>(77);
std::cout << std::get<0>(d) << '\n'; // prints: 77
```


C++17: std::variant<> Visitors

- **Using visitors:**

```
std::variant<int, std::string> var(42);
…

struct MyVisitor
{
  void operator()(double d) const {
    std::cout << d << '\n';
  }
  void operator()(int i) const {
    std::cout << i << '\n';
  }
  void operator()(const std::string& s) const {
    std::cout << s << '\n';
  }
};
std::visit(MyVisitor(), var);      // compile-time error if not all possible types supported
                                   // or with ambiguities

std::visit([](const auto& val) { // call lambda for the appropriate type
               std::cout << val << '\n';
           },
           var);
```

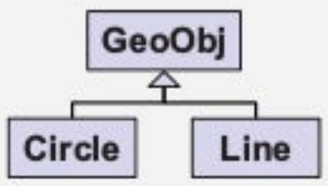
C++17: `std::variant<>` Visitors

- **Multi-Visitors**

```
struct MyVisitor {
  template <typename T>          // visitor for single value of any type
  void operator() (T& var) {
    std::cout << var;
  }
  void operator() (int i, double d, char c) {   // for specific 3 values
    std::cout << i << ' ' << d << ' ' << c;
  }
  void operator() (...) {        // for all other cases
    std::cout << "no match";
  }
};
std::variant<int, std::string> var1{12};
std::variant<std::string, double> var2{13};
std::variant<std::string, char> var3{'x'};
std::visit(MyVisitor(), var1, var2, var3);   // prints:  12 13.0 x
var2 = "hello";
std::visit(MyVisitor(), var1, var2, var3);   // prints:  no match
std::visit(MyVisitor(), var2);               // prints:  hello
```


Runtime Polymorphism Example with Heap Memory

```
class GeoObj {
 public:
  virtual void move(Coord) = 0;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  ...
};
```



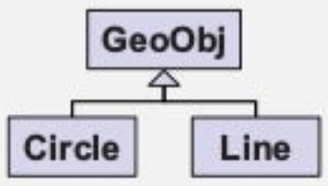

```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle (Coord c, int r);
  virtual void move(Coord c) override;
  virtual void draw() const override;
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line (Coord f, Coord t);
  virtual void move(Coord c) override;
  virtual void draw() const override;
};
```

```
std::vector<GeoObj*> createFig()
{
  std::vector<GeoObj*> f;
  Line* lp = new Line(Coord(1,2), Coord(3,4));
  Circle* cp = new Circle(Coord(5,5), 2);
  f.push_back(lp);
  f.push_back(cp);
  return f;
}

void drawElems (const std::vector<GeoObj*>& v)
{
  for (const auto& geoobjptr : v) {
    geoobjptr->draw();
  }
}

std::vector<GeoObj*> fig = createFig();
drawElems(fig);
for (const auto& geoobjptr : fig) {
  geoobjptr->move(Coord(2,2));
}
drawElems(fig);

// remove all elements in the vector:
for (auto& geoobjptr : fig) {
  delete geoobjptr;
  geoobjptr = nullptr;   // NULL before C++11
}
fig.clear();
```

Polymorphism Example with `std::variant<>`

- no pointers
- no new/delete in appl. code
- objects located together

```
class GeoObj {
 public:
  virtual void move(Coord) = 0;
  virtual void draw() const = 0;
  virtual ~GeoObj() = default;
  ...
};
```




```
class Circle : public GeoObj {
 private:
  Coord center;
  int rad;
 public:
  Circle (Coord c, int r);
  virtual void move(Coord c) override;
  virtual void draw() const override;
};

class Line : public GeoObj {
 private:
  Coord from;
  Coord to;
 public:
  Line (Coord f, Coord t);
  virtual void move(Coord c) override;
  virtual void draw() const override;
};
```

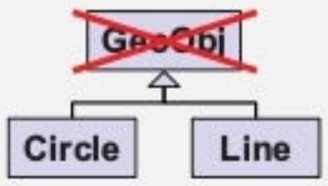
```
using GeoObjVar = std::variant<Circle, Line>;

std::vector<GeoObjVar> createFig()
{
  std::vector<GeoObjVar> f;
  f.push_back(Line(Coord(1,2),Coord(3,4)));
  f.push_back(Circle(Coord(5,5),2));
  return f;
}

void drawElems (const std::vector<GeoObjVar>& v)
{
  for (const auto& geoobj : v) {
    std::visit([] (const auto& obj) {
                 obj.draw();
               },
               geoobj);
  }
}

std::vector<GeoObjVar> fig = createFig();
drawElems(fig);
for (auto& geoobj : fig) {
  std::visit([] (auto& obj) {
               obj.move(Coord(2,2));
             },
             geoobj);
}
drawElems(fig);

fig.clear();    // remove all elements in the vector
```

visitor uses a local vtable

Polymorphism Example with `std::variant<>`

- no pointers
- no new/delete in appl. code
- objects located together

- no common base class
- no virtual functions




```
class Circle {
 private:
  Coord center;
  int rad;
 public:
  Circle (Coord c, int r);
  void move(Coord c);
  void draw() const;
};

class Line {
 private:
  Coord from;
  Coord to;
 public:
  Line (Coord f, Coord t);
  void move(Coord c);
  void draw() const;
};
```

```
using GeoObjVar = std::variant<Circle, Line>;

std::vector<GeoObjVar> createFig()
{
  std::vector<GeoObjVar> f;
  f.push_back(Line(Coord(1,2),Coord(3,4)));
  f.push_back(Circle(Coord(5,5),2));
  return f;
}

void drawElems (const std::vector<GeoObjVar>& v)
{
  for (const auto& geoobj : v) {
    std::visit([] (const auto& obj) {
                 obj.draw();
               },
               geoobj);
  }
}

std::vector<GeoObjVar> fig = createFig();
drawElems(fig);
for (auto& geoobj : fig) {
  std::visit([] (auto& obj) {
               obj.move(Coord(2,2));
             },
             geoobj);
}
drawElems(fig);

fig.clear();    // remove all elements in the vector
```

visitor uses a local vtable

C++17: `std:any`

- **Type to hold any possible value of any type**
  - may be empty
- **Adapted from boost::any**

```
std::any empty;
std::any anyVal(42);
anyVal = std::string("hello");
anyVal = "oops";                    // Beware: type is const char*, so stores the address
...
if (anyVal.has_value()) {  // 'if (anyVal)' not supported
  if (anyVal.type() == typeid(std::string)) {  // 'type()==typeid(void)' if empty
    std::string s = std::any_cast<std::string>(anyVal);
  }
}

try {
  int i = std::any_cast<int>(anyVal);   // type must match exactly (only const/refs ignored)
}
catch (std::bad_any_cast&) {
  anyVal.reset();    // makes it empty
}
```


C++17: `std::any` Operations

Operation	Effect
constructors	Create a any object (might call constructor for underlying type)
`make_any()`	Create a any object (passing value(s) to initialize it)
destructor	Destroys an any object
`=`	Assign a new value
`emplace()`	Assign a new value to the underlying actual type
`reset()`	Destroys any value (makes the object empty)
`has_value()`	Returns whether the object has a value
`type()`	Returns the current type as `std::type_info` object
`any_cast<T>()`	Use any abject as value of type T (exception if other type)
`swap()`	Swaps values between two optional objects

C++17: Move Semantics with `std:any`

- **Types in std::any must be copyable**
  - Move-only types are not supported
- **But you can use move semantics**

```
std::string s("a pretty long string value (disabling SSO)");
std::cout << "s: " << s << '\n';                    // outputs the initialized string

std::any a;
a = std::move(s);                                   // move s into a
std::cout << "s: " << s << '\n';                    // probably outputs an empty string
…
s = std::any_cast<std::string>(std::move(a));       // move string in a into s
std::cout << "s: " << s << '\n';                    // outputs the original string
```


C++17: `std::byte`

see P0298R3

- **Type representing a byte**
  - Not an integer, not a character
  - Operators: `=, <<, <<=, >>, >>=, |, |=, &, &=, ^, ^=, ~, ==, !=, <, <=, >, >=, sizeof`
    - Plus: `to_integer<`*integraltype*`>(`*byte*`)`

```
std::byte b1{0xFF};                 // OK (as for all enums with fixed underlying type since C++17)
std::byte b2{0b1111'0000};

std::byte[4] b4{0xFF, 0, 0, 0};
if (b1 == b4[0]) {
  b1 <<= 4;
}

std::byte b0;                       // undefined value
std::byte bx(42);                   // Error
std::byte by = 42;                  // Error

if (b1) …                           // Error
if (b1 != std::byte{0}) …           // OK
if (to_integer<bool>(b2)) …         // Error (ADL doesn't work here)
if (std::to_integer<bool>(b2)) …    // OK

std::cout << sizeof(b0);            // always 1
```

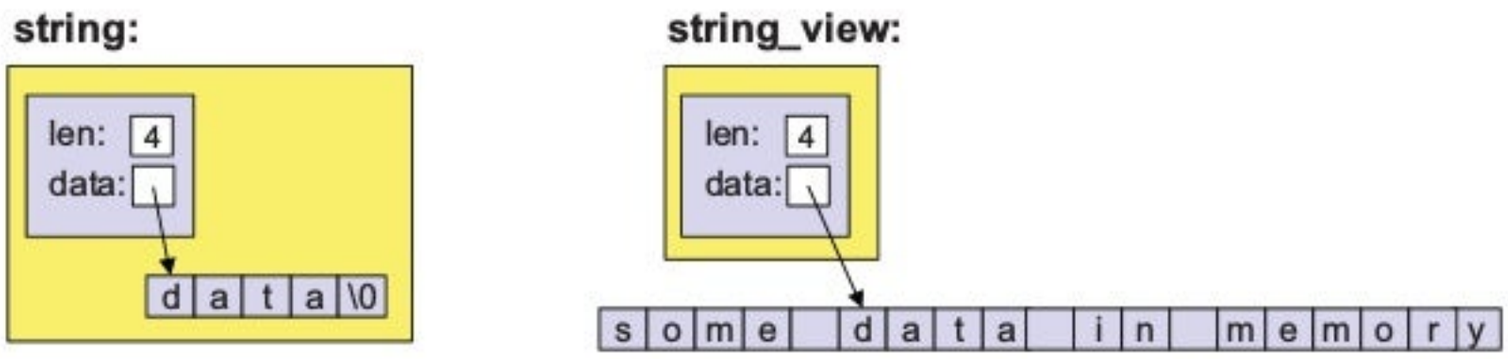
C++17: std::byte Definition in the Standard

```
namespace std {
  enum class byte : unsigned char {};

  // 21.2.5, byte type operations
  template <typename IntType>
   constexpr byte  operator<< (byte b, IntType shift) noexcept;
  template <typename IntType>
   constexpr byte& operator<<= (byte& b, IntType shift) noexcept;
  template <typename IntType>
   constexpr byte  operator>> (byte b, IntType shift) noexcept;
  template <typename IntType>
   constexpr byte& operator>>= (byte& b, IntType shift) noexcept;
  constexpr byte&  operator|= (byte& l, byte r) noexcept;
  constexpr byte   operator| (byte l, byte r) noexcept;
  constexpr byte&  operator&= (byte& l, byte r) noexcept;
  constexpr byte   operator& (byte l, byte r) noexcept;
  constexpr byte&  operator^= (byte& l, byte r) noexcept;
  constexpr byte   operator^ (byte l, byte r) noexcept;
  constexpr byte   operator~ (byte b) noexcept;
  template <typename IntType>
    constexpr IntType to_integer (byte b) noexcept;
}
```


C++17: `std::string_view`

- **Handle for read-only character sequence**
  - Lifetime of the data not controlled be the object
  - No allocator support
  - Passing by value is cheap
- **Differs from strings as follows:**
  - Not guaranteed to be null terminated (no NTBS)
    - You can place a '\0' as last character, though
  - Value is `nullptr` after default construction
    - Then: `data()==nullptr`, `size()==0`

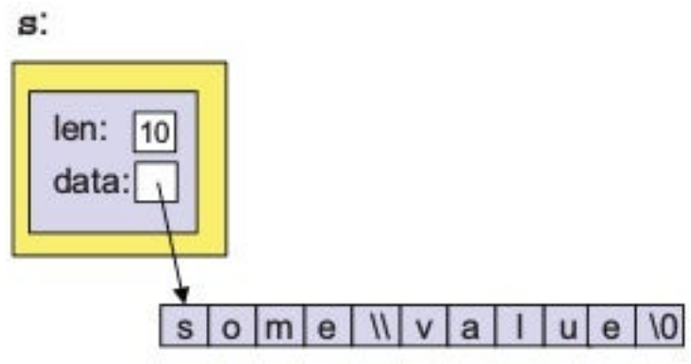
- Always use `size()` before using the data (`operator[]`,`data()`,...)



C++17: `std::string_view`

- **Header:** `<string_view>`
- **Has some typical string support:**
  - Also: `u16string_view`, `u32string_view`, `wstring_view`
  - Corresponding literal operator is defined
    - Suffix: **`sv`**
  - `std::quoted()` is supported
  - Hash values match with hash values of `std::string`
- **Still open integrations:**
  - No regex support

```
#include <chrono>
using namespace std::literals;

auto s = R"(some\value)"sv;
std::cout << quoted(s);   // output: "some\\value"
```



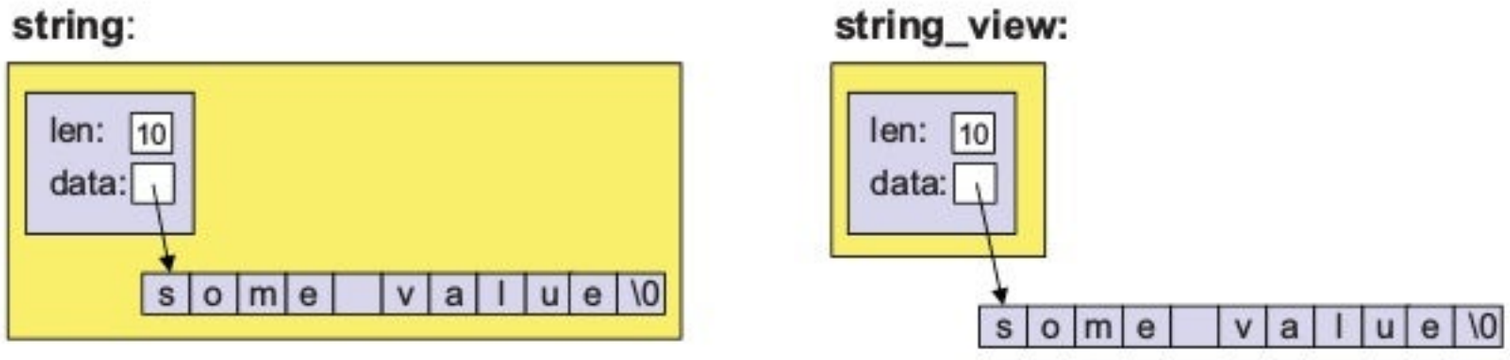
C++17: `std::string_view` Operations

Operation	Effect
constructors	Create or copy a string view
destructor	Destroys a string view
`=`	Assign a new value
`swap()`	Swaps values between two strings view
`==, !=, <, <=, >, >=, compare()`	Compare string views
`empty()`	Returns whether the string view is empty
`size(), length()`	Return the number of characters
`max_size()`	Returns the maximum possible number of characters
`[], at()`	Access a character
`front(), back()`	Access the first or last character
`<<`	Writes the value to a stream
`copy()`	Copies or writes the contents to a character array
`data()`	Returns the value as constant C-string or character array (note: no terminating null character)
`substr()`	Returns a certain substring
find functions	Search for a certain substring or character
`begin(), end()`	Provide normal iterator support
`cbegin(), cend()`	Provide constant iterator support
`rbegin(), rend()`	Provide reverse iterator support
`crbegin(), crend()`	Provide constant reverse iterator support
`hash<>`	Function object type to compute hash values
`remove_prefix()`	Remove leading characters } unlike std::string
`remove_suffix()`	Remove trailing characters } unlike std::string

C++17: `std::string_view`

- **Conversion string => string_view is cheap**
  => Implicit conversion
- **Conversion string_view => string is expensive**
  => Explicit conversion

```
void foo_s (const string& s);
void foo_v (string_view sv);

foo_s("some value");    // computes length, allocates memory, copies characters
foo_v("some value");    // computes length only
```



C++17: Converting API's from `string` to `string_view`

```
#include <string>

template<typename Coll>
void printElems(const Coll& coll, const std::string& prefix)
{
  // print each element with a leading prefix:
  for (const auto& elem : coll) {
    std::cout << prefix << elem << '\n';
  }
}
```

```
// saves 1 new/malloc for:
printElems(myVector,
           "- Coll Element: ");

// still works as before for:
std::string getPrefix(...);
...
printElems(myVector, getPrefix());
```

```
#include <string_view>

template<typename Coll>
void printElems(const Coll& coll, std::string_view prefix)
{
  // print each element with a leading prefix:
  for (const auto& elem : coll) {
    std::cout << prefix << elem << '\n';
  }
}
```

Beware: Only OK because:
- no null terminator required
- `data()` not used (`nullptr`!)
- lifetime OK

C++17: Converting API's from `string` to `string_view`

```
#include <string>

template<typename Coll>
void printElems(const Coll& coll, const std::string& prefix)
{
  // print each element with a leading prefix:
  for (const auto& elem : coll) {
    std::cout << prefix << elem << '\n';
  }
}
```

```
#include <string_view>

template<typename Coll>
void printElems(const Coll& coll, std::string_view prefix)
{
  // print each element with a leading prefix:
  for (const auto& elem : coll) {
    std::cout << prefix << elem << '\n';
  }
}
```

```
// with both provided:
printElems(myVector,
           "- Coll Element: ");
```

Error: ambiguous

C++17: Converting API's from `string` to `string_view`

```
#include <string>

class Customer {
  std::string name;
 public:
  Customer (const std::string& n) : name(n) {
  }
  …
};
```

Don't use `std::string_view` in call sequences for `std::string`

```
#include <string>
#include <string_view>

class Customer {
  std::string name;
 public:
  Customer (std::string_view n) : name(n) {
  }
  …
};
```

```
// saves 1 new/malloc for:
Customer c1("Enterprise Applications");

// but adds 1 new/malloc for:
std::string getName(…);
…
Customer c2(getName());
```

C++17: Initializing Data Members since C++11

```
#include <string>

class Customer {
  std::string name;
 public:
  Customer (const std::string& n) : name(n) {
  }
  …
};
```

Constructors should move initialize members from by-value parameters

```
// saves 1 new/malloc for:
Customer c1("Enterprise Applications");

// and saves any new/malloc for:
std::string getName(…);
…
Customer c2(getName());
```

```
#include <string>
#include <string_view>

class Customer {
  std::string name;
 public:
  Customer (std::string n) : name(std::move(n)) {
  }
  …
};
```


C++17: string_view Considered Harmful

- **NEVER use std::string_view as return type**
  - String views are as bad as pointers (worse than references)

```
std::string_view concat (std::string_view sv1, std::string_view sv2) {
  return std::string(sv1) + std::string(sv2);   // Fatal Runtime ERROR:
}                                               //  return value refers to memory of destructed string

std::string str = "hello";

std::cout << concat(str, "world") << '\n';      // fatal, but compiles
std::string s1 = concat(str, "world");          // fatal, but compiles
auto s2 = concat(str, "world");                 // fatal, but compiles
auto&& s3 = concat(str, "world");               // fatal, but compiles
std::string_view sv = concat(str, "world");     // fatal, but compiles
```

- Not even compiler warnings (yet)
- There might come something with C++20:
  - mark result of conversion form `string` to `string_view` as lifetime-dependent
  - see http://wg21.link/p0936

C++17: string_view Considered Harmful

- **Prefer `auto` as return type in templates**
- **Avoid `auto` for initializations from return values**

```
std::string operator+ (std::string_view sv1, std::string_view sv2) {
  return std::string(sv1) + std::string(sv2);
}

template<typename T>
T concat (const T& x, const T& y) {
    return x + y;
}

std::string_view hi = "hi";
auto xy = concat(hi, hi);   // OOPS: xy is std::string_view
std::cout << xy << '\n';    // Run-time ERROR (returned string already destructed)
```

better declare the **return type** of template as `auto`

better not declare **object** initialized with return value as `auto`

C++17

Filesystem Library

C++17: Filesystem Library

- **C++ API to deal with files, directories, paths, ...**
  - Adapted from boost
  - Make filesystem operations as portable as possible
    - Encourages behavior as defined by POSIX
    - Implementations have to document differences and report an error if there is not reasonable behavior
- **Modifications against Boost / Filesystem TS:**
  - Root elements are not longer filenames
  - A path ending with `/` no longer yields `"."` as last element
- **Extensions:**
  - Relative path functions
    - lexically (according to syntax) or operational (according to file system)
    - with and without absolute path as fallback

C++17: Filesystem: Simple Example

No change from Boost/FS-TS API except header file and namespace

```
#include <iostream>
#include <filesystem>

int main (int argc, char* argv[])
{
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " <path> \n";
    return EXIT_FAILURE;
  }

  std::filesystem::path p(argv[1]);   // p represents a filesystem path (might not exist)
  if (exists(p)) {                    // does path p actually exist?
    if (is_regular_file(p)) {         // is path p a regular file?
      std::cout << " size of " << p << " is " << file_size(p) << '\n';
    }
    else if (is_directory(p)) {       // is path p a directory?
      std::cout << p << " is a directory containing:\n";
      for (auto& e : std::filesystem::directory_iterator(p)) {
        std::cout << "   " << e.path() << '\n';
      }
    }
    else {
      std::cout << p << " exists, but is neither regular file nor directory\n";
    }
  }
  else {
    std::cout << p << " doesn't exist\n";
  }
}
```

Might Output:

```
"/" is a directory containing:
  "/proc"
  "/dev"
  ...
```

C++17: Filesystem: Simple Example

```
#include <iostream>
#include <filesystem>

int main (int argc, char* argv[])
{
  if (argc < 2) {
    …
  }

  namespace fs = std::filesystem;

  fs::path p(argv[1]);   // p represents a filesystem path (might not exist)
  if (exists(p)) {                          // does path p actually exist?
    if (is_regular_file(p)) {               // is path p a regular file?
      std::cout << " size of " << p << " is " << file_size(p) << '\n';
    }
    else if (is_directory(p)) {             // is path p a directory?
      std::cout << p << " is a directory containing:\n";
      for (auto& e : fs::directory_iterator(p)) {
        std::cout << "  " << e.path() << '\n';
      }
    }
    else {
      std::cout << p << " exists, but is neither regular file nor directory\n";
    }
  }
  else {
    std::cout << p << " doesn't exist\n";
  }
}
```

Common convention for filesystem namespace

Might Output:

```
"/" is a directory containing:
  "/proc"
  "/dev"
  …
```


C++17: Filesystem: Modifying Example

```
#include <iostream>
#include <fstream>
#include <filesystem>

int main ()
{
  try {
    std::filesystem::path tmpDir("/tmp");
    std::filesystem::path testPath = tmpDir / "test";

    if (!create_directory(testPath)) {
      std::cout << "test directory already exists" << '\n';
    }

    testPath /= "data.txt";
    std::ofstream dataFile(testPath.string());
    if (dataFile) {
      dataFile << "the answer is 42\n";
    }

    create_symlink(testPath, std::filesystem::path("slinkToTestDir"));
  }
  catch (std::filesystem::filesystem_error& e) {
    std::cout << "error creating " << e.path1() << ": " << e.what() << '\n';
  }
}
```

No change from Boost/FS-TS API except header file and namespace

C++17: Switching over Types of an Initialized Filesystem Path

```
void checkFilename(const std::string& name)
{
  namespace fs = std::filesystem;

  switch (fs::path p(name); status(p).type()) {

   case fs::file_type::not_found:
     std::cout << p << " not found\n";
     break;

   case fs::file_type::directory:
     std::cout << p << ":\n";
     for (const auto& entry : fs::directory_iterator(p)) {
        std::cout << "- " << entry.path() << '\n';
     }
     break;

   default:
     std::cout << p << " exists\n";
     break;
  }
}
```

new `switch` with initialization

How to use the Filesystem Library

- **Use boost:**
  - Boost Filesystem support (close to Filesystem TS)

```
#include <boost/filesystem.hpp>
namespace std {
  namespace filesystem = boost::filesystem;
}
```

- **Use gcc/g++:**
  - C++17 Filesystem support
  - since g++ 8.0 with `-lstdc++fs`

```
#include <filesystem>
```

- **Use Visual C++:**
  - Experimental Filesystem TS support
  - since VS2015

```
#include <experimental/filesystem>
namespace std {
  namespace filesystem = std::experimental::filesystem;
}
```

C++17: Filesystem: Principles

- **Common namespace convention:**
  - `namespace fs = std::namespace;`

- **Member functions versus free-standing functions**
  - Member functions are cheap
    - Pure lexical operations (e.g. `p.is_absolute()`)
  - Free-standing functions are expensive
    - Take actual filesystem into account (e.g. `equivalent(p1, p2)`)

- **Flexible error handling**
  - Functions might:
    - return something special on specific errors and otherwise throw `std::filesystem::filesystem_error`
    - return error code in passed `std::error_code` argument

Filesystem Error Handling

- **Functions might:**
  - return something special on specific errors and otherwise throw `std::filesystem::filesystem_error`
  - return error code in passed `std::error_code` argument

```
try {
  if (!create_directory(p)) {                          // exception on error (unless path exists)
    std::cout << p << " already exists\n";             // path exists
  }
}
catch (const std::filesystem::filesystem_error& e) {   // derived from std::exception
  std::cout << "EXCEPTION: " << e.what() << '\n';
  std::cout << "     path: " << e.path1() << '\n';
}
```

```
std::error_code ec;
if (!create_directory(p, ec)) {                        // set error code on error
  std::cout << "can't create directory " << p << "\n"; // any error occurred
}
if (ec) {                                              // if error code set (due to other error)
  std::cout << "ERROR: " << ec.message() << "\n";
}
if (ec == std::errc::read_only_file_system) {          // if specific error code set
  std::cout << "ERROR: " << p << " is read-only\n";
}
```

C++11 System Error Conditions

C++11 Error Condition	Corresponding Errno Value
address_family_not_supported	EAFNOSUPPORT
address_in_use	EADDRINUSE
address_not_available	EADDRNOTAVAIL
already_connected	EISCONN
argument_list_too_long	E2BIG
argument_out_of_domain	EDOM
bad_address	EFAULT
bad_file_descriptor	EBADF
bad_message	EBADMSG
broken_pipe	EPIPE
connection_aborted	ECONNABORTED
connection_already_in_progress	EALREADY
connection_refused	ECONNREFUSED
connection_reset	ECONNRESET
cross_device_link	EXDEV
destination_address_required	EDESTADDRREQ
device_or_resource_busy	EBUSY
directory_not_empty	ENOTEMPTY
executable_format_error	ENOEXEC
file_exists	EEXIST
file_too_large	EFBIG
filename_too_long	ENAMETOOLONG
function_not_supported	ENOSYS
host_unreachable	EHOSTUNREACH
identifier_removed	EIDRM
illegal_byte_sequence	EILSEQ
inappropriate_io_control_operation	ENOTTY
interrupted	EINTR
invalid_argument	EINVAL
invalid_seek	ESPIPE
io_error	EIO
is_a_directory	EISDIR
message_size	EMSGSIZE
network_down	ENETDOWN
network_reset	ENETRESET
network_unreachable	ENETUNREACH
no_buffer_space	ENOBUFS
no_child_process	ECHILD
no_link	ENOLINK

might be error code `value()`

portable

C++11 Error Condition	Corresponding Errno Value
no_lock_available	ENOLCK
no_message_available	ENODATA
no_message	ENOMSG
no_protocol_option	ENOPROTOOPT
no_space_on_device	ENOSPC
no_stream_resources	ENOSR
no_such_device_or_address	ENXIO
no_such_device	ENODEV
no_such_file_or_directory	ENOENT
no_such_process	ESRCH
not_a_directory	ENOTDIR
not_a_socket	ENOTSOCK
not_a_stream	ENOSTR
not_connected	ENOTCONN
not_enough_memory	ENOMEM
not_supported	ENOTSUP
operation_canceled	ECANCELED
operation_in_progress	EINPROGRESS
operation_not_permitted	EPERM
operation_not_supported	EOPNOTSUPP
operation_would_block	EWOULDBLOCK
owner_dead	EOWNERDEAD
permission_denied	EACCES
protocol_error	EPROTO
protocol_not_supported	EPROTONOSUPPORT
read_only_file_system	EROFS
resource_deadlock_would_occur	EDEADLK
resource_unavailable_try_again	EAGAIN
result_out_of_range	ERANGE
state_not_recoverable	ENOTRECOVERABLE
stream_timeout	ETIME
text_file_busy	ETXTBSY
timed_out	ETIMEDOUT
too_many_files_open_in_system	ENFILE
too_many_files_open	EMFILE
too_many_links	EMLINK
too_many_symbolic_link_levels	ELOOP
value_too_large	EOVERFLOW
wrong_protocol_type	EPROTOTYPE

C++17: Filesystem: UTF8 Support

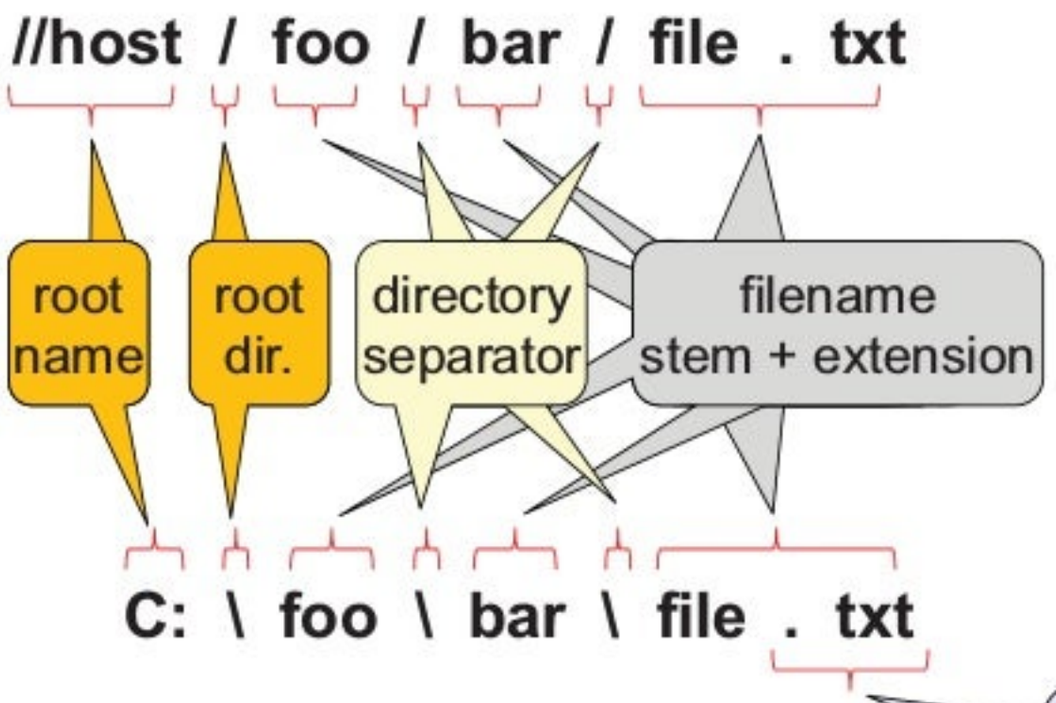
- **Filesystem paths supports UTF8 strings**

```
namespace fs = std::filesystem;

// create directory from returned UTF8 string:
fs::create_directory(fs::u8path(u8"K\u00F6ln"));    // "Köln" (Cologne native)

std::string utf8String = readUTF8String(...);
fs::create_directory(fs::u8path(utf8String));

// store paths as UTF8 string:
std::vector<std::string> utf8paths;
for (const auto& entry : fs::directory_iterator(p)) {
   utf8paths.push_back(entry.path().u8string());
}

// ???: conversion between native and UTF8 character set (must be valid path):
utf8str = std::filesystem::path(str).u8string();   // convert native to UTF8 string
str = std::filesystem::u8path(utf8str).string();   // convert UTF8 to native string
```

`vector<u8string>` with C++20 (introducing `char8_t` for UTF8 chars)

Non-intended usage (hint by Tom Honermann)

Terminology: Elements of a Path



extension
- from last "." til end
- unless stem would be empty (new)

- **Terms:**
  - Root-Name: //host or C: or ...
  - Root-Directory: same as directory separator
  - Directory-Separator: / or \ or // or \\ ...
    - multiple separators are interpreted as one

C++17: Filesystem: Modifying Example

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

int main ()
{
  path p0 = path("foo") / "bar";
  cout << p0 << '\n';

  path p("/foo/bar/data.txt");
  cout << p << '\n';
  p.make_preferred();
  cout << p << '\n';

  cout  << "decomposition:\n";
  cout  << p.root_name() << '\n';
  cout  << p.root_directory() << '\n';
  cout  << p.root_path() << '\n';
  cout  << p.relative_path() << '\n';
  cout  << p.parent_path() << '\n';
  cout  << p.filename() << '\n';
  cout  << p.stem() << '\n';
  cout  << p.extension() << '\n';
  cout  << p.is_absolute() << '\n';
}
```

Unix:	Windows:
"foo/bar"	"foo\bar"
"/foo/bar/data.txt"	"/foo/bar/data.txt"
"/foo/bar/data.txt"	"\foo\bar\data.txt"
""	""
"/"	"\"
"/"	"\"
"foo/bar/data.txt"	"foo\bar\data.txt"
"/foo/bar"	"\foo\bar"
"data.txt"	"data.txt"
"data"	"data"
".txt"	".txt"
1 (true)	0 (false)

Filename Changes for C++17

- **Filenames starting with "." are no extensions**
- **Filename is what is between root or directory separators**
  - Root elements are no filenames
    - `has_filename()` is no longer true for "`//host`" or under Windows: "`C:`"
  - Paths with trailing / have no filenames
    - `has_filename()` is no longer true for "`foo/`"
  - Always: `filename() == stem() + extension()`
- **When iterating over a path, the empty string represents a trailing non-root directory separator**
  - We can now distinguish between root-dir, trailing /, and "." in paths
- **Loops of remove_filename() no longer unwind a path**
  - Use iterators and/or loops over `parent_path()`

Clarify filename() for C++17

	old filename()	old stem()
"."	"."	"."
".."	".."	".."
"/foo/bar.txt"	"bar.txt"	"bar"
"/foo/bar"	"bar"	"bar"
"/foo/bar/.git"	".git"	""
"/foo/bar/."	"."	"."
"/foo/bar/"	"."	"."
"/"	"/"	"/"
"//host"	"//host"	"//host"
"//host/"	"/"	"/"
"C:"	"C:"	"C:" or ""
"C:/"	"." or "/"	"." or "/"
"C:foo/"	"."	"."
"C:."	"C:." or "."	"C:" or "."
""	""	""

or: Unix vs. Windows
- Remember: In Unix "C:" is nothing special

Clarify filename() for C++17

	old filename()	old stem()	C++17 filename()	C++17 has_filename()	C++17 stem()	C++17 extension()
"."	"."	"."	"."	true	"."	""
".."	".."	".."	".."	true	".."	""
"/foo/bar.txt"	"bar.txt"	"bar"	"bar.txt"	true	"bar"	".txt"
"/foo/bar"	"bar"	"bar"	"bar"	true	"bar"	""
"/foo/bar/.git"	".git"	""	".git"	true	".git"	""
"/foo/bar/."	"."	"."	"."	true	"."	""
"/foo/bar/"	"."	"."	""	false	""	""
"/"	"/"	"/"	""	false	""	""
"//host"	"//host"	"//host"	""	false	""	""
"//host/"	"/"	"/"	""	false	""	""
"C:"	"C:"	"C:" or ""	"C:" or ""	true or false	"C:" or ""	""
"C:/"	"." or "/"	"." or "/"	""	false	""	""
"C:foo/"	"."	"."	""	false	""	""
"C:."	"C:." or "."	"C:" or "."	"C:." or "."	true	"C:" or "."	"." or ""
""	""	""	""	false	""	""

C++17: Filesystem: Semantic Changes with C++17

```
#include <experimental/filesystem>   // old lib (from Filesystem TS)
#include <filesystem>                // new lib (since C++17)
#include <iostream>

int main (int argc, char* argv[])
{
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " <path> \n";
    return EXIT_FAILURE;
  }

  namespace fs = std::experimental::filesystem;   // old lib (from Filesy
  namespace fs = std::filesystem;                 // new lib (from C++1

  fs::path p(argv[1]);   // p represents a filesystem path (might or might not
  std::cout << "fn():   " << p.filename() << '\n';
  std::cout << "stem(): " << p.stem() << '\n';        // changed with C++17
  std::cout << "ext():  " << p.extension() << '\n';   // changed with C++17

  while (p.has_filename()) {                          // changed with C++17
    p.remove_filename();                              // changed with C++17
    std::cout << "p:  " << p << '\n';
    std::cout << "fn: " << p.filename() << '\n';
  }
}
```

Alternatives

Experimental Output:

```
$ prog /usr/home/nico/.git
fn():   ".git"
stem(): ""
ext():  ".git"
p:  "\usr\home\nico"
fn: "nico"
p:  "\usr\home"
fn: "home"
p:  "\usr"
fn: "usr"
p:  "\"
fn: "\"
p:  "\"
fn: "\"
p:  "\"
fn: "\"
...
```

C++17: Filesystem: Semantic Changes with C++17

```
#include <experimental/filesystem>   // old lib (from Filesystem TS)
#include <filesystem>                // new lib (since C++17)
#include <iostream>

int main (int argc, char* argv[])
{
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " <path> \n";
    return EXIT_FAILURE;
  }

  Alternatives:
  namespace fs = std::experimental::filesystem;   // old lib (from Filesy
  namespace fs = std::filesystem;                 // new lib (from C++1

  fs::path p(argv[1]);  // p represents a filesystem path (might or might not
  std::cout << "fn():   " << p.filename() << '\n';
  std::cout << "stem(): " << p.stem() << '\n';          // changed with
  std::cout << "ext():  " << p.extension() << '\n';     // changed with

  while (p.has_filename()) {                            // changed with
    p.remove_filename();                                // changed with
    std::cout << "p:  " << p << '\n';
    std::cout << "fn: " << p.filename() << '\n';
  }
}
```

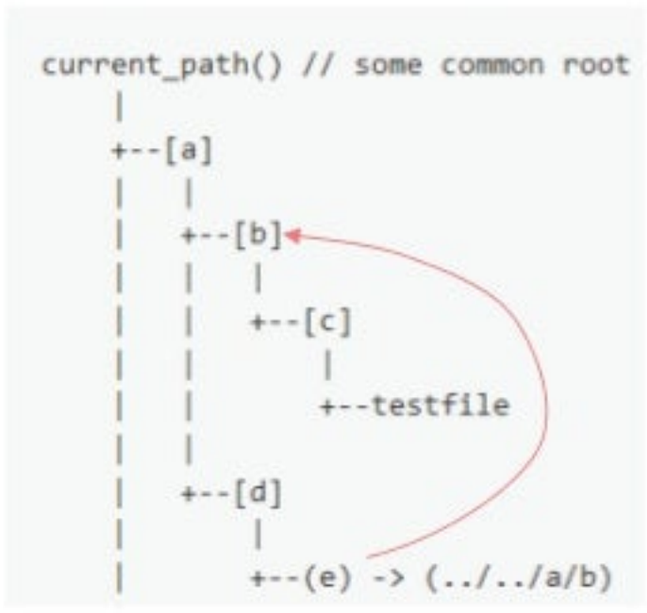
Experimental Output:

```
$ prog /usr/home/nico/.git
fn():   ".git"
stem(): ""
ext():  ".git"
p:  "\usr\home\nico"
fn: "nico"
p:  "\usr\home"
fn: "home"
p:  "\usr"
fn: "usr"
p:  "\"
fn: "\"
p:  "\"
fn: "\"
p:  "\"
fn: "\"
...
```

Output since C++17:

```
$ prog /usr/home/nico/.git
fn():   ".git"
stem(): ".git"
ext():  ""
p:  "/usr/home/nico/"
fn: ""
```

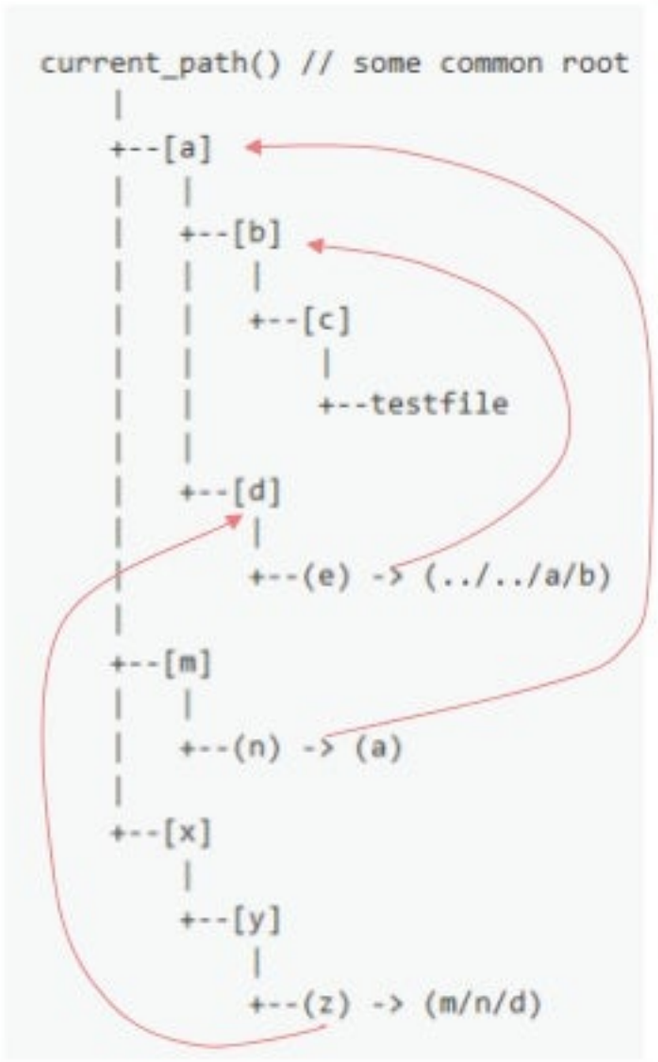

Examples with Symbolic Links



Command-line shells are cheating:

- After `cd a/d/e`
  - I am in `a/b`
  - But shells `pwd` claim to be in `a/d/e`
    - `ls ../e` is not an error
    - `cd ../e` is not an error
- Try:
  `cd -P`
  `pwd -P`

Examples with Symbolic Links

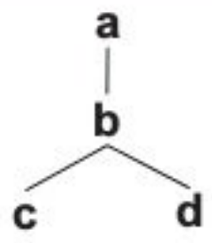


Note some command-line shells lie:

- After `cd a/d/e`
  - I am in `a/b`
  - But shells claim I am in `a/d/e`
    - `ls ../e` is not an error
    - `cd ../e` is not an error
- Try: `pwd -P` and `cd -P`

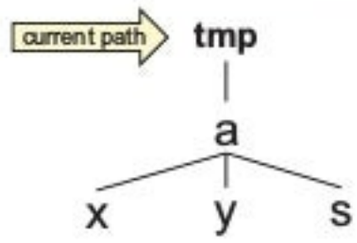
Start (From)	Path (To)	Relative	Lexically Relative
"x/y/z"	"a/b/c/testfile"	"../b/c/testfile"	"../../../a/b/c/testfile"
"m/n"	"a/b/c/testfile"	"./b/c/testfile"	"../../a/b/c/testfile"
"a/b/c/testfile"	"m/n"	"../../.."	"../../../../m/n"
"a/b/c/testfile"	"a/d/e"	"../.."	"../../../d/e"
"a/b/c/testfile"	"a/d"	"../../../d"	"../../../d"
"x/y"	"a/d/e"	"../../a/b"	"../../a/d/e"
"a/d/e"	"x/y"	"../../x/y"	"../../../x/y"

C++17: Filesystem: Relative Path Function




```
namespace fs = std::filesystem;

fs::path from("/a/b/c");
fs::path to("/a/b/d");

fs::path rel = fs::lexically_relative(to, from);   // cheap
         rel = fs::relative(to, from);             // expensive
std::cout << rel << '\n';                          // outputs: "../d"

auto abs = fs::absolute(from/rel);
std::cout << abs << '\n';                          // outputs: "/a/b/c/../d"
std::cout << to << '\n';                           // outputs: "/a/b/d"

std::cout << "equal: " << (abs == to) << '\n';     // outputs: equal: 0   (false)

std::cout << "equivalent: ";
if (fs::exists(abs) && fs::exists(to)) {
  std::cout << fs::equivalent(abs, to) << '\n';    // error if one doesn't exist
}
else {
  std::cout << (abs.lexically_normal() == to.lexically_normal()) << '\n';
}
```

C++17: Filesystem: Path Example

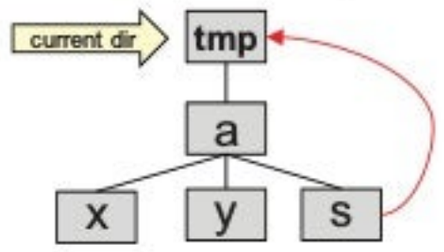



```
std::filesystem::path px("/tmp/a/x");
std::filesystem::path py("/tmp/a/y");

std::filesystem::current_path("/tmp");          // change current path
std::cout << std::filesystem::current_path(); // "/tmp"

std::cout << px.relative_path();                // path from top without root: "tmp/a/x"

std::cout << px.lexically_relative(py);         // to px from py: "../x"
std::cout << relative(px,py);                   // to px from py: "../x"
std::cout << relative(px);                      // to px from current path: "a/x"

std::filesystem::path s{"/tmp/a/s"};
std::cout << px.lexically_relative(s);          // to px from s: "../x"
std::cout << relative(px, s);                   // to px from s: "../x"
```

cheap

expensive

C++17: Filesystem: Path Example




```
std::filesystem::path px("/tmp/a/x");
std::filesystem::path py("/tmp/a/y");

std::filesystem::current_path("/tmp");          // change current path
std::cout << std::filesystem::current_path(); // "/tmp"

std::cout << px.relative_path();                // path from top without root: "tmp/a/x"

std::cout << px.lexically_relative(py);         // to px from py: "../x"
std::cout << relative(px,py);                   // to px from py: "../x"
std::cout << relative(px);                      // to px from current path: "a/x"

std::filesystem::path s{"/tmp/a/s"};
std::cout << px.lexically_relative(s);          // to px from s: "../x"
std::cout << relative(px, s);                   // to px from s: "../x"

create_directories(px);
create_directories(py);
std::filesystem::create_symlink("/tmp", s);
std::cout << px.lexically_relative(s);          // to px from s: "../x"
std::cout << relative(px, s);                   // to px from s: "a/x"  (slink resolved!)
```

cheap

expensive

C++17

Library Extensions and Modifications

C++17: Parallel Execution of STL Algorithms

see N3724 N4352

- **Execution of STL algorithms in parallel**
- **Different execution policies:**
  - sequential
  - parallel
  - parallel sequenced (vectorized)
- **Note:**
  - All algorithms require at least forward iterators (P0467R2)
  - Supplementary algorithms for parallelization

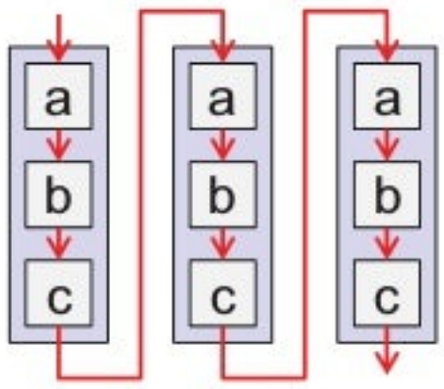
- **Beta implementations:**
  - Microsoft: http://parallelstl.codeplex.com
  - HPX: http://stellar-group.github.io/hpx/docs/html/hpx/manual/parallel.html
  - Codeplay: http://github.com/KhronosGroup/SyclParallelSTL
  - HSA: http://www.hsafoundation.com/hsa-for-math-science
  - Thibaut Lutz: http://github.com/t-lutz/ParallelSTL
  - NVIDIA: http://github.com/n3554/n3554

Using Parallel Algorithms

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>   // for the execution policy

template <typename C, typename P>
void count(const C& coll, P execpolicy)
{
  // count elements with even value:
  auto num = std::count_if(execpolicy,                    // execution policy
                           coll.cbegin(), coll.cend(),    // range
                           [](int elem){                  // criterion
                             return elem % 2==0;
                           });
  std::cout << "number of elements with even value: " << num << '\n';
}

int main()
{
  std::vector<int> coll;
  …
  count(coll, std::execution::seq);
  count(coll, std::execution::par);
  count(coll, std::execution::par_unseq);
}
```

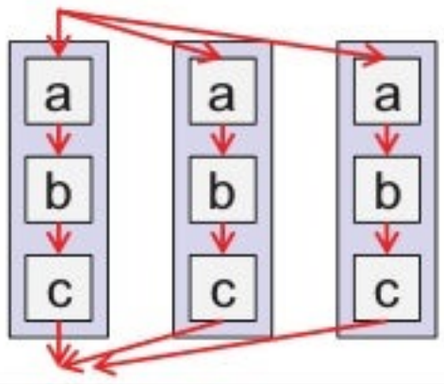

C++17: Parallel Execution Policies

Sequential execution:



- One thread executes all tasks, one after the other
- Sequenced from begin to end

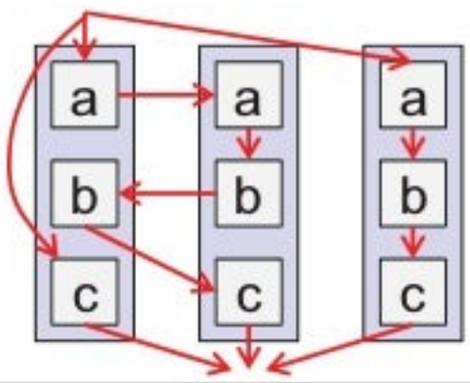
Parallel sequenced execution:



- Multiple threads might execute tasks
- Tasks are processed sequenced from begin to end

⇒ All steps must be able to be done concurrently

Parallel unsequenced execution:



- Multiple threads might execute tasks
- Threads might execute parts of other tasks before ending the first task

⇒ Don't lock on the same mutex

⇒ No memory (de)allocation

C++17: When to use which Parallel Execution Policy

```
transform (std::execution::par_unseq,
           coll1.begin(),coll1.end(),    // source range
           coll2.begin(),                // destination range
           [] (auto x) {                 // operation
                auto y = x * x;          // OK, for each element the only requirement is
                return y;                //      to keep the order of statements
           });

transform (std::execution::par,
           coll1.begin(),coll1.end(),    // source range
           coll2.begin(),                // destination range
           [] (auto x) {                 // operation
                std::lock_guard lg(m);   // OK for par but not for par_unseq
                return x*x;
           });

transform (std::execution::seq,
           coll1.begin(),coll1.end(),    // source range
           coll2.begin(),                // destination range
           [] (auto x) {                 // operation
                logFile << x;            // concurrent access to files is a race
                return x*x;
           });
```


Parallel Algorithms Support with Visual Studio 2017

http://blogs.msdn.microsoft.com/vcblog/2017/12/19/c17-progress-in-vs-2017-15-5-and-15-6/:

- ***We're gradually implementing experimental support for the parallel algorithms.***
- ***Here are the parallel algorithms that we've implemented:***
  - 15.5, intentionally not parallelized:
    `copy()`, `copy_n()`, `fill()`, `fill_n()`, `move()`, `reverse()`, `reverse_copy()`, `rotate()`, `rotate_copy()`, `swap_ranges()`
    - *The signatures for these parallel algorithms are added but not parallelized at this time; profiling showed no benefit in parallelizing algorithms that only move or permute elements*
  - 15.5: `all_of()`, `any_of()`, `for_each()`, `for_each_n()`, `none_of()`, `reduce()`, `replace()`, `replace_if()`, `sort()`
  - 15.6 Preview 1: `adjacent_find()`, `count()`, `count_if()`, `equal()`, `find()`, `find_end()`, `find_first_of()`, `find_if()`, `find_if_not()`
  - 15.6 after Preview 1: `mismatch()`, `remove()`, `remove_if()`, `search()`, `search_n()`, `transform()`
  - 15.x: `partition()`, `stable_sort()`

C++17: New STL Algorithms

see P0024R2

- **Supplementary algorithms support parallel processing**
  - **for_each_n**(*beg*, *count*, *op*)
    for_each_n(*pol*, *beg*, *count*, *op*)
    - Applies *op* to *count* elements of range starting from *beg* (using execution policy *pol*)
  - **reduce**(*beg*,*end*)
    reduce(*beg*,*end*, *initValue*)
    reduce(*beg*,*end*, *initValue*, *op*)
    - Like `accumulate()`, but applies *op* in an unspecified order
    - Yields a non-deterministic result for non-associative or non-commutative *op* such as floating-point addition
  - **transform_reduce**(...)
    - Accumulate transformed passed element (e.g. sum/product of ***op*** (***elem***))
  - **exclusive_scan**(...)
  - **inclusive_scan**(...)
  - **transform_exclusive_scan**(...)
  - **transform_inclusive_scan**(...)

Using `transform_reduce()`

```
#include <numeric>      // for transform_reduce()
#include <execution>    // for the execution policy
...
void printDirSize(const std::string& dir)
{
  // init paths with all paths from directory dir:
  std::vector<std::filesystem::path> paths;
  std::filesystem::recursive_directory_iterator dirpos(dir);
  std::copy(begin(dirpos), end(dirpos),
            std::back_inserter(paths));

  // sum size of regular files:
  auto sz = std::transform_reduce(
                    std::execution::par,                    // parallel execution policy (opt)
                    paths.cbegin(), paths.cend(),           // range
                    std::uintmax_t{0},                      // initial value (and return type)
                    std::plus<>(),                          // add ...
                    [](const std::filesystem::path& p) {    //  ... file size if regular file
                      return is_regular_file(p) ? file_size(p)
                                                : std::uintmax_t{0};
                    });
  std::cout << "sum of size of regular files: " << sz << '\n';
}
```

Can't pass directory iterator to parallel algorithms, because they require forward iterators

Call lambda for each element and accumulate results

C++17: clamp() and sample()

see P0025R0

- **clamp**(*val*, *min*, *max*)
  clamp(*val*, *min*, *max*, *op*)
  - yield value constraint to be between *min* and *max* (using op to compare values)
  - First form yields: std::min(std::max(*val*,*min*),*max*)

- **sample**(*beg*,*end*, *out*, *maxCount*, *gen*)
  - extract sample from [*beg*,*end*) to *out* according to random number generator *gen*
    - "selection sampling" and "reservoir sampling"

see N3925

C++17: Boyer Moore Substring Searchers

see P0220R1 P0253R1

- **default_search**(*beg*,*end*)
  default_search(*beg*,*end*, *pred*)
- **boyer_moore_searcher**(*beg*,*end*)
  boyer_moore_searcher(*beg*,*end*, *hash*)
  boyer_moore_searcher(*beg*,*end*, *hash*, *pred*)
- **boyer_moore_horspool_searcher**(*beg*,*end*)
  boyer_moore_horspool_searcher(*beg*,*end*, *hash*)
  boyer_moore_horspool_searcher(*beg*,*end*, *hash*, *pred*)
  - Function objects for faster substring search implementing the Boyer-Moore(-Horspool) algorithms ("needle in haystack")
    - see https://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm
    - see http://www.boost.org/doc/libs/1_58_0/libs/algorithm/doc/html/algorithm/Searching.html

- **std::search**(*beg*,*end*, *searcher*)
  - to switch between traditional and Bower-Moore search

C++17: Boyer Moore (Horspool) Searchers

```
#include <iostream>
#include <string>
#include <functional>   // for searchers

int main()
{
  std::string in = R"(Remember, an easy question can have an easy answer.
                      But a hard question must have a hard answer.
                      And for the hardest questions of all,
                      there may be no answer - except faith.)";
  std::string sub = "question";

Alternatives
  std::default_searcher searcher{sub.begin(), sub.end()};                  // classic
  std::boyer_moore_searcher searcher{sub.begin(), sub.end()};              // fastest
  std::boyer_moore_horspool_searcher searcher{sub.begin(), sub.end()};     // less space

  for (auto pos=in.begin(); pos!=in.end(); ) {
    auto [beg, end] = searcher(pos, in.end());
    std::cout << "found '" << sub << "' at offsets "
              << beg - in.begin() << "-" << end - in.begin() << "\n";
    pos = end;
  }
}
```

Output:

```
found 'question' at offsets 18-26
found 'question' at offsets 85-93
found 'question' at offsets 161-169
found 'question' at offsets 236-236
```


C++17: Support for Incomplete Types

- **Vectors and (forward) lists now support incomplete types**

see N4510

```
class Node
{
  private:
    std::string value;
    std::vector<Node> next;   // OK since C++17 (Node is an incomplete type here)
  public:
    Node(const std::string& s) : value{s}, next{} {
    }
    void add(const Node& n) {
      next.push_back(n);
    }
    Node& operator[](std::size_t idx) {
      return next.at(idx);
    }
    ...
};

Node root{"root"};
root.add(Node{"add1"});
root.add(Node{"add2"});
root[0].add(Node{"add3"});
...
```

C++17: Container Improvements

- **try_emplace()**
  - for map, unordered_map

see N4279

- **emplace_back(), emplace_front()**
  - now return a reference to the inserted element
  - for vector, deque, list, forward_list, stack, queue

see P0084R2

- **non-const data() for strings**

see P0272R1

- **Interface to associative/unordered containers to splice elements or modify keys without reallocation**

see P0083R3

C++17: Splicing (Unordered) Set and Map Elements

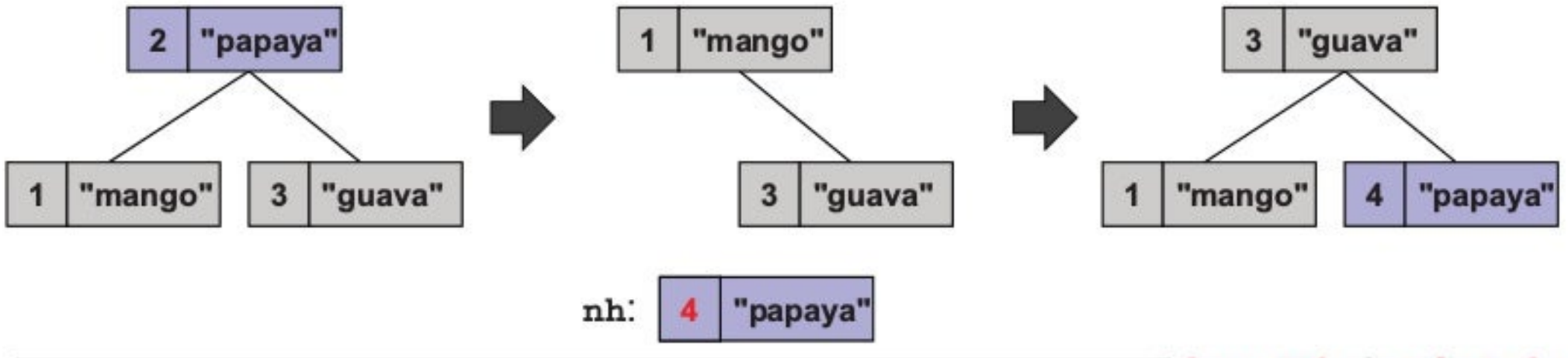
- **Interface and type to splice elements (nodes) between (unordered) sets and maps**
  - For:
    - Cheap modification of keys
    - Interface to support move-only element types
  - Better performance than `remove()` and `insert()`
  - References/pointers to elements remain valid
  - Only element types (and allocator) have to match
    - Comparison criterion, hash function, ... might differ
    - Can move from multi to non-multi container
- **Type for "node handle":** *container*`::node_type`
  - Member `value()` for all **set** types
  - Members `key()` and `mapped()` for all **map** types
  - Uses move semantics

C++17: Cheap Modification of (Unordered) Set/Map Keys

```
// Modifying the key of a set/map element:
std::map<int, std::string> m{{1,"mango"},
                             {2,"papaya"},
                             {3,"guava"}};

auto nh = m.extract(2);   // nh has type decltype(m)::node_type
nh.key() = 4;
m.insert(std::move(nh)); // Error without move()
```

Between:
- set and multiset,
- map and multimap
- unordered_set and unordered_multiset
- unordered_map and unordered_multimap

- References/pointers to elements remain valid
- But undefined behavior if used while node handle



C++17: Example for Splicing Set and Map Elements

```
// Moving elements from one hash table to another:
std::unordered_multimap<double, std::string> src{{1.1,"one"},
                                                 {2.2,"two"},
                                                 {3.3,"three"}};
std::unordered_map<double, std::string> dst{{3.3,"new"}};

dst.insert(src.extract(src.find(1.1)));   // splice using an iterator
dst.insert(src.extract(2.2));             // splice using the key

auto [pos,done,node] = dst.insert(src.extract(3.3)); // extracts but
if (!done) {                                         //  doesn't insert
  std::cout << "insert() of key '" << node.key()
            << "' with value '" << node.mapped()
            << "' failed due to existing elem with value '"
            << pos->second << "'\n";
}
```

insert(node_type&&)
returns **insert_return_type**
with members
- iterator **position**;
- bool **inserted**;
- node_type **node**;

C++17: New little helpers

- **std::as_const(*obj*)**
  - yields constant view to *obj* (not for temporaries (rvalues))
  - avoids use of const_cast and/or add_const<>
  - in <utility>

see P0007R1

- **std::size(*obj*), std::empty(*obj*), std::data(*obj*)**
  - Uniform API to check raw arrays, containers, initializer_list's
  - in <iterator>

see N4280

```
template <typename T>
void check (const T& t)
{
  std::cout << "size:  " << std::size(t) << '\n';
  std::cout << "empty: " << std::empty(t) << '\n';
  std::cout << "data:  " << std::data(t) << '\n';   // prints address
}

int arr[10] = { 42, 43, -1 };
check(arr);                    // empty() is always false for arrays
check(std::as_const(arr));     // yields same address with data()
std::vector<int> v;
check(v);
check(std::as_const(v));       // yields same address with data()
std::initializer_list<int> il = { 42, 43, -1 };
check(il);                     // data() is il.begin()
```


Mutex and Guard Classes

- **Mutex classes by C++11:**
  - Simple mutex: `std::mutex`
  - Nested mutexes: `std::recursive_mutex`
  - Simple mutex with timeout: `std::timed_mutex`
  - Nested mutex with timeout: `std::recursive_timed_mutex`
  - Read/Write mutex: `std::shared_mutex` (since C++17)
  - Read/Write mutex with timeout: `std::shared_timed_mutex` (since C++14)

- **Lock guards:**
  - Simple guard for 1 mutex: `std::lock_guard`
  - Simple shared read for 1 mutex: `std::shared_lock` (since C++14)
  - Simple guard for multiple mutexes: `std::scoped_lock` (since C++17)
  - Sophisticated guard for 1 mutex: `std::unique_lock`
    + lock(), try_lock_for(), try_lock_until(), unlock(), ...

C++14/C++17: Shared Locks

- **To support Read/Write Locks (RW locks)**
  - C++14 introduces
    - Class `std::shared_timed_mutex`
    - Class `std::shared_lock<>`
  - C++17 introduces
    - Class `std::shared_mutex`
- **Allows to share a lock for multiple concurrent reads**
  - No reader or writer preference defined (quality of implementation)
- **Generic approach:**
  - Use existing lock interfaces for exclusive locks
  - Use additional interfaces for shared locks
    - special state, available only in shared mutexes

see N4508

```
std::shared_mutex sm;   // a mutex to lock()/try_lock() and lock_shared()
...
{
   std::shared_lock<std::shared_mutex> sharedLock(sm);   // calls sm.lock_shared()
   ...
} // sm.unlock_shared() called here
```


C++17: std::scoped_lock<>

- std::**scoped_lock<>**
  - Variadic version of `std::lock_guard<>`
  - Thus, in C++17 we have:
    `lock_guard<>, unique_lock<>, scoped_lock<>`

```
void swap(const MyType& l, const MyType& r)
{
  std::lock(l.mtx, r.mtx);       // Note: deadlock avoidance algorithm used
  std::lock_guard<std::mutex> lg1(l.mtx, std::adopt_lock);
  std::lock_guard<std::mutex> lg2(r.mtx, std::adopt_lock);
  ...
}
```

```
void swap(const MyType& l, const MyType& r)
{
  std::scoped_lock lg(l.mtx, r.mtx);
  ...
}
```

uses class template argument deduction

C++17: Using Lock Guards

```
#include <shared_mutex>
#include <mutex>
...
std::vector<double> v;      // shared resource
std::shared_mutex vMutex;   // control access to v
```

```
void threadUpdate()
{
  while (true) {
    static double val = 1;
    { // update vector:
      std::scoped_lock sl(vMutex);
      v.push_back(val);
    }
    val *= 2.3;
    std::this_thread::sleep_for(1s);
  }
}
```

```
void threadRead()
{
  while (true) {
    double val = 0;
    // read from vector:
    if (std::shared_lock sl(vMutex);
        v.size()>0) {
      val = v.back();
    }
    ... // use val
  }
}
```


C++17: Using Lock Guards

a lot of shared reads of **done** block each other

```
#include <shared_mutex>
#include <mutex>
...
std::vector<double> v;      // shared resource
std::shared_mutex vMutex;   // control access to v
bool done = false;          // signal end
std::mutex doneMutex;
```

```
void threadUpdate()
{
  std::unique_lock lg(doneMutex);
  while (!done) {
    lg.unlock();
    static double val = 1;
    { // update vector:
      std::scoped_lock sl(vMutex);
      v.push_back(val);
    }
    val *= 2.3;
    std::this_thread::sleep_for(1s);
    lg.lock();
  }
} // finally release done lock
```

```
void threadRead()
{
  std::unique_lock<mutex> lg(doneMutex);
  while (!done) {
    lg.unlock();
    double val = 0;
    // read from vector:
    if (std::shared_lock sl(vMutex);
        v.size()>0) {
      val = v.back();
    }
    ... // use val
    lg.lock();
  }
} // finally release done lock
```

```
void threadSignalEnd()
{
  ...
  std::lock_guard lg(doneMutex);
  done = true;
} // release done lock
```

C++17: Using Lock Guards

better to use a condition variable here, of course

```
#include <shared_mutex>
#include <mutex>
...
std::vector<double> v;       // shared resource
std::shared_mutex vMutex;    // control access to v
bool done = false;           // signal end
std::shared_mutex doneMutex;
```

```
void threadUpdate()
{
  while (true) {
    if (std::shared_lock dl(doneMutex);
        done) {
      break;
    }
    static double val = 1;
    { // update vector:
      std::scoped_lock sl(vMutex);
      v.push_back(val);
    }
    val *= 2.3;
    std::this_thread::sleep_for(1s);
  }
}
```

```
void threadRead()
{
  while (true) {
    if (std::shared_lock dl(doneMutex);
        done) {
      break;
    }
    double val = 0;
    // read from vector:
    if (std::shared_lock sl(vMutex);
        v.size()>0) {
      val = v.back();
    }
    ... // use val
  }
}
```

```
void threadSignalEnd()
{
  ...
  std::lock_guard lg(doneMutex);
  done = true;
} // release done lock
```


atomic<> Operations

- **To check whether a type is natively atomic:**
  - Preprocessor constants (taken from C):

```
#define ATOMIC_BOOL_LOCK_FREE unspecified
#define ATOMIC_CHAR_LOCK_FREE unspecified
#define ATOMIC_INT_LOCK_FREE unspecified
#define ATOMIC_LONG_LOCK_FREE unspecified
#define ATOMIC_LLONG_LOCK_FREE unspecified
#define ATOMIC_POINTER_LOCK_FREE unspecified
...
```

    - might yield `0` for never, `1` for sometimes, `2` for always lock-free

C++17 will add:
`constexpr bool atomic<T>::is_always_lock_free`

  - Member function `is_lock_free()`
    - yields true if the **object** has native atomic support, so that it doesn't use locks
      - N2427: The proposal provides lock-free query functions on individual objects rather than whole types to permit unavoidably misaligned atomic variables without penalizing the performance of aligned atomic variables.
- **Note that `atomic<int>` differs from `int` even if it is lock free, because it introduces memory barriers.**

C++17: New Concurrency Informations

- **Helper to check whether atomic types use locks**

```
namespace std {
  template <typename T> struct atomic {
    static constexpr bool is_always_lock_free = implementation-defined;
    ...
  };
}
```

- **Portable way to access the L1 data cache line size**

```
namespace std {
  // minimum recommended offset between two concurrently-accessed objects:
  inline constexpr size_t
    hardware_destructive_interference_size = implementation-defined;
  // maximum recommended size of contiguous memory occupied by two objects
  inline constexpr size_t
    hardware_constructive_interference_size = implementation-defined;
}
```


C++17: New Mathematical Features

- **Special numerical functions:**
  - Elliptic integrals, Polynomials, Bessel functions, ...
  - IS 29124:2010
  - Only in <cmath> within namespace std (not in <math.h>)

see P0226R1

- **add `gcd()` and `lcm()`**
  - Templates for all integral types (also char) except `bool`

see P0295R0

- **3-argument `hypot()`**
  - For all floating-point types
  - Only in <cmath> within namespace std (not in <math.h>)

see P0030R1

C++17: Elementary String Conversions

- **Low-level C++ API for printf()-like formatting**
  - simple, fast, locale-independent
- **Including round-trip support**

see P0067R5 P0682R1

```
#include <charconv>

const char* str = "12 monkeys";
int value;
std::from_chars_result res = std::from_chars(str, str+10,
                                             value);

if (auto[ptr, ec] = std::from_chars(str, str+10, value); ec != std::errc{}) {
  ...      // error handling
}

int value = 42;
char str[10];
std::to_chars_result res = std::to_chars(str, str+10,
                                         value);

if (auto[ptr, ec] = std::to_chars(str, str+10, value); ec != std::errc{}) {
  ...      // error handling
}
```


C++17: Elementary String Conversions

- **Low-level C++ API for printf()-like formatting**
  - simple, fast, locale-independent
- **Including round-trip support**

see P0067R5

```
to_chars(beg,end, 44)                          // decimal
to_chars(beg,end, 44, 16)                      // hexadecimal
to_chars(beg,end, 44, 13)                      // to the base of 13 (base between [2,36])

to_chars(beg,end, 44.1234567)                  // decimal, roundtrip-able, %f or %e acc. to minimal # chars

to_chars(beg,end, 44.1234567, chars_format::general)        // decimal, %g, roundtrip-able
to_chars(beg,end, 44.1234567, chars_format::fixed)          // decimal, %f, roundtrip-able
to_chars(beg,end, 44.1234567, chars_format::scientific)     // decimal, %e, roundtrip-able
to_chars(beg,end, 44.1234567, chars_format::hex)            // hex, %a, roundtrip-able

to_chars(beg,end, 44.1234567, chars_format::general, 4)     // decimal, %g, precision: 4
to_chars(beg,end, 44.1234567, chars_format::fixed, 4)       // decimal, %f, precision: 4
to_chars(beg,end, 44.1234567, chars_format::scientific, 4)  // decimal, %e, precision: 4
to_chars(beg,end, 44.1234567, chars_format::hex, 4)         // hex, %a, precision: 4
```

Shared Pointers and Deleters

- By default `delete` (for single objects) is called
- You can (and might have to) pass other deleters as second argument to the constructor

```
std::shared_ptr<FILE> pp(popen(...),                    // pp represents open pipe
                         [] (FILE* fp) {
                           std::cout << "close pipe" << '\n';
                           pclose(fp);
                         });

std::shared_ptr<std::string> p(new std::string[10]);    // ERROR, but compiles

std::shared_ptr<std::string> p(new std::string[10],     // OK, own deleter for arrays
                               [](std::string* p) {
                                   delete[] p;
                               });

std::shared_ptr<std::string> p(new std::string[10],     // OK, default deleter for arrays
                               std::default_delete<std::string[]>());

std::shared_ptr<std::string[]> p(new std::string[10]);  // OK, since C++17
```


C++17: (Bug) Fixes

- Guarantee that shared_from_this() throws `bad_weak_ptr` if no shared pointer was created for an object yet
- Adding `constexpr` in a lot of places
  - For example:
    - char_traits
    - chrono is complete constexpr except `now()`, `to_time_t()`, and `from_time_t()`
- Polishing <chrono>:
  - Add floor(), ceil(), abs(), round() where missing

see P0505R0

see P0092R2

C++17

Expert Utilities

New Type Traits since C++17

- **is_aggregate**
  - to find out whether an object is an aggregate type
- **has_unique_object_representations**
  - to find out whether equal objects have equal hash values
- **is_swappable, is_swappable_with, is_nothrow_swappable, is_nothrow_swappable_with**
- **is_invocable, is_invocable_r, is_nothrow_invocable, is_nothrow_invocable_r, invoke_result**
  - for objects that can be used as functions
  - Temporarily we had: is_callable
- **conjunction, disjunction, negation**
  - Combine type traits for variadic templates

Deprecated: is_literal_type, result_of

C++17: Type Trait std::is_aggregate<>

see LWG Issue 2911

- **To enable special handling if (not) aggregate**

```
#include <type_traits>

template <typename Coll, typename... T>
void insert(Coll& coll, T&&... val)
{
  if constexpr(!std::is_aggregate_v<typename Coll::value_type>) {
    coll.emplace_back(std::forward<T>(val)...);   // invalid for aggregates
  }
  else {
    coll.emplace_back(typename Coll::value_type{std::forward<T>(val)...});
  }
}
```

```
class C {
    int i;
    double d;
  public:
    C(int x, double y) : i(x), d(y) {
    }
};

std::vector<C> vc;
vc.emplace_back(42, 7.5);   // OK
insert(vc, 42, 7.5);        // OK
```

```
struct A {
  int i;
  double d;
};



std::vector<A> va;
va.emplace_back(42, 7.5);   // Error
insert(va, 42, 7.5);        // OK
```


Combining Type Traits since C++17

see P0013R1

`std::conjunction<>`, `std::disjunction<>`, `std::negation<>`

- **allow to combine type trait definitions similar to `&&`, `||`, `!`**
- **`conjunction` and `disjunction` short circuit**
  - end evaluation as soon as outcome is clear

```
template<typename T>
struct IsNoPtrT
 : std::negation<std::disjunction<std::is_null_pointer<T>,
                                  std::is_member_pointer<T>,
                                  std::is_pointer<T>>> {
};

std::cout << isNoPtrT<void*>::value << '\n';          // false
std::cout << isNoPtrT<std::string>::value << '\n';    // true
auto np = nullptr;
std::cout << isNoPtrT<decltype(np)>::value << '\n';   // false
```

Combining Type Traits since C++17

- `std::conjunction<>` and `std::disjunction<>`
  - short circuit

see P0013R1

```
struct X {
  X(int);
};
struct Y;     // incomplete type

// undefined behavior
// because Y is incomplete is_constructible<> might not compile:
static_assert(std::is_constructible<X,int>{}
                || std::is_constructible<Y,int>{},
              "can't init X or Y from int");

// OK, will compile and has defined behavior:
static_assert(std::disjunction<std::is_constructible<X, int>,
                               std::is_constructible<Y, int>>{},
              "can't init X or Y from int");
```

Thanks to Howard Hinnant for pointing that out

C++17: std::invoke<>()

- **Utility for generic code to call callables:**
  - functions, function pointers
  - function objects, lambdas
  - member function for objects passed as additional argument

see N4169

```
template <typename Iter, typename Callable, typename... Args>
void foreach (Iter act, Iter end, Callable&& op, Args&&... args)
{
  for ( ; act != end; ++act) {   // as long as not reached the end call:
    std::invoke(std::forward<Callable>(op),    // callable
                std::forward<Args>(args)...,   // additional args
                *act);                         // current element
  }
}

std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };

foreach(primes.begin(), primes.end(),                  // elements used as 2nd arg
        [] (std::string const& prefix, int i) {        // lambda to call
          std::cout << prefix << i << '\n';
        },
        "- value: ");                                  // 1st arg for lambda
```

C++17: std::invoke<>() for Member Functions

```
#include <functional>
#include <iostream>
#include <vector>

template <typename Iter, typename Callable, typename... Args>
void foreach (Iter act, Iter end, Callable&& op, Args&&... args)
{
  for ( ; act != end; ++act) {   // as long as not reached the end, call:
    std::invoke(std::forward<Callable>(op),     // passed callable for
                std::forward<Args>(args)...,    // any additional arguments
                *act);                          // and the current element
  }
}

class MyClass {
  public:
    void memfunc(int i) const {
      std::cout << "MyClass::memfunc() called for: " << i << '\n';
    }
};

int main()
{
  std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };

  MyClass obj;
  foreach(primes.begin(), primes.end(),   // elements used as args
          &MyClass::memfunc,              // member function to call
          obj);                           // object to call memfunc() for
}
```


C++17: std::invoke<>() with Perfect Forwarding Return Values

```
#include <utility>       // for std::invoke()
#include <functional>    // for std::forward()
#include <type_traits>   // for std::is_same<> and std::invoke_result<>

template<typename Callable, typename... Args>
decltype(auto) call(Callable&& op, Args&&... args)
{
  if constexpr(std::is_same_v<std::invoke_result_t<Callable, Args...>,
                              void>) {
    // return type is void:
    std::invoke(std::forward<Callable>(op),
                std::forward<Args>(args)...);
    …
    return;
  }
  else {
    // return type is not void:
    decltype(auto) ret{std::invoke(std::forward<Callable>(op),
                                   std::forward<Args>(args)...)};
    …
    return ret;
  }
}
```

invalid for `void` (because it is an incomplete type)

C++17: is_invocable<> and invoke_result<> Type Traits

- **find out, whether a callable can be called for an object and which type it returns**
  - **is_invocable**<*T*, *Args*...>
  - **is_invocable_r**<*RET_T*, *T*, *Args*...>
    - yields whether you can use *T* as callable for *Args*... (returning a value convertible to type *RET_T*)
  - **is_nothrow_invocable**<*T*, *Args*...>
  - **is_nothrow_invocable_r**<*RET_T*, *T*, *Args*...>
    - same with the guarantee that no exception is thrown
  - **invoke_result**<*T*, *Args*...>
    - yields the return type when using *T* as callable for *Args*...
    - type is not defined if no call is possible
- **Replace: result_of<>** (since C++11) and **is_callable<>** (first proposed for C++17)
  - Better names, modern declaration, they decay now before any check, argument types can be abstract class, and function or array types can be used as "return type"

C++17: Using invoke_result<> and is_invocable<>

```
struct C {
  bool operator() (int) const {
    return true;
  }
};

struct Abstract {
   virtual ~Abstract() = 0;
   void operator() () const {
   }
};

typename std::result_of<C(int)>::type          // yields bool
typename std::invoke_result<C,int>::type       // yields bool

typename std::result_of<Abstract()>::type      // ERROR: Abstract() is ill-formed
typename std::invoke_result<Abstract>::type    // OK, yields void

std::is_invocable<C,int>::value                // true
std::is_invocable<Abstract>::value             // true
std::is_invocable<int*>::value                 // false
std::is_invocable<int(*)()>::value             // true

std::is_invocable_r<bool,C,int>::value         // true
std::is_invocable_r<int,C,long>::value         // true
std::is_invocable_r<void,C,int>::value         // true
std::is_invocable_r<char*,C,int>::value        // false
```

C++17: Other Improvements for Templates

- **Defining std::void_t**
  - map any given sequence of types to a single type, `void`
  - Simply:

```
template<typename... > using void_t = void;
```

see N3911

```
#include <utility>       // for declval<>
#include <type_traits>   // for true_type, false_type, and void_t

// primary template:
template<typename, typename = std::void_t<>>
struct HasVariousT : std::false_type {
};

// partial specialization (may be SFINAE'd away):
template<typename T>
struct HasVariousT<T, std::void_t<decltype(std::declval<T>().begin()),
                                  typename T::difference_type,
                                  typename T::iterator>>
 : std::true_type {
};
```


C++17: Other Improvements for Templates

- **Special std::bool_constant<> type for type_traits that yield `true` or `false`**
  - does better support implicit conversions

see N4389

```
namespace std {
  template<bool B>
    using bool_constant = integral_constant<bool, B>;
  using true_type = bool_constant<true>;
  using false_type = bool_constant<false>;
}


template<typename T>
struct IsLargerThanInt
 : std::bool_constant<(sizeof(T)>sizeof(int))> {
}

if constexpr(IsLargerThanInt<T>::value) {
    ...
}
```

C++17: `std::uncaught_exceptions()`

see N4259

`int std::uncaught_exceptions()`

- **Correct interface to detect relevant stack unwinding**
- **Replaces:** `bool std::uncaught_exception()`
  - Did not work if object was temporarily created during stack unwinding

```
class C {
  private:
    int initialUncaught = std::uncaught_exceptions();
  public:
    …
    ~C() {
      if (initialUncaught == std::uncaught_exceptions()) {
        commit();     // destruction not caused by stack unwinding
      }
      else {
        rollback();   // destruction caused by stack unwinding
      }
    }
};
```


C++17: Using `std::uncaught_exceptions()`

```
class C {
 private:
  int initialUncaught = std::uncaught_exceptions();
 public:
  …
  ~C() {
    if (initialUncaught == std::uncaught_exceptions()) {
      commit();     // destruction not caused by stack unwinding
    }
    else {
      rollback();   // destruction caused by stack unwinding
    }
  }
};
```

```
C x("x");
D d("d");
throw "oops";
```

```
C::C() for x: initialUncaught: 0
D::D() for d
-------------------- throw
D::~D() for d
  C::C() for y: initialUncaught: 1
    C::C() for z: initialUncaught: 1
    -------------------- throw
    C::~C() for z: 1 => 2 uncaught => rollback()
  C::~C() for y: 1 => 1 uncaught => commit()
C::~C() for x: 0 => 1 uncaught => rollback()
```

```
class D {
 public:
   …
   ~D() {
      cleanup();
   }
};


void cleanup()
{
  C y("y");
  try {
    C z("z");
    throw "oops";
  }
  catch (...) {
  }
}
```

C++17: Fix noexcept Declarations of some Containers

- **In Rapperswil 2014/06 we discussed this topic based on paper N4002 in LEWG and came to the following concluding vote:**
  - For **vectors** we want to have
    - noexcept default constructor
    - noexcept move constructor
    - conditional noexcept move assignment
    - conditional noexcept swap
  - For **strings** we want to have
    - noexcept default constructor
    - noexcept move constructor
    - conditional noexcept move assignment
    - conditional noexcept swap
  - For **all other containers** we want to have
    - conditional noexcept move assignment
    - conditional noexcept swap
    - **no** required noexcept for move constructor

Best performance guaranteed in all modes for our "almost fundamental data types"
- **vector**
- **string**

C++17: Final Changes in Library Working Group

In Urbana 2014/11 we agreed in LWG on N4258 (extract of 18 pages):

- In §**21.4 [basic.string]** in class std::basic_string

Modify (add):

```
basic_string() noexcept : basic_string(Allocator()) { }

explicit basic_string(const Allocator& a) noexcept;
```

Unlike library issue 2319 proposed,

keep:

```
basic_string(basic_string&& str) noexcept;
```

Modify (add):

```
basic_string& operator=(basic_string&& str)
noexcept(allocator_traits<Allocator>
           ::propagate_on_container_move_assignment::value
         || allocator_traits<Allocator>::is_always_equal::value);
```

similar changes for swap

Modern C++

std::launder()
and
Why push_back() is Broken

The Problem

3.8 Object lifetime [basic.life]

If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

(8.1) — the storage for the new object exactly overlays the storage location which the original object occupied, and

(8.2) — the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and

(8.3) — the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and

(8.4) — the original object was a most derived object (1.8) of type T and the new object is a most derived object of type T (that is, they are not base class subobjects).

The Problem

3.8 Object lifetime [basic.life]

If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime begins, can be used to manipulate the new object, if

In other words:

The name of an object
that dies and gets replaced by a new object,
can be reused for the new object unless ...
- the object is const or does
  contain any non-static const/ref members

qualified version of its type, and

(8.4) — the original object was a most derived object (1.8) of type T and the new object is a most derived object of type T (that is, they are not base class subobjects).

The Consequence

```
#include <new>

struct C {
  int i;
  const int c;
};

C* p = new C{0,0};
int a = p->i;
new(p) C{42,42};
```

// OK: a is 0
// request to place new C initialized by 42,42 to *p
// compilers may assume that value did not change

The Consequence

```
#include <new>

struct C {
  int i;
  const int c;
};

C* p = new C{0,0};
int a = p->i;
new(p) C{42,42};

int b = p->c;
int c = p->i;
```

// OK: a is 0
// request to place new C initialized by 42,42 to *p
// compilers may assume that value did not change
// **undefined behavior** (might work)
// **undefined behavior** (might work)

The Consequence

```
#include <new>

struct C {
  int i;
  const int c;
};

C* p = new C{0,0};
int a = p->i;
new(p) C{42,42};

int b = p->c;
int c = p->i;
```

Would not be a problem with:

```
p = new(p) C{42,42};
```

// OK: a is 0
// request to place new C initialized by 42,42 to *p
// compilers may assume that value did not change
// **undefined behavior** (might work)
// **undefined behavior** (might work)

C+17: std::launder()

```
#include <new>

struct C {
  int i;
  const int c;
};

C* p = new C{0,0};
int a = p->i;
new(p) C{42,42};

int b = p->c;
int c = p->i;

int d = std::launder(p)->c;
int e = std::launder(p)->i;
int f = p->c;
int g = p->i;

auto q = std::launder(p);
int h = q->c;
```

Would not be a problem with:
`p = new(p) C{42,42};`

```
// OK: a is 0
// request to place new C initialized by 42,42 to *p
// compilers may assume that value did not change
// undefined behavior (might work)
// undefined behavior (might work)

// OK, d is 42 (launder(p) forces to double-check)
// OK, e is 42 (launder(p) forces to double-check)
// still undefined behavior
// still undefined behavior

// OK, h is 42
```


push_back() with Elements with Constant Members

```
class C {
 private:
  int i;
  const int c;            // constant member !
 public:
  C(int x, int y) : i(x), c(y) {
  }
  friend std::ostream& operator<< (std::ostream& os, const C& c) {
    return os << c.i << " " << c.c;
  }
  ...
};

std::vector<C> v;
v.push_back(C{0,0});      // insert first element
v.clear();
v.push_back(C{42,0});     // insert a new first element
std::cout << v[0];        // should output '42 0'
```

Implementing a Container Using an Allocator

```
template <typename T, typename A = std::allocator<T>>
class vector
{
  public:
    using ATR = typedef typename std::allocator_traits<A>;
    using pointer = typename ATR::pointer;
  private:
    A _alloc;          // current allocator
    pointer _elems;    // array of elements
    size_t _size;      // number of elements
    size_t _capa;      // capacity
  public:
    …
    void push_back(const T& t) {
      if (_capa == _size) {     // if necessary reserve more memory
        reserve((_capa+1)*2);
      }
      ATR::construct(_alloc, _elems+_size, t);   // construct the new element
      ++_size;
    }
    T& operator[] (size_t i) {
      return _elems[i];
    }
    …
};
```

Missing:
- move semantics
- exception handling

What (default) Allocator-Traits do

```
namespace std {
 template <class Alloc> struct allocator_traits
 {
   …
   static pointer allocate(Alloc& a, size_type n);
   static pointer allocate(Alloc& a, size_type n,
                           const_void_pointer hint);
   static void deallocate(Alloc& a, pointer p, size_type n);

   template <class T, class... Args>
   static void construct(Alloc& a, T* p, Args&&... args) {
     // directly or via a.construct() calls:
     ::new(static_cast<void*>(p)) T(std::forward<Args>(args)...);
   }

   template <class T>
   static void destroy(Alloc& a, T* p);
 };
```

Implementing a Container Using an Allocator

```
template <typename T, typename A = std::allocator<T>>
class vector
{
  public:
    using ATR = typedef typename std::allocator_traits<A>;
    using pointer = typename ATR::pointer;
  private:
    A _alloc;          // current allocator
    pointer _elems;    // array of elements
    size_t _size;      // number of elements
    size_t _capa;      // capacity
  public:
    …
    void push_back(const T& t) {
      if (_capa == _size) {
        reserve((_capa+1)*2);
      }
      ATR::construct(_alloc, _elems+_size, t);
      ++_size;
    }
    T& operator[] (size_t i) {
      return _elems[i];   // UB for replaced elements with constant members
    }
    …
};
```


std::launder() does not help here

```
template <typename T, typename A = std::allocator<T>>
class vector
{
  public:
    using ATR = typedef typename std::allocator_traits<A>;
    using pointer = typename ATR::pointer;
  private:
    A _alloc;          // current allocator
    pointer _elems;    // array of elements
    size_t _size;      // number of elements
    size_t _capa;      // capacity
  public:
    …
    void push_back(const T& t) {
      if (_capa == _size) {
        reserve((_capa+1)*2);
      }
      ATR::construct(_alloc, _elems+_size, t);
      ++_size;
    }
    T& operator[] (size_t i) {
      return std::launder(_elems)[i];   // still UB for replaced elems with const members
    }
    …
};
```

std::launder(p) requires p to be a pointer.
ATR::pointer might not be a raw pointer.

std::launder() does not help here

```
template <typename T, typename A = std::allocator<T>>
class vector
{
  public:
    typedef typename std::allocator_traits<A> ATR;
    typedef typename ATR::pointer pointer;
  private:
    A _alloc;          // current allocator
    pointer _elems;    // array of elements
    size_t _size;      // number of elements
    size_t _capa;      // capacity
public:
    …
    void push_back(const T& t) {
      if (_capa == _size) {
        reserve((_capa+1)*2);
      }
      ATR::construct(_alloc, _elems+_size, t);
      ++_size;
    }
    T& operator[] (size_t i) {
      return std::launder(this)->_elems[i];   // still UB for repl. elems with const mem.
    }
    …
};
```

std::launder(p) is a no-op unless p points to an object whose lifetime has ended and where a new object has been created in the same storage.

push_back() results into Undefined Behavior

see P0532R0 for details

```
class C {
 private:
  int i;
  const int c;          // constant member !
 public:
  C(int x, int y) : i(x), c(y) {
  }
  friend std::ostream& operator<< (std::ostream& os, const C& c) {
    return os << c.i << " " << c.c;
  }
  …
};

std::vector<C> v;
v.push_back(C{0,0});    // insert first element
v.clear();
v.push_back(C{42,0});   // undefined behavior (due to const member)
std::cout << v[0];      // might output '0 0' or '42 0'
```

Also a Problem for std::optional<> and std::variant<>

```
template <typename T>
class coreoptional
{
  private:
    T payload;
  public:
    coreoptional(const T& t)
     : payload(t) {
    }
    template<typename... Args>
    void emplace(Args&&... args) {
      payload.~T();
      ::new(&payload) T(std::forward<Args>(args)...);
    }
    const T& operator*() const & {
      return payload;
    }
};
```

see **P0532R0** for details

std::launder() Summary

- **for std::vector<> and std::deque<>**
  - push_back(), emplace_back(), insert(), ...
    is broken if elements have const members
  - Note: Problem since C++11
    - Needs move semantics to move them in

breaks binary compatibility

- **std::optional<> and std::variant<>**
  - need additional pointer members to refer to their own member to work with values having const members

C++17

Other Aspects of C++ 17

C++17 Based on C11

see P0063R3

- **C++17 refers to C11 instead of C99**
  - C11 Draft:
    - www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
  - Changes:
    - https://en.wikipedia.org/wiki/C11_(C_standard_revision)

  - C functions for "safer, more secure programming" with suffix `_s` and additional size argument (e.g. `strcpy_s()`, `sprintf_s()`)
    - *may* be available in the C++ headers
    - but not defined in namespace `std`

C++17: Deprecated / Removed in Core

- **Removed:**
  - Trigraphs
  - Operator `++` for `bool`
  - keyword `register` (but keep reserved for future use)
  - Dynamic exception specifications with Types:

```
void foo() throw(bad_alloc);  // invalid since C++17
void foo() throw();           // still valid, but might not unwind stack
```

  - `auto_ptr<>`, `unary_function`, `binary_function`, `ptr_fun`, `mem_fun`, `mem_fun_ref`, `bind1st`, `bind2nd`, `random_shuffle()`
  - `std::function<>` no longer has allocator support
- **Deprecated in Library:**
  - `<codecvt>`, `wstring_convert`, `wbuffer_convert` (was new in C++11)
  - `result_of<>` trait (use `invoke_result<>` instead)
- **Temporarily discouraged:**
  - `memory_order_consume`

Modern C++

C++ 17 Conclusion

C++17 Highlights

- **Structured Bindings**
- **Fold expressions**
- **Inline Variables**
- **Compile-Time if**
- **Class Template Argument Deduction**
- **std::optional**
- **std::variant**
- **std::any**
- **std::byte**
- **std::string_view** (but considered harmful)
- **Filesystem Library**
- **Parallel STL Algorithms**
- **Several Small Helpers**
- **more and more constexpr all over the place**

Uniform Initialization now Almost Done

```
int i2 = 0;      // "copy initialization" (from C)
int i3(0);       // "direct initialization" (since C++98)
int i4{0};       // "direct initialization" from Modern C++ (since C++11)
int i5 = {0};    // "copy initialization" from Modern C++

unsigned long l1{-17};      // ERROR (good!): "narrowing" detected
unsigned long l2{4.8};      // ERROR (good!): "narrowing" detected

auto x{42};                 // x is int now (was std::initializer_list<int> before C++17)
auto y{0,8,15};             // ERROR now (good!, was std::initializer_list<int> before C++17)
auto z = {0,8,15};          // still std::initializer_list<int>

enum class MyIntegralEnum;
MyIntegralEnum e{17};   // OK since C++17 (all other initializations are still ill-formed)

struct CData {
  int amount;
  double value;
};
struct DData : CData {
  std::string name;
  void print() const;
};
DData a{{42, 6.7}, "book"};   // initialize all elements

AnyType t{};                // value initialization (default value or 0/false/nullptr for FDT's)
int i6{};                   // initialized with 0 (since C++11)
DData b{};                  // same as: {0,0.0},""}
MyIntegralEnum f{};         // still OK
```

- Use uniform direct initialization
  - uniform: **with** {...}
  - direct: **without** =
- Still `std::atomic<>` needs a fix
  - see **P0883R0**
- Still some flaws with **aggregates**
  - `AggrT x{};` is possible with deleted constructor

What was *not* voted into C++17

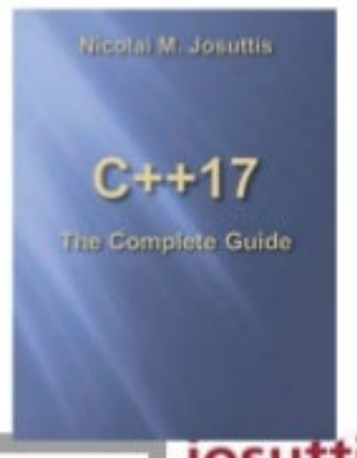
- **Modules**
- **Concepts**
- **Ranges**
- **Reflection**
- **Co-Routines**

C++20 will be the next revolution

- **Overloading `operator.()`**
- **Default comparison operators**
- **Uniform Call Syntax**
- **Networking library**
  - Boost.Asio

C++17 Overviews (Extract)

- **Changes between C++14 and C++17 DIS**
  - https://isocpp.org/files/papers/p0636r0.html
  - List of all papers with all features
  - by Thomas Köppe
- **Descriptions of C++17 features**
  - presented mostly in "Tony Tables"
  - https://github.com/tvaneerd/cpp17_in_TTs/
  - by Tony Van Eerd
- **C++17 Overview Slides**
  - http://www.codeplay.com/public/uploaded/public/0cbdaf_c++17post-oulu2016.pdf
  - by Michael Wong
- **Draft Available: C++17 – The Complete Guide**
  - see http://cppstd17.com/
  - by Nicolai M. Josuttis



C++17: Summary

- **C++17 is a significant improvement**
  - Big step out of many small steps
- **Compiler support**
  - gcc/g++ 7
  - Clang 4
  - VS2017.x

nico@josuttis.de

- **Some books already cover C++17:**
  - ***C++17 – The Complete Guide***
    - see **http://cppstd17.com**
  - ***C++ Templates, 2nd ed.***
    - see **http://tmplbook.com**

- **We are still looking forward to C++20**