train IT

C++11 was only the beginning

DO NOT STAY BEHIND

Mateusz Pusz

June 17, 2019

Who am I?

- **Modern C++ Evangelist**
- Hacking C++ for more than *15 years* for fun and living

Who am I?

- **Modern C++ Evangelist**
- Hacking C++ for more than *15 years* for fun and living
- *Systems*, *Software*, and *Solution Architect*, *Chief Software Engineer*, *Security Champion*, *Validation Lead*
- *Trainer*, *coach*, *mentor*, *consultant*, and *conference speaker*

Who am I?

- **Modern C++ Evangelist**
- Hacking C++ for more than *15 years* for fun and living
- *Systems*, *Software*, and *Solution Architect*, *Chief Software Engineer*, *Security Champion*, *Validation Lead*
- *Trainer*, *coach*, *mentor*, *consultant*, and *conference speaker*
- Active voting member and *contributor* of the **ISO C++ Committee** (WG21), WG21 **Study Group 14** (SG14), and **Study Group 20** (SG20)

Who am I?

- **Modern C++ Evangelist**
- Hacking C++ for more than *15 years* for fun and living
- *Systems*, *Software*, and *Solution Architect*, *Chief Software Engineer*, *Security Champion*, *Validation Lead*
- *Trainer*, *coach*, *mentor*, *consultant*, and *conference speaker*
- Active voting member and *contributor* of the **ISO C++ Committee** (WG21), WG21 **Study Group 14** (SG14), and **Study Group 20** (SG20)
- Founder of **Train IT** and several Open Source projects and libraries
- Worked at **Intel** for 13 years and now the *head of the C++ Competency Center* at **EPAM Systems**

Who am I?

- **Modern C++ Evangelist**
- Hacking C++ for more than *15 years* for fun and living
- *Systems*, *Software*, and *Solution Architect*, *Chief Software Engineer*, *Security Champion*, *Validation Lead*
- *Trainer*, *coach*, *mentor*, *consultant*, and *conference speaker*
- Active voting member and *contributor* of the **ISO C++ Committee** (WG21), WG21 **Study Group 14** (SG14), and **Study Group 20** (SG20)
- Founder of **Train IT** and several Open Source projects and libraries
- Worked at **Intel** for 13 years and now the *head of the C++ Competency Center* at **EPAM Systems**
- Mainly interested in *code performance*, *low latency*, *stability*, and *security*

Who am I?

- **Modern C++ Evangelist**
- Hacking C++ for more than *15 years* for fun and living
- *Systems*, *Software*, and *Solution Architect*, *Chief Software Engineer*, *Security Champion*, *Validation Lead*
- *Trainer*, *coach*, *mentor*, *consultant*, and *conference speaker*
- Active voting member and *contributor* of the **ISO C++ Committee** (WG21), WG21 **Study Group 14** (SG14), and **Study Group 20** (SG20)
- Founder of **Train IT** and several Open Source projects and libraries
- Worked at **Intel** for 13 years and now the *head of the C++ Competency Center* at **EPAM Systems**
- Mainly interested in *code performance*, *low latency*, *stability*, and *security*
- *The winner of Bench Games 2013* – worldwide competition in the C++ language knowledge

What is C++?

What is C++?

- *C++ is no longer C with classes*

What is C++?

- *C++ is no longer C with classes*
- C++ is a **general-purpose programming language**
- It has *imperative*, *object-oriented*, and *generic programming* features, while also providing facilities for *low-level memory manipulation*

What is C++?

- *C++ is no longer C with classes*
- C++ is a **general-purpose programming language**
- It has *imperative*, *object-oriented*, and *generic programming* features, while also providing facilities for *low-level memory manipulation*
- If used correctly, provides hard to beat **performance**

What is C++?

- *C++ is no longer C with classes*
- C++ is a **general-purpose programming language**
- It has *imperative*, *object-oriented*, and *generic programming* features, while also providing facilities for *low-level memory manipulation*
- If used correctly, provides hard to beat **performance**

C++ motto:
You don't pay for what you don't use

Modern C++

- Philosophy of code design

Modern C++

- **Philosophy of code design**
  - Extensive and wise usage of a *subset of C++ language features and C++ standard library*
  - *Implementing the code with performance in mind*, achieved by the awareness of interactions between the software and the hardware
  - Following the *best practices, coding guidelines, and idioms*
  - Following the *latest versions of the C++ standard* to make the code development more efficient for engineers, and the resulting products safer, and even faster

Modern C++

- **Philosophy of code design**
  - Extensive and wise usage of a *subset of C++ language features and C++ standard library*
  - *Implementing the code with performance in mind*, achieved by the awareness of interactions between the software and the hardware
  - Following the *best practices, coding guidelines, and idioms*
  - Following the *latest versions of the C++ standard* to make the code development more efficient for engineers, and the resulting products safer, and even faster

> Within C++, there is a much smaller and cleaner language struggling to get out.
>
> *-- Bjarne Stroustrup'2007*

C++11 (ISO/IEC 14882:2011)

C++11 (ISO/IEC 14882:2011)

CORE LANGUAGE RUNTIME PERFORMANCE

- Move Semantics
- Constant Expressions

C++11 (ISO/IEC 14882:2011)

CORE LANGUAGE RUNTIME PERFORMANCE

- Move Semantics
- Constant Expressions

CORE LANGUAGE USABILITY

- Lambdas
- Object construction improvements
- Explicit and final overrides
- Range-based-for-loop
- Scoped enumeration

C++11 (ISO/IEC 14882:2011)

CORE LANGUAGE RUNTIME PERFORMANCE

- Move Semantics
- Constant Expressions

CORE LANGUAGE USABILITY

- Lambdas
- Object construction improvements
- Explicit and final overrides
- Range-based-for-loop
- Scoped enumeration

CORE LANGUAGE FUNCTIONALITY

- Variadic templates
- Multitasking memory model
- Explicit defaulted and deleted functions
- Static assertions

C++11 (ISO/IEC 14882:2011)

CORE LANGUAGE RUNTIME PERFORMANCE

- Move Semantics
- Constant Expressions

CORE LANGUAGE USABILITY

- Lambdas
- Object construction improvements
- Explicit and final overrides
- Range-based-for-loop
- Scoped enumeration

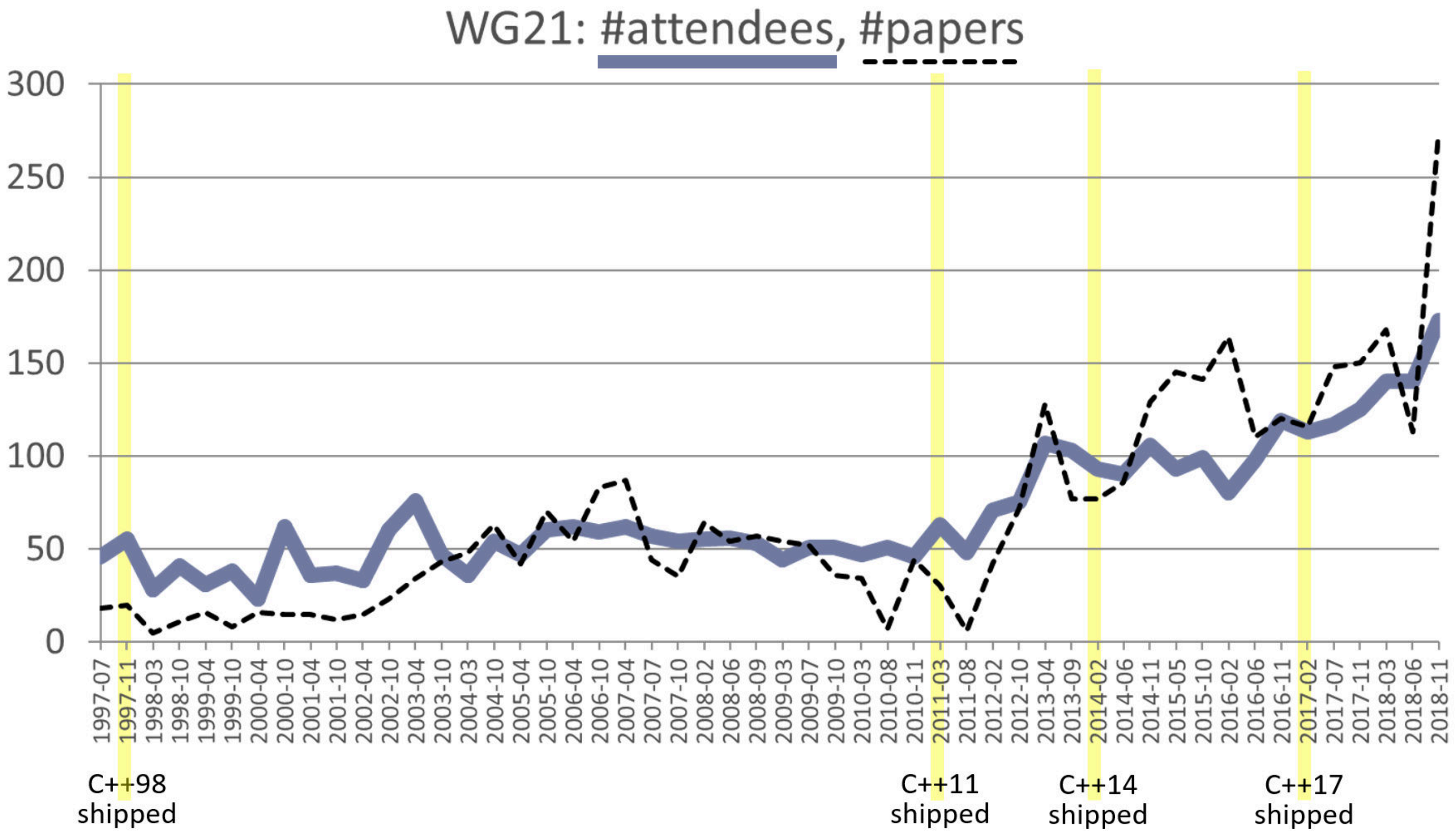
CORE LANGUAGE FUNCTIONALITY

- Variadic templates
- Multitasking memory model
- Explicit defaulted and deleted functions
- Static assertions

C++ STANDARD LIBRARY

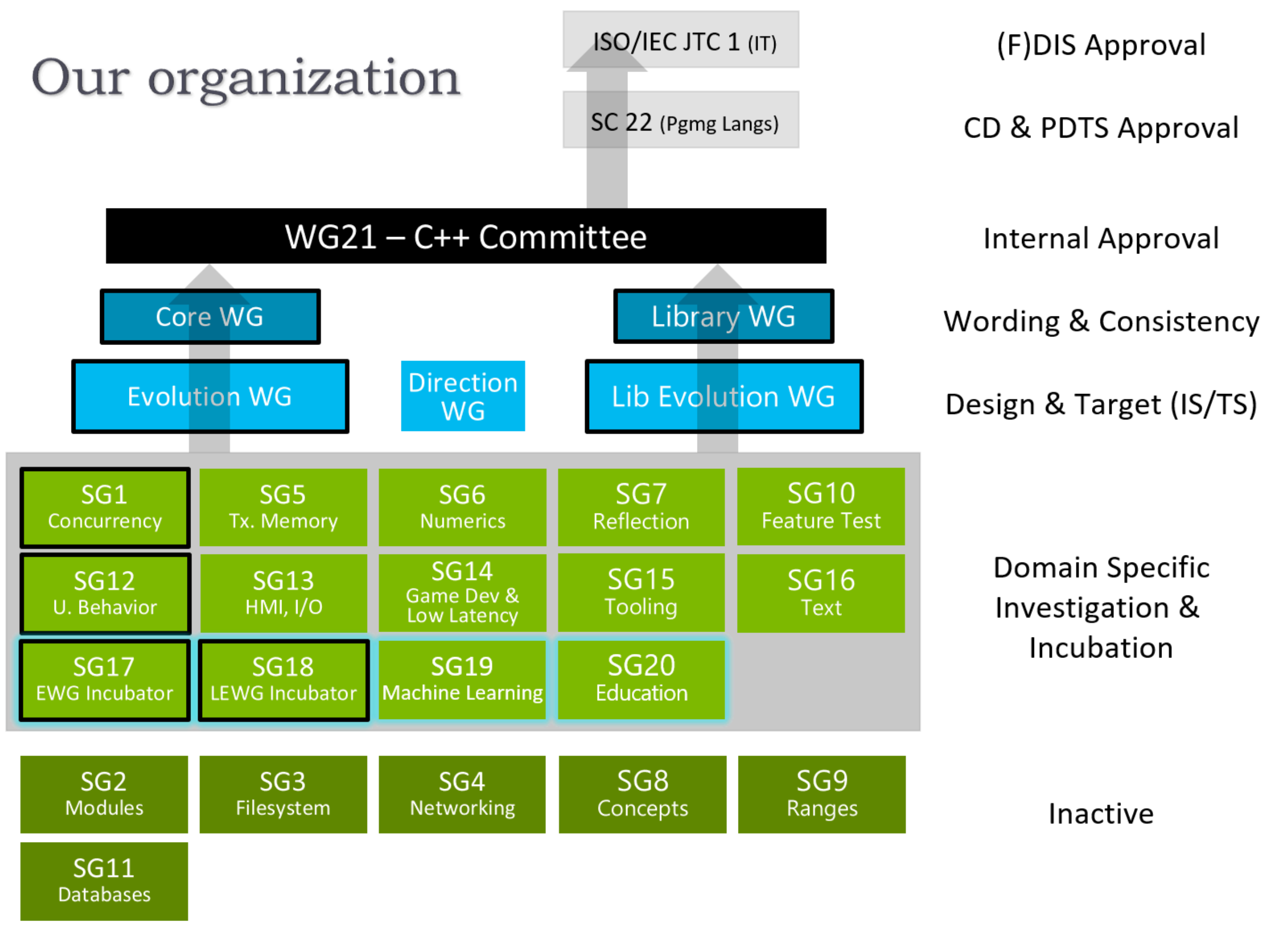
- Threading facilities
- Smart pointers and more containers
- Type traits
- Tools (regexp, chrono, random)

C++ Momentum



WG21: #attendees, #papers
300
250
200
150
100
50
0
1997-07
1997-11
1998-03
1998-10
1999-04
1999-10
2000-04
2000-10
2001-04
2001-10
2002-04
2002-10
2003-04
2003-10
2004-03
2004-10
2005-04
2005-10
2006-04
2006-10
2007-04
2007-07
2007-10
2008-02
2008-06
2008-09
2009-03
2009-07
2009-10
2010-03
2010-08
2010-11
2011-03
2011-08
2012-02
2012-10
2013-04
2013-09
2014-02
2014-06
2014-11
2015-05
2015-10
2016-02
2016-06
2016-11
2017-02
2017-07
2017-11
2018-03
2018-06
2018-11
C++98 shipped
C++11 shipped
C++14 shipped
C++17 shipped

ISO C++ Committee structure



Our organization

ISO/IEC JTC 1 (IT)

SC 22 (Pgmg Langs)

WG21 – C++ Committee

Core WG

Library WG

Evolution WG

Direction WG

Lib Evolution WG

SG1 Concurrency

SG5 Tx. Memory

SG6 Numerics

SG7 Reflection

SG10 Feature Test

SG12 U. Behavior

SG13 HMI, I/O

SG14 Game Dev & Low Latency

SG15 Tooling

SG16 Text

SG17 EWG Incubator

SG18 LEWG Incubator

SG19 Machine Learning

SG20 Education

SG2 Modules

SG3 Filesystem

SG4 Networking

SG8 Concepts

SG9 Ranges

SG11 Databases

(F)DIS Approval

CD & PDTS Approval

Internal Approval

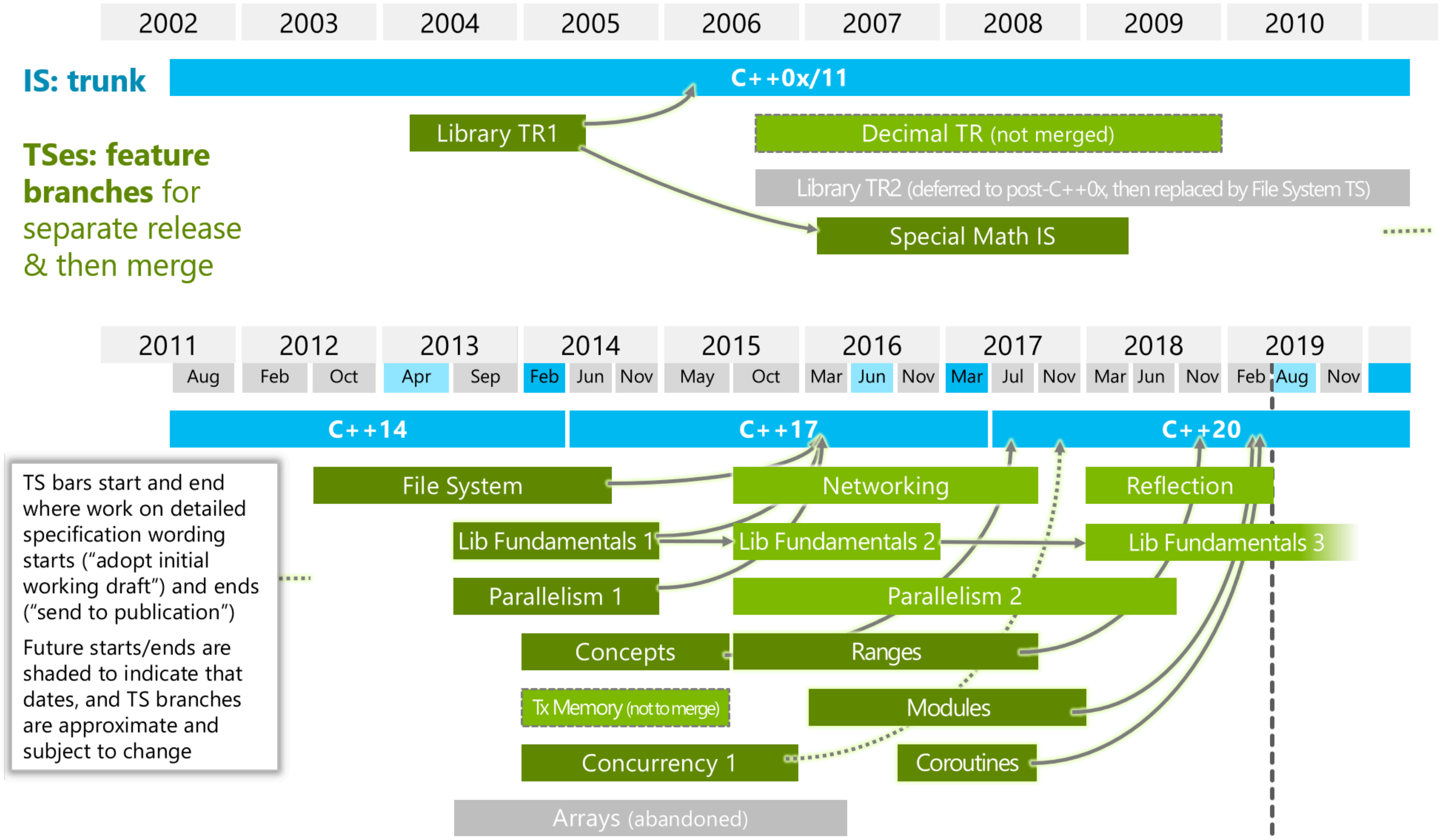
Wording & Consistency

Design & Target (IS/TS)

Domain Specific Investigation & Incubation

Inactive

C++ Timeline



IS: trunk

TSes: feature branches for separate release & then merge

TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send to publication")

Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change

Why do we need so many changes?

- Each new C++ release introduces a lot of new features
- They are *not about* "complicating stuff" even more
- They are here to
  - make our code **easier to design, develop, and maintain**
  - make our products **more stable, secure, and even faster**

HOW THOSE CHANGES AFFECT THE CODE WE WRITE EVERY DAY?

Variable number of function parameters

Implement non-member function template that will allow pushing any number of values to the **`std::vector`** container.

```
std::vector<T> v;
push_back(v, args...);
```

Variable number of function parameters

C++98

```
template<typename T>
void push_back(std::vector<T>&)
{
}

template<typename T, typename Arg1>
void push_back(std::vector<T>& v, const Arg1& arg1)
{
  v.push_back(arg1);
}

template<typename T, typename Arg1, typename Arg2>
void push_back(std::vector<T>& v, const Arg1& arg1, const Arg2& arg2)
{
  v.push_back(arg1);
  v.push_back(arg2);
}

template<typename T, typename Arg1, typename Arg2, typename Arg3>
void push_back(std::vector<T>& v, const Arg1& arg1, const Arg2& arg2, const Arg3& arg3)
// ...
```

Variable number of function parameters

C++11

```
template<typename T>
void push_back(std::vector<T>&)
{
}

template<typename T, typename Arg, typename... Rest>
void push_back(std::vector<T>& v, Arg&& arg, Rest&&... rest)
{
  v.push_back(std::forward<Arg>(arg));
  push_back(v, std::forward<Rest>(rest)...);
}
```

Variable number of function parameters

C++17

```
template<typename T, typename... Args>
void push_back(std::vector<T>& v, Args&&... args)
{
  (v.push_back(std::forward<Args>(args)), ...);
}
```

Time handing

Implement **`run_until(timeout)`** function that will call **`run()`** function in a simple spin-loop until the timeout occurs.

Time handing

C++98

```
void run_until(const timespec& timeout)
{
  const int64_t timeout_ns = 1000000000L * timeout.tv_sec + timeout.tv_nsec;

  // simple spin-loop
  while(true) {
    timespec now;
    clock_gettime(CLOCK_MONOTONIC, &now);
    if(1000000000L * now.tv_sec + now.tv_nsec >= timeout_ns)
      break;
    run();
  }
}
```

Time handing

C++98

```
void run_until(const timespec& timeout);
```

```
timespec now;
clock_gettime(CLOCK_MONOTONIC, &now);
const int64_t end = 1000000000L * now.tv_sec + now.tv_nsec + 300000000L;
const int64_t end_s = end / 1000000000L;
const int64_t end_ns = end - end_s;
const timespec timeout = { end_s, static_cast<long>(end_ns) };
run_until(timeout);
```

Time handing

C++98

```
void run_until(const timespec& timeout);
```

```
timespec now;
clock_gettime(CLOCK_MONOTONIC, &now);
const int64_t end = 1000000000L * now.tv_sec + now.tv_nsec + 300000000L;
const int64_t end_s = end / 1000000000L;
const int64_t end_ns = end - end_s;
const timespec timeout = { end_s, static_cast<long>(end_ns) };
run_until(timeout);
```

Not portable!

Time handing

C++11

```
using namespace std::chrono;
```

Time handing

C++11

```
using namespace std::chrono;
```

```
void run_until(steady_clock::time_point timeout)
{
  // simple spin-loop
  while(steady_clock::now() < timeout)
    run();
}
```

Time handing

C++11

```
using namespace std::chrono;
```

```
void run_until(steady_clock::time_point timeout)
{
  // simple spin-loop
  while(steady_clock::now() < timeout)
    run();
}
```

```
const auto start = steady_clock::now();
run_until(start + milliseconds(300));
```

Time handing

C++14

```cpp
using namespace std::chrono;
```

```cpp
void run_until(steady_clock::time_point timeout)
{
  // simple spin-loop
  while(steady_clock::now() < timeout)
    run();
}
```

```cpp
const auto start = steady_clock::now();
run_until(start + 300ms);
```

Date handling

What day of the week is July 4, 2001?

Date handling

C++98

```
static const char* const wday[] =
{
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "-unknown-"
};
```

```
tm time_str;
time_str.tm_year   = 2001 - 1900;
time_str.tm_mon    = 7 - 1;
time_str.tm_mday   = 4;
time_str.tm_hour   = 0;
time_str.tm_min    = 0;
time_str.tm_sec    = 0;
time_str.tm_isdst  = -1;
if(mktime(&time_str) == static_cast<time_t>(-1))
  time_str.tm_wday = 7;
std::cout << wday[time_str.tm_wday] << '\n';
```

Date handling

C++20

```
using namespace std::chrono;
```

Date handling

C++20

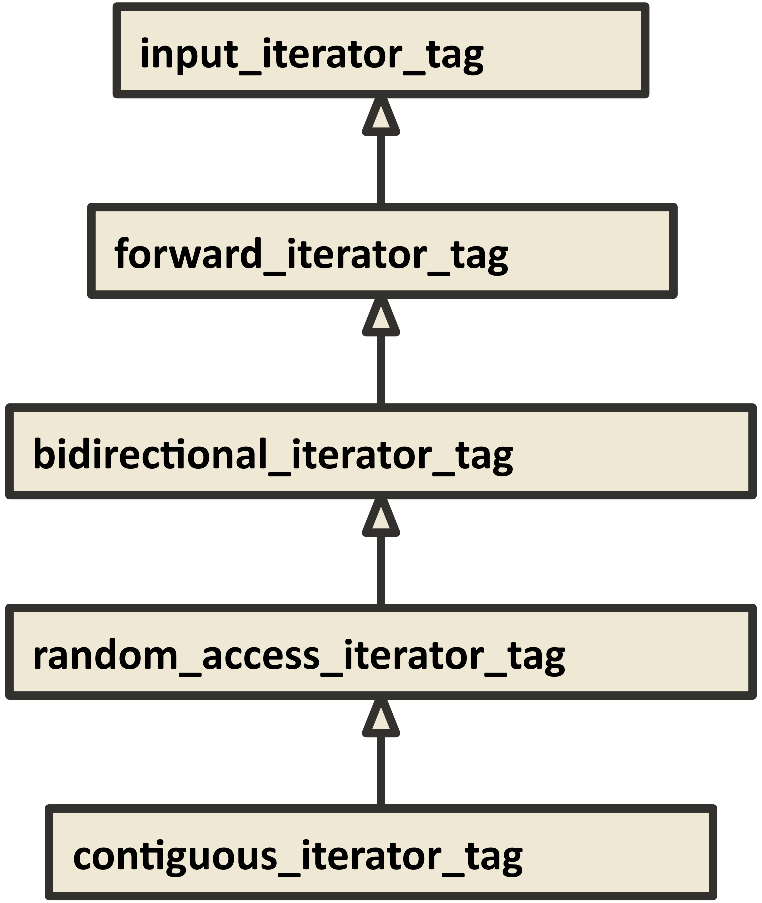
```
using namespace std::chrono;
```

```
std::cout << weekday{jul/4/2001} << '\n';
```

Compile-time dispatch

Implement efficient input iterator advance algorithm.

```
advance(iterator, distance);
```



input_iterator_tag

forward_iterator_tag

bidirectional_iterator_tag

random_access_iterator_tag

contiguous_iterator_tag

Compile-time dispatch

C++98

```cpp
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n)
{
  typedef std::iterator_traits<InputIt> traits;
  advance(it, n, typename traits::iterator_category());
}
```

Compile-time dispatch

C++98

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n)
{
  typedef std::iterator_traits<InputIt> traits;
  advance(it, n, typename traits::iterator_category());
}
```

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::input_iterator_tag)
{
  assert(n >= 0);
  for(; 0 < n; --n)
    ++it;
}
```

Compile-time dispatch

C++98

```cpp
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n)
{
  typedef std::iterator_traits<InputIt> traits;
  advance(it, n, typename traits::iterator_category());
}
```

```cpp
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::input_iterator_tag)
{
  assert(n >= 0);
  for(; 0 < n; --n)
    ++it;
}
```

```cpp
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::bidirectional_iterator_tag)
{
  for(; 0 < n; --n)
    ++it;
  for(; n < 0; ++n)
    --it;
}
```

Compile-time dispatch

C++98

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n)
{
  typedef std::iterator_traits<InputIt> traits;
  advance(it, n, typename traits::iterator_category());
}
```

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::input_iterator_tag)
{
  assert(n >= 0);
  for(; 0 < n; --n)
    ++it;
}
```

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::bidirectional_iterator_tag)
{
  for(; 0 < n; --n)
    ++it;
  for(; n < 0; ++n)
    --it;
}
```

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::random_access_iterator_tag)
{
  it += n;
}
```

Compile-time dispatch

C++11

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n)
{
  using traits = std::iterator_traits<InputIt>;
  advance(it, n, typename traits::iterator_category());
}
```

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::input_iterator_tag)
{
  assert(n >= 0);
  for(; 0 < n; --n)
    ++it;
}
```

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::bidirectional_iterator_tag)
{
  for(; 0 < n; --n)
    ++it;
  for(; n < 0; ++n)
    --it;
}
```

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n,
             std::random_access_iterator_tag)
{
  it += n;
}
```

Compile-time dispatch

C++17

```
template<typename InputIt, typename Distance>
constexpr void advance(InputIt& it, Distance n)
{
  using category = typename std::iterator_traits<InputIt>::iterator_category;
  if constexpr(std::is_base_of_v<std::random_access_iterator_tag, category>) {
    it += n;
  }
  else if constexpr(std::is_base_of_v<std::bidirectional_iterator_tag, category>) {
    for(; 0 < n; --n)
      ++it;
    for(; n < 0; ++n)
      --it;
  }
  else {
    assert(n >= 0);
    for(; 0 < n; --n)
      ++it;
  }
}
```

Compile-time dispatch

C++20

```
template<std::InputIterator It>
constexpr void my_advance(It& it, std::iter_difference_t<It> n)
{
  if constexpr(std::RandomAccessIterator<It>) {
    it += n;
  }
  else if constexpr(std::BidirectionalIterator<It>) {
    for(; 0 < n; --n)
      ++it;
    for(; n < 0; ++n)
      --it;
  }
  else {
    [[assert: n >= 0]]
    for(; 0 < n; --n)
      ++it;
  }
}
```

Compile-time calculations

Provide the implementation of some calculation algorithm that will work in run-time and will allow to pre-calculate an array of typical values in the compile-time.

Compile-time calculations

C++98

```
int factorial(int n)
{
  assert(n >= 0);
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
foo(factorial(n));  // not compile-time
foo(factorial(4));  // not compile-time
```

Compile-time calculations

C++98

```cpp
int factorial(int n)
{
  assert(n >= 0);
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```cpp
foo(factorial(n));   // not compile-time
foo(factorial(4));   // not compile-time
```

```cpp
template<int N>
struct Factorial {
  static const int value =
      N * Factorial<N - 1>::value;
};

template<>
struct Factorial<0> {
  static const int value = 1;
};
```

```cpp
int array[Factorial<4>::value];
```

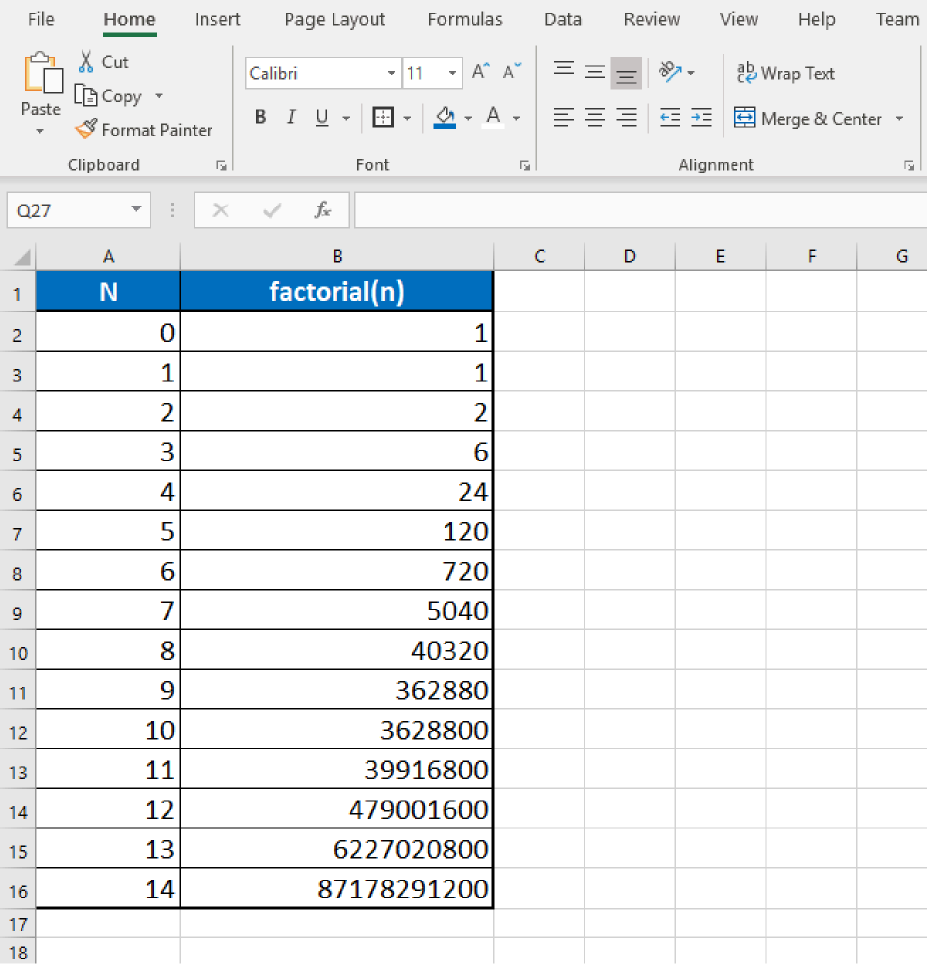
```cpp
const int precalc_values[] = {
  Factorial<0>::value,
  Factorial<1>::value,
  Factorial<2>::value
};
```

```cpp
foo(factorial<4>::value);   // compile-time
// foo(factorial<n>::value);   // compile-time error
```

Just too hard!

- 2 separate implementations
  - hard to keep in sync
- Template *metaprogramming* is hard!
- Easier to precalculate in Excel and hardcode

  results in the code ;-)

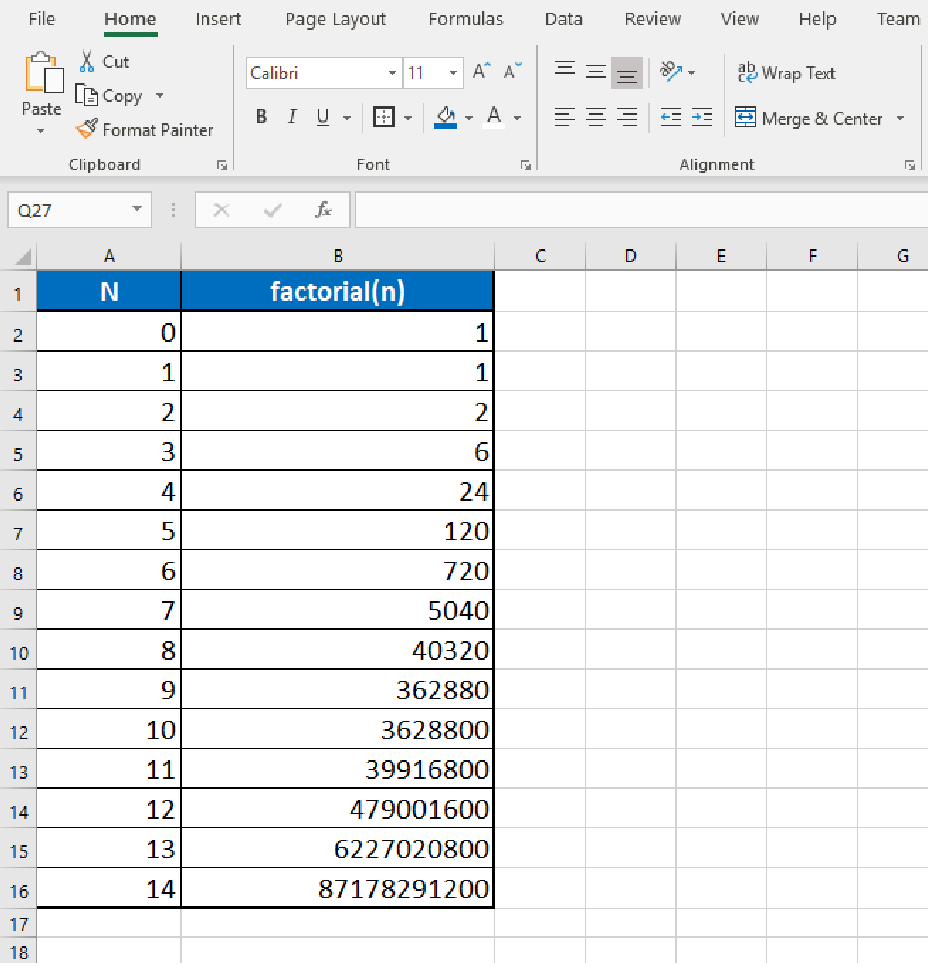


	A	B
1	N	factorial(n)
2	0	1
3	1	1
4	2	2
5	3	6
6	4	24
7	5	120
8	6	720
9	7	5040
10	8	40320
11	9	362880
12	10	3628800
13	11	39916800
14	12	479001600
15	13	6227020800
16	14	87178291200
17		
18		

Just too hard!

- 2 separate implementations
  - hard to keep in sync
- Template *metaprogramming* is hard!
- Easier to precalculate in Excel and hardcode results in the code ;-)

```
const int precalc_values[] = {
  1,
  1,
  2,
  6,
  24,
  120,
  720,
  // ...
};
```



N	factorial(n)
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600
13	6227020800
14	87178291200

Compile-time calculations

C++11

```
constexpr int factorial(int n)
{
  // assert(n >= 0);
  return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array<int, 3> precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24, "Error");
foo(factorial(4));  // compile-time not guaranteed
foo(factorial(n));  // not compile-time
```

Compile-time calculations

C++14

```cpp
constexpr int factorial(int n)
{
  assert(n >= 0);
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```cpp
std::array<int, factorial(4)> array;
```

```cpp
constexpr std::array<int, 3> precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```cpp
static_assert(factorial(4) == 24, "Error");
foo(factorial(4));  // compile-time not guaranteed
foo(factorial(n));  // not compile-time
```

Compile-time calculations

C++17

```
constexpr int factorial(int n)
{
  assert(n >= 0);
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24);
foo(factorial(4));  // compile-time not guaranteed
foo(factorial(n));  // not compile-time
```

Compile-time calculations

C++20

```
constexpr int factorial(int n)
  [[expects: n >= 0]]
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24);
foo(factorial(4));  // compile-time not guaranteed
foo(factorial(n));  // not compile-time
```

Compile-time calculations

C++20

```
constexpr int factorial(int n)
  [[expects: n >= 0]]
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24);
foo(factorial(4));  // compile-time not guaranteed
foo(factorial(n));  // not compile-time
```

```
consteval int factorial(int n)
  [[expects: n >= 0]]
{
  int result = n;
  while(n > 1)
    result *= --n;
  return result;
}
```

```
std::array<int, factorial(4)> array;
```

```
constexpr std::array precalc_values = {
  factorial(0),
  factorial(1),
  factorial(2)
};
```

```
static_assert(factorial(4) == 24);
foo(factorial(4));  // compile-time
// foo(factorial(n));  // compile-time error
```

What does that function do? Is it correct?

```cpp
auto foo(const orders& o, client_id c)
{
  auto first = o.begin();
  orders::const_iterator it;
  typename std::iterator_traits<orders::const_iterator>::difference_type count, step;
  count = std::distance(first, o.end());

  while(count > 0) {
    it = o.begin();
    step = count / 2;
    std::advance(it, step);
    if(get_by_client_id(*it, c)) {
      first = ++it;
      count -= step + 1;
    }
    else
      count = step;
  }
  return first;
}
```

What does that function do? Is it correct?

```cpp
auto foo(const orders& o, client_id c)
{
  return std::lower_bound(begin(o), end(o), c, get_by_client_id);
}
```

What does that function do? Is it correct?

```
auto foo(const orders& o, client_id c)
{
  return std::lower_bound(begin(o), end(o), c, get_by_client_id);
}
```

Code that says WHAT is just as readable as code that says HOW.
We are used to seeing code that says HOW. It's more familiar.
Code that says WHAT is more likely to remain robust.

Evolution of algorithms

Implement a **get_order** that will return an iterator to **order** for a given **client_id** from a range of orders sorted by **client_id**.

```cpp
using client_id = int;

struct account {
  client_id client;
  // ...
};

struct order {
  client_id client;
  // ...
};

using orders = std::vector<order>;
```

```cpp
orders get_orders();
```

```cpp
auto get_order(const orders& o, client_id c);
```

Evolution of algorithms

C++98

```
struct compare_orders_by_client_id {
  bool operator()(const order& l, const order& r) const { return l.client < r.client; };
};

void sort_orders_by_client_id(orders& o)
{
  std::sort(o.begin(), o.end(), compare_orders_by_client_id());
}
```

Evolution of algorithms

C++98

```cpp
struct compare_orders_by_client_id {
  bool operator()(const order& l, const order& r) const { return l.client < r.client; };
};

void sort_orders_by_client_id(orders& o)
{
  std::sort(o.begin(), o.end(), compare_orders_by_client_id());
}
```

```cpp
struct get_order_by_client_id {
  bool operator()(const order& o, const client_id& c) const { return o.client < c; };
};

orders::const_iterator get_order(const orders& o, client_id c)
{
  return std::lower_bound(o.begin(), o.end(), c, get_order_by_client_id());
}
```

Evolution of algorithms

C++98

```
struct compare_orders_by_client_id {
  bool operator()(const order& l, const order& r) const { return l.client < r.client; };
};

void sort_orders_by_client_id(orders& o)
{
  std::sort(o.begin(), o.end(), compare_orders_by_client_id());
}
```

```
struct get_order_by_client_id {
  bool operator()(const order& o, const client_id& c) const { return o.client < c; };
};

orders::const_iterator get_order(const orders& o, client_id c)
{
  return std::lower_bound(o.begin(), o.end(), c, get_order_by_client_id());
}
```

```
orders::const_iterator it = get_order(get_orders(), 123);
use(*it);
```

Evolution of algorithms

C++98

```
struct get_account_by_client_id {
  bool operator()(const account& a, const client_id& c) const { return a.client < c; };
};

struct compare_accounts_by_client_id {
  bool operator()(const account& l, const account& r) const { return l.client < r.client; };
};
```

Evolution of algorithms

C++11

```
void sort_orders_by_client_id(orders& o)
{
  auto compare_orders_by_client_id = [](const order& l, const order& r){ return l.client < r.client; };
  std::sort(begin(o), end(o), compare_orders_by_client_id);
}
```

```
orders::const_iterator get_order(const orders& o, client_id c)
{
  auto get_order_by_client_id = [](const order& o, const client_id& c){ return o.client < c; };
  return std::lower_bound(begin(o), end(o), c, get_order_by_client_id);
}
```

```
auto it = get_order(get_orders(), 123);
use(*it);
```

Evolution of algorithms

C++11

```
void sort_orders_by_client_id(orders& o)
{
  auto compare_orders_by_client_id = [](const order& l, const order& r){ return l.client < r.client; };
  std::sort(begin(o), end(o), compare_orders_by_client_id);
}
```

```
orders::const_iterator get_order(const orders& o, client_id c)
{
  auto get_order_by_client_id = [](const order& o, const client_id& c){ return o.client < c; };
  return std::lower_bound(begin(o), end(o), c, get_order_by_client_id);
}
```

```
auto it = get_order(get_orders(), 123);
use(*it);
```

```
auto get_account_by_client_id = [](const account& a, const client_id& c){ return a.client < c; };
auto compare_accounts_by_client_id = [](const account& l, const account& r){ return l.client < r.client; };
```

Evolution of algorithms

C++14

```
auto compare_by_client_id = [](const auto& l, const auto& r){ return l.client < r.client; };
auto get_by_client_id = [](const auto& e, const client_id& c){ return e.client < c; };
```

Evolution of algorithms

C++14

```cpp
auto compare_by_client_id = [](const auto& l, const auto& r){ return l.client < r.client; };
auto get_by_client_id = [](const auto& e, const client_id& c){ return e.client < c; };
```

```cpp
void sort_orders_by_client_id(orders& o)
{
  std::sort(begin(o), end(o), compare_by_client_id);
}
```

```cpp
orders::const_iterator get_order(const orders& o, client_id c)
{
  return std::lower_bound(begin(o), end(o), c, get_by_client_id);
}
```

```cpp
auto it = get_order(get_orders(), 123);
use(*it);
```

Evolution of algorithms

C++20

```
auto to_client_id = [](const auto& e){ return e.client; };
```

Evolution of algorithms

C++20

```
auto to_client_id = [](const auto& e){ return e.client; };
```

```
void sort_orders_by_client_id(orders& o)
{
  std::ranges::sort(o, std::ranges::less(), to_client_id);
}
```

```
orders::const_iterator get_order(const orders& o, client_id c)
  [[expects audit: std::ranges::is_sorted(o, std::ranges::less(), to_client_id)]]
{
  return std::ranges::lower_bound(o, c, std::ranges::less(), to_client_id);
}
```

```
auto it = get_order(get_orders(), 123);
use(*it);
```

DID YOU SPOT A BUG ON THE LAST SLIDE?

New family of lifetime issues

```
orders get_orders();
```

New family of lifetime issues

```
orders get_orders();
```

```
orders::const_iterator get_order(const orders& o, client_id c)
  [[expects audit: std::ranges::is_sorted(o, std::ranges::less(), to_client_id)]]
{
  return std::ranges::lower_bound(o, c, std::ranges::less(), to_client_id);
}
```

```
auto it = get_order(get_orders(), 123);
use(*it);
```

Safety included

```
orders get_orders();
```

```
template<std::ranges::ForwardRange R>
auto get_order(R&& o, client_id c)
  [[expects audit: std::ranges::is_sorted(o, std::ranges::less(), to_client_id)]]
{
  return std::ranges::lower_bound(std::forward<R>(o), c, std::ranges::less(), to_client_id);
}
```

```
auto it = get_order(get_orders(), 123);
use(*it);
```

Safety included

```
orders get_orders();
```

```
template<std::ranges::ForwardRange R>
auto get_order(R&& o, client_id c)
  [[expects audit: std::ranges::is_sorted(o, std::ranges::less(), to_client_id)]]
{
  return std::ranges::lower_bound(std::forward<R>(o), c, std::ranges::less(), to_client_id);
}
```

```
auto it = get_order(get_orders(), 123);
use(*it);
```

```
<source>:53:7: error: no match for 'operator*' (operand type is 'std::ranges::dangling')
   53 |   use(*it);
      |       ^~~
```

Safety included

```
orders get_orders();
```

```
template<std::ranges::ForwardRange R>
auto get_order(R&& o, client_id c)
  [[expects audit: std::ranges::is_sorted(o, std::ranges::less(), to_client_id)]]
{
  return std::ranges::lower_bound(std::forward<R>(o), c, std::ranges::less(), to_client_id);
}
```

```
auto orders = get_orders();
auto it = get_order(orders, 123);
use(*it);
```

Data range processing

Print the age of the first N adult persons in any range of persons.

```cpp
struct person {
  std::string name;
  std::string age;
};
```

```cpp
std::vector<person> people;
print_age_of_first_n_adults(people, 10);
```

Data range processing

C++98

```cpp
template<typename InputRange>
void print_age_of_first_n_adults(const InputRange& people, int n)
{
  int count = 0;
  for(typename InputRange::const_iterator it = people.begin(); it != people.end(); ++it) {
    const int age = std::atoi(it->age.c_str());
    if(age >= 18) {
      if(count++ == n)
        break;
      std::cout << age << '\n';
    }
  }
}
```

Data range processing

C++98

```cpp
template<typename InputRange>
void print_age_of_first_n_adults(const InputRange& people, int n)
{
  int count = 0;
  for(typename InputRange::const_iterator it = people.begin(); it != people.end(); ++it) {
    const int age = std::atoi(it->age.c_str());
    if(age >= 18) {
      if(count++ == n)
        break;
      std::cout << age << '\n';
    }
  }
}
```

Does not work with C-arrays

Data range processing

C++11

```
template<typename InputRange>
void print_age_of_first_n_adults(const InputRange& people, int n)
{
  static_assert(std::is_same<typename std::iterator_traits<decltype(std::begin(people))>::value_type,
                             person>::value, "Bad type");
  int count = 0;
  for(const person& p : people) {
    const int age = std::stoi(p.age);
    if(age >= 18) {
      if(count++ == n)
        break;
      std::cout << age << '\n';
    }
  }
}
```

Data range processing

C++17

```cpp
template<typename InputRange>
void print_age_of_first_n_adults(const InputRange& people, int n)
{
  static_assert(std::is_same_v<typename std::iterator_traits<decltype(std::begin(people))>::value_type,
                               person>);
  int count = 0;
  for(const person& p : people) {
    const int age = to_int(p.age);
    if(age >= 18) {
      if(count++ == n)
        break;
      std::cout << age << '\n';
    }
  }
}
```

```cpp
int to_int(const std::string_view& txt)
{
  int age = 0;
  std::from_chars(begin(txt), end(txt), age);
  return age;
}
```

Data range processing

C++20

```cpp
using namespace std::ranges;

template<InputRange R>
  requires Same<range_value_t<R>, person>
void print_age_of_first_n_adults(const R& people, int n)
{
  using namespace std::ranges;

  auto to_age = [](const person& p) { return to_int(p.age); };
  auto adult = [](int age) { return age >= 18; };

  for(int age : people | view::transform(to_age) | view::filter(adult) | view::take(n))
    std::cout << age << '\n';
}
```

```cpp
int to_int(const std::string_view& txt)
{
  int age = 0;
  std::from_chars(begin(txt), end(txt), age);
  return age;
}
```

Wrapper types

Wrapper types of type **T** should be implicitly constructible from the type **U** only if type **U** is implicitly convertible to type **T**.

```
std::pair<std::string, std::string> safe()
{
  return {"meow", "purr"}; // ok
}

std::pair<std::vector<int>, std::vector<int>> unsafe()
{
  return {11, 22}; // error
}
```

Wrapper types

C++11

```
template<typename T1, typename T2>
class pair {
public:
  template<typename U1 = T1, typename U2 = T2,
           typename std::enable_if<
               std::is_constructible<T1, U1>::value &&
               std::is_constructible<T2, U2>::value &&
               std::is_convertible<U1, T1>::value &&
               std::is_convertible<U2, T2>::value,
               int>::type = 0>
  constexpr pair(U1&&, U2&&);

  template<typename U1 = T1, typename U2 = T2,
           typename std::enable_if<
               std::is_constructible<T1, U1>::value &&
               std::is_constructible<T2, U2>::value &&
               !(std::is_convertible<U1, T1>::value &&
                 std::is_convertible<U2, T2>::value),
               int>::type = 0>
  explicit constexpr pair(U1&&, U2&&);

  // ...
};
```

Wrapper types

C++14

```
template<typename T1, typename T2>
class pair {
public:
  template<typename U1 = T1, typename U2 = T2,
           std::enable_if_t<
               std::is_constructible<T1, U1>::value &&
               std::is_constructible<T2, U2>::value &&
               std::is_convertible<U1, T1>::value &&
               std::is_convertible<U2, T2>::value,
               int> = 0>
  constexpr pair(U1&&, U2&&);

  template<typename U1 = T1, typename U2 = T2,
           std::enable_if_t<
               std::is_constructible<T1, U1>::value &&
               std::is_constructible<T2, U2>::value &&
               !(std::is_convertible<U1, T1>::value &&
                 std::is_convertible<U2, T2>::value),
               int> = 0>
  explicit constexpr pair(U1&&, U2&&);

  // ...
};
```

Wrapper types

C++17

```
template<typename T1, typename T2>
class pair {
public:
  template<typename U1 = T1, typename U2 = T2,
           std::enable_if_t<
               std::is_constructible_v<T1, U1> &&
               std::is_constructible_v<T2, U2> &&
               std::is_convertible_v<U1, T1> &&
               std::is_convertible_v<U2, T2>,
               int> = 0>
  constexpr pair(U1&&, U2&&);

  template<typename U1 = T1, typename U2 = T2,
           std::enable_if_t<
               std::is_constructible_v<T1, U1> &&
               std::is_constructible_v<T2, U2> &&
               !(std::is_convertible_v<U1, T1> &&
                 std::is_convertible_v<U2, T2>),
               int> = 0>
  explicit constexpr pair(U1&&, U2&&);

  // ...
};
```

Wrapper types

C++20

```cpp
template<typename T1, typename T2>
class pair {
public:
  template<typename U1 = T1, typename U2 = T2>
      requires std::is_constructible_v<T1, U1> &&
               std::is_constructible_v<T2, U2> &&
               std::is_convertible_v<U1, T1> &&
               std::is_convertible_v<U2, T2>
  constexpr pair(U1&&, U2&&);

  template<typename U1 = T1, typename U2 = T2>
      requires std::is_constructible_v<T1, U1> &&
               std::is_constructible_v<T2, U2>
  explicit constexpr pair(U1&&, U2&&);

  // ...
};
```

Wrapper types

C++20

```
template<typename T1, typename T2>
class pair {
public:
  template<typename U1 = T1, typename U2 = T2>
      requires std::is_constructible_v<T1, U1> &&
               std::is_constructible_v<T2, U2>
  explicit(!std::is_convertible_v<U1, T1> || !std::is_convertible_v<U2, T2>)
  constexpr pair(U1&&, U2&&);

  // ...
};
```

Custom regular types

Implement custom string type and make it constructible from and comparable with C-like strings.

Custom regular types

C++11

```
class ci_string {
  std::string s;
public:
  // ...
  friend bool operator==(const ci_string& a, const ci_string& b) { return ci_compare(a.s.c_str(), b.s.c_str()) != 0; }
  friend bool operator< (const ci_string& a, const ci_string& b) { return ci_compare(a.s.c_str(), b.s.c_str()) <  0; }
  friend bool operator!=(const ci_string& a, const ci_string& b) { return !(a == b); }
  friend bool operator> (const ci_string& a, const ci_string& b) { return b < a; }
  friend bool operator>=(const ci_string& a, const ci_string& b) { return !(a < b); }
  friend bool operator<=(const ci_string& a, const ci_string& b) { return !(b < a); }
  friend bool operator==(const ci_string& a, const char* b) { return ci_compare(a.s.c_str(), b) != 0; }
  friend bool operator< (const ci_string& a, const char* b) { return ci_compare(a.s.c_str(), b) <  0; }
  friend bool operator!=(const ci_string& a, const char* b) { return !(a == b); }
  friend bool operator> (const ci_string& a, const char* b) { return b < a; }
  friend bool operator>=(const ci_string& a, const char* b) { return !(a < b); }
  friend bool operator<=(const ci_string& a, const char* b) { return !(b < a); }
  friend bool operator==(const char* a, const ci_string& b) { return ci_compare(a, b.s.c_str()) != 0; }
  friend bool operator< (const char* a, const ci_string& b) { return ci_compare(a, b.s.c_str()) <  0; }
  friend bool operator!=(const char* a, const ci_string& b) { return !(a == b); }
  friend bool operator> (const char* a, const ci_string& b) { return b < a; }
  friend bool operator>=(const char* a, const ci_string& b) { return !(a < b); }
  friend bool operator<=(const char* a, const ci_string& b) { return !(b < a); }
};
```

Custom regular types

C++20

```
class ci_string {
  std::string s;
public:
  // ...

  std::weak_ordering operator<=>(const ci_string& b) const { return ci_compare(s.c_str(), b.s.c_str()); }
  std::weak_ordering operator<=>(const char* b) const      { return ci_compare(s.c_str(), b); }
};
```

A simple string view

Implement a simple string view class that will work with the below code.

```cpp
constexpr str_view s1 = "alabama";
constexpr str_view s2 = "mississippi";
for(auto p1=s1.begin(), p2=s2.begin(); p1!=s1.end(), p2!=s2.end(); ++p1, ++p2)
  std::cout << "1: " << *p1 << "; 2: " << *p2 << '\n';
```

A simple string view

C++11

```cpp
class str_view {
  const char* ptr_ = nullptr;
  std::size_t size_ = 0;
public:
  using size_type = std::size_t;
  class const_iterator;

  constexpr str_view() = default;
  constexpr str_view(const char* ptr) noexcept : ptr_(ptr)
  {
    while(*ptr++)
      size_++;
  }
  constexpr str_view(const char* ptr, std::size_t size) noexcept : ptr_(ptr), size_(size) {}
  constexpr size_t size() const noexcept { return size_; }
  constexpr const_iterator begin() const noexcept { return ptr_; }
  constexpr const_iterator end() const noexcept { return ptr_ + size_; }
};
```

```cpp
constexpr str_view s1 = "alabama";
constexpr str_view s2 = "mississippi";
for(auto p1=s1.begin(), p2=s2.begin(); p1!=s1.end(), p2!=s2.end(); ++p1, ++p2)
  std::cout << "1: " << *p1 << "; 2: " << *p2 << '\n';
```

A simple string view

C++11

```cpp
class str_view::const_iterator {
  const char* ptr_ = nullptr;
public:
  constexpr const_iterator() = default;
  constexpr const_iterator(const char* ptr) noexcept : ptr_(ptr) {}

  constexpr char operator*() const noexcept { return *ptr_; }

  const_iterator& operator++() noexcept { ++ptr_; return *this; }
  const_iterator operator++(int) noexcept { auto temp = ptr_++; return temp; }

  constexpr bool operator==(const const_iterator& other) noexcept { return ptr_ == other.ptr_; }
  constexpr bool operator!=(const const_iterator& other) noexcept { return !(*this == other); }
};
```

```cpp
constexpr str_view s1 = "alabama";
constexpr str_view s2 = "mississippi";
for(auto p1=s1.begin(), p2=s2.begin(); p1!=s1.end(), p2!=s2.end(); ++p1, ++p2)
  std::cout << "1: " << *p1 << "; 2: " << *p2 << '\n';
```

DID YOU SPOT A BUG ON THE LAST SLIDE?

A simple string view

C++17

```cpp
class str_view {
  const char* ptr_ = nullptr;
  std::size_t size_ = 0;
public:
  using size_type = std::size_t;
  class const_iterator;

  constexpr str_view() = default;
  constexpr str_view(const char* ptr) noexcept : ptr_(ptr)
  {
    while(*ptr++)
      size_++;
  }
  constexpr str_view(const char* ptr, std::size_t size) noexcept : ptr_(ptr), size_(size) {}
  [[nodiscard]] constexpr size_t size() const noexcept { return size_; }
  [[nodiscard]] constexpr const_iterator begin() const noexcept { return ptr_; }
  [[nodiscard]] constexpr const_iterator end() const noexcept { return ptr_ + size_; }
};
```

```cpp
constexpr str_view s1 = "alabama";
constexpr str_view s2 = "mississippi";
for(auto p1=s1.begin(), p2=s2.begin(); p1!=s1.end(), p2!=s2.end(); ++p1, ++p2)
  std::cout << "1: " << *p1 << "; 2: " << *p2 << '\n';
```

A simple string view

C++17

```
class str_view::const_iterator {
  const char* ptr_ = nullptr;
public:
  constexpr const_iterator() = default;
  constexpr const_iterator(const char* ptr) noexcept : ptr_(ptr) {}

  [[nodiscard]] constexpr char operator*() const noexcept { return *ptr_; }

  constexpr const_iterator& operator++() noexcept { ++ptr_; return *this; }
  constexpr const_iterator operator++(int) noexcept { auto temp = ptr_++; return temp; }

  [[nodiscard]] constexpr bool operator==(const const_iterator& other) noexcept { return ptr_ == other.ptr_; }
  [[nodiscard]] constexpr bool operator!=(const const_iterator& other) noexcept { return !(*this == other); }
};
```

```
constexpr str_view s1 = "alabama";
constexpr str_view s2 = "mississippi";
for(auto p1=s1.begin(), p2=s2.begin(); p1!=s1.end(), p2!=s2.end(); ++p1, ++p2)
  std::cout << "1: " << *p1 << "; 2: " << *p2 << '\n';
```

A simple string view

C++17

```
class str_view::const_iterator {
  const char* ptr_ = nullptr;
public:
  constexpr const_iterator() = default;
  constexpr const_iterator(const char* ptr) noexcept : ptr_(ptr) {}

  [[nodiscard]] constexpr char operator*() const noexcept { return *ptr_; }

  constexpr const_iterator& operator++() noexcept { ++ptr_; return *this; }
  constexpr const_iterator operator++(int) noexcept { auto temp = ptr_++; return temp; }

  [[nodiscard]] constexpr bool operator==(const const_iterator& other) noexcept { return ptr_ == other.ptr_; }
  [[nodiscard]] constexpr bool operator!=(const const_iterator& other) noexcept { return !(*this == other); }
};
```

```
constexpr str_view s1 = "alabama";
constexpr str_view s2 = "mississippi";
for(auto p1=s1.begin(), p2=s2.begin(); p1!=s1.end(), p2!=s2.end(); ++p1, ++p2)
  std::cout << "1: " << *p1 << "; 2: " << *p2 << '\n';
```

```
main.cpp:304:44: error: ignoring return value of 'bool str_view::const_iterator::operator!=(const str_view::const_iterator&)',
                        declared with attribute nodiscard [-Werror=unused-result]
   for(auto p1=s1.begin(), p2=s2.begin(); p1!=s1.end(), p2!=s2.end(); ++p1, ++p2) {
                                          ~~^~~~~~~~~
```

Custom containers

Implement custom hash map.
NOTE: Please use Empty Base Optimization (EBO) wherever possible to save memory footprint.

```cpp
template<typename Key, typename Value, typename Hash, typename Pred, typename Allocator>
class my_hash_map;
```

Custom containers

C++14

```
template<typename T, int Idx = 0, bool UseEbo = !std::is_final_v<T> && std::is_empty_v<T>>
struct ebo_helper;

template<typename T, int Idx>
struct ebo_helper<T, Idx, true> : private T {
  template<typename U>
  constexpr explicit ebo_helper(U&& t) : T{std::forward<U>(t)} {}
  constexpr T& get() { return *this; }
};

template<typename T, int Idx>
struct ebo_helper<T, Idx, false> {
  template<typename U>
  constexpr explicit ebo_helper(U&& t) : t_{std::forward<U>(t)} {}
  constexpr T& get() { return t_; }
private:
  T t_;
};
```

Custom containers

C++14

```cpp
template<typename Key, typename Value, typename Hash, typename Pred, typename Allocator>
class my_hash_map : ebo_helper<Hash, 0>, ebo_helper<Pred, 1>, ebo_helper<Allocator, 2> {
  Hash& hasher() { return ebo_helper<Hash, 0>::get(); }
  Pred& pred() { return ebo_helper<Pred, 1>::get(); }
  Allocator& alloc() { return ebo_helper<Allocator, 2>::get(); }
  // ...
};
```

Custom containers

C++20

```
template<typename Key, typename Value, typename Hash, typename Pred, typename Allocator>
class my_hash_map {
  [[no_unique_address]] Hash hasher;
  [[no_unique_address]] Pred pred;
  [[no_unique_address]] Allocator alloc;
  // ...
};
```

Reflection

Check if provided integral value is legal value of an enumeration type.

Reflection

C++98

```cpp
enum fruit { apple = 1; banana; orange; };
```

```cpp
bool is_valid_fruit(int value)
{
  return value == apple || value == banana || value == orange;
}
```

```cpp
bool not_true = is_valid_fruit(55);
```

Reflection

C++23 (???)

```cpp
template<typename Enum, typename Integral>
  requires std::is_enum_v<Enum> && std::is_integral_v<Integral>
constexpr bool is_one_of_enumerators(Integral value)
{
  for(constexpr e : reflexpr(Enum).enumerators())
    if(e.value() == value)
      return true;
  return false;
}
```

```cpp
enum fruit { apple = 1; banana; orange; };
constexpr bool not_true = is_one_of_enumerators<fruit>(55);
```

C++ Language Evolution

C++11 was a game changer on the market, C++14 and C++17 provided a lot of important improvements that allow us to write portable, safer, and faster programs in a shorter time.

C++ Language Evolution

C++11 was a game changer on the market, C++14 and C++17 provided a lot of important improvements that allow us to write portable, safer, and faster programs in a shorter time.

BEWARE, C++20 is going to be HUGE again!

C++ Language Evolution

C++11 was a game changer on the market, C++14 and C++17 provided a lot of important improvements that allow us to write portable, safer, and faster programs in a shorter time.

BEWARE, C++20 is going to be HUGE again!

Do not stay behind...

THANK YOU!

https://train-it.eu