C++11 was only the beginning

DO NOT STAY BEHIND

Mateusz Pusz
September 21, 2019

Confidential Information

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**
  - *imperative*, *object-based*, *object-oriented*, *generic*, and *functional programming* features
  - developer is not tied to any specific programming paradigm by the language

What is C++?

- *C++ is no longer C with classes*
- C++ is a **general-purpose programming language**
  - *imperative*, *object-based*, *object-oriented*, *generic*, and *functional programming* features
  - developer is not tied to any specific programming paradigm by the language
- **High-level abstractions at low cost**

What is C++?

- *C++ is no longer C with classes*
- C++ is a **general-purpose programming language**
  - *imperative*, *object-based*, *object-oriented*, *generic*, and *functional programming* features
  - developer is not tied to any specific programming paradigm by the language
- **High-level abstractions at low cost**
- **Low-level access** when needed

What is C++?

- *C++ is no longer C with classes*
- C++ is a **general-purpose programming language**
  - *imperative*, *object-based*, *object-oriented*, *generic*, and *functional programming* features
  - developer is not tied to any specific programming paradigm by the language
- **High-level abstractions at low cost**
- **Low-level access** when needed
- 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**
  - *imperative*, *object-based*, *object-oriented*, *generic*, and *functional programming* features
  - developer is not tied to any specific programming paradigm by the language
- **High-level abstractions at low cost**
- **Low-level access** when needed
- 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

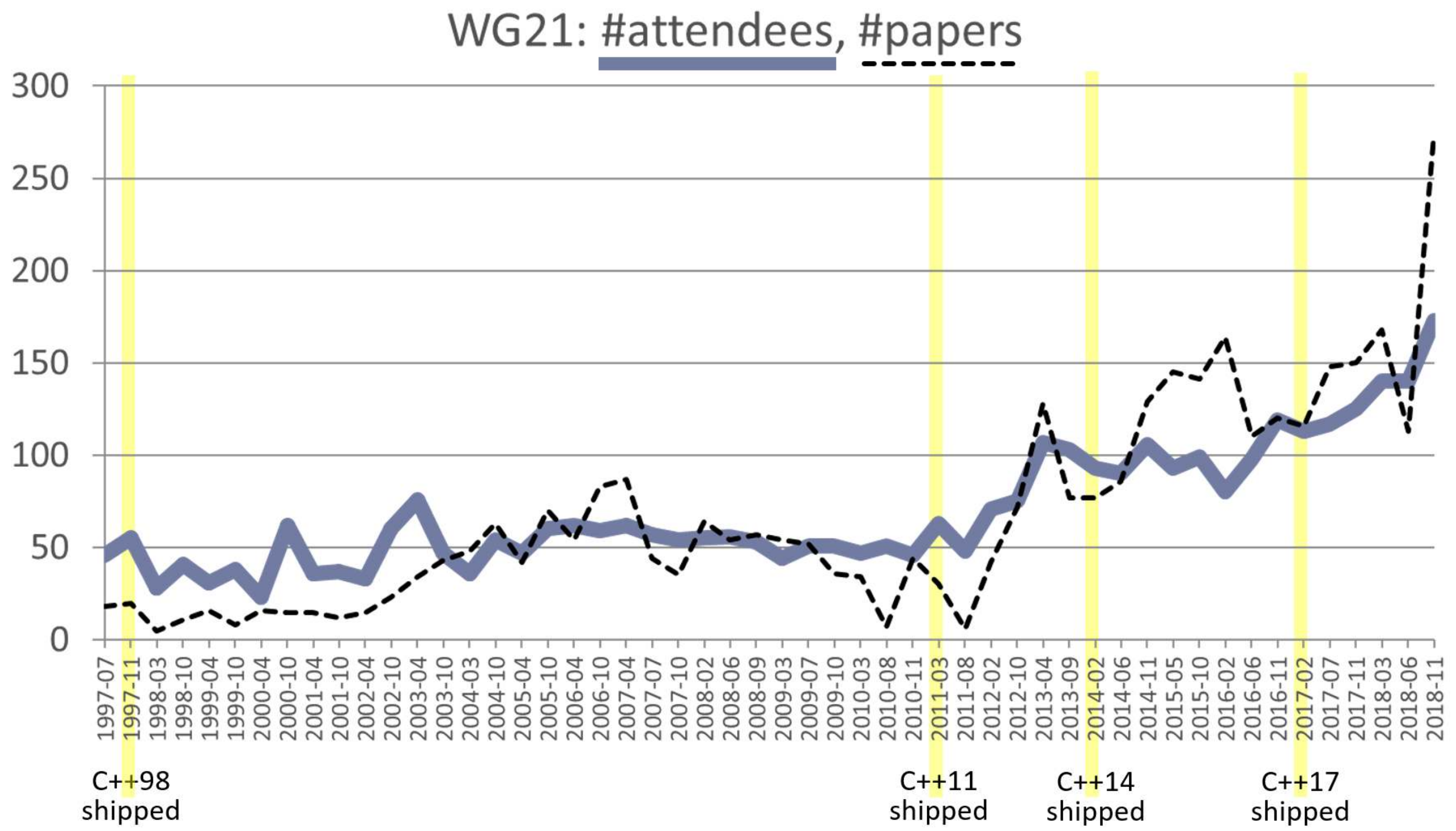
CORE LANGUAGE FUNCTIONALITY

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

C++ STANDARD LIBRARY

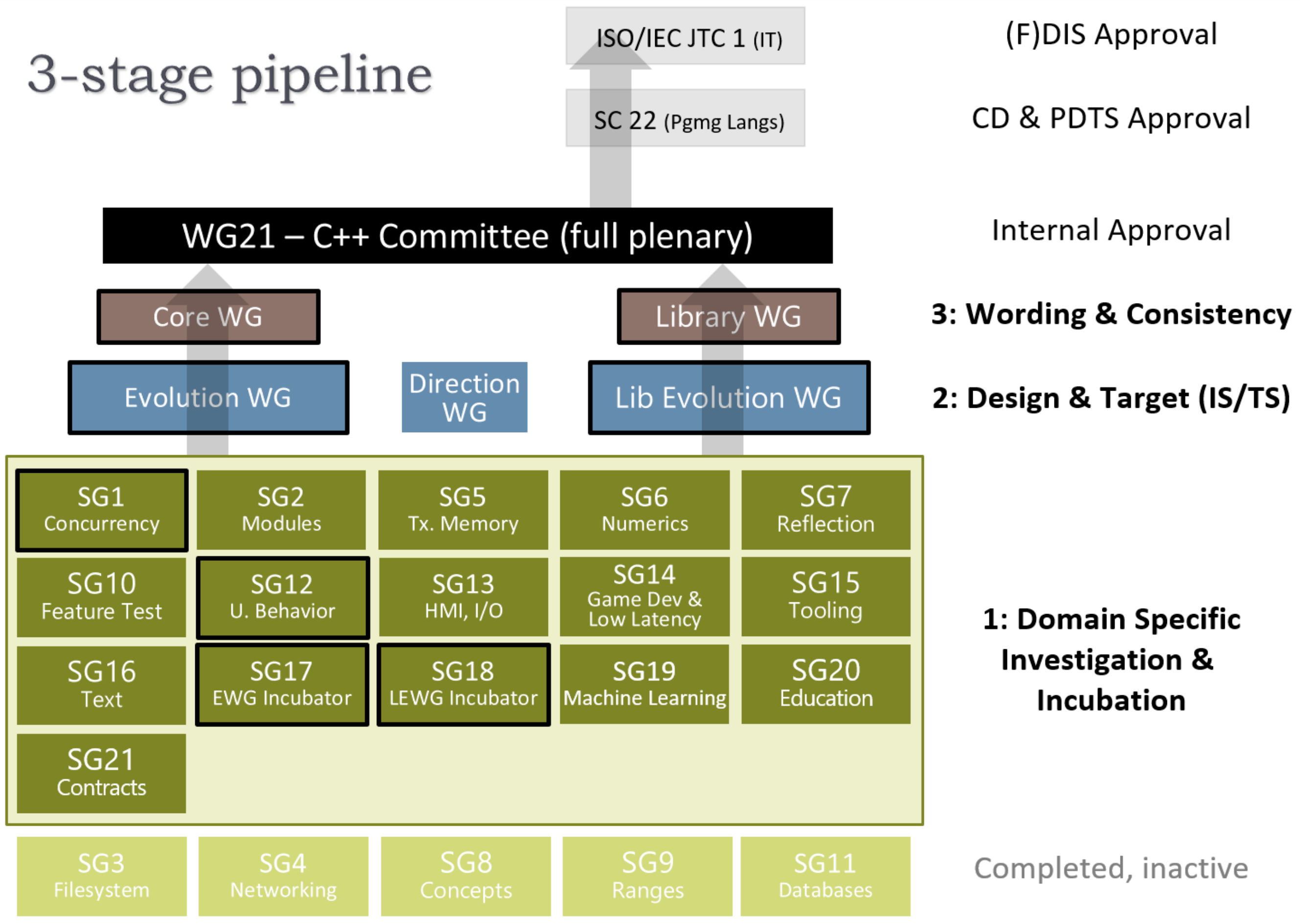
- Threading facilities
- Smart pointers and more containers
- Type traits
- Tools (regexp, chrono, random)

C++ Momentum

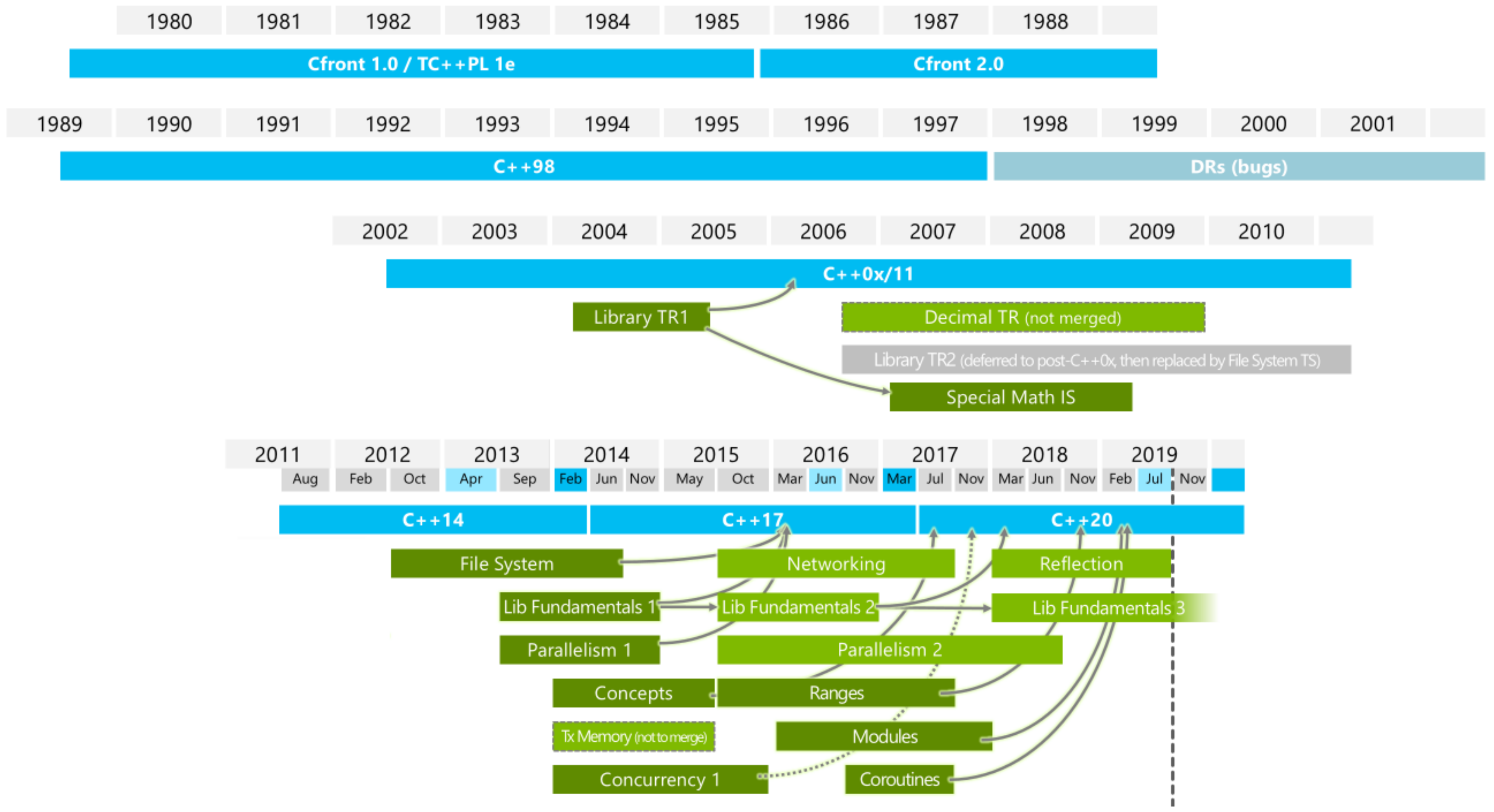


WG21: #attendees, #papers
300
250
200
150
100
50
0
1997-07
1997-11
1998-03
1998-10
1999-04
1999-10
2000-04
2000-10
2001-04
2001-10
2002-04
2002-10
2003-04
2003-10
2004-03
2004-10
2005-04
2005-10
2006-04
2006-10
2007-04
2007-07
2007-10
2008-02
2008-06
2008-09
2009-03
2009-07
2009-10
2010-03
2010-08
2010-11
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



C++ Timeline



1980
1981
1982
1983
1984
1985
1986
1987
1988
Cfront 1.0 / TC++PL 1e
Cfront 2.0
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
C++98
DRs (bugs)
2002
2003
2004
2005
2006
2007
2008
2009
2010
C++0x/11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS
2011
2012
2013
2014
2015
2016
2017
2018
2019
Aug
Feb
Oct
Apr
Sep
Feb
Jun
Nov
May
Oct
Mar
Jun
Nov
Mar
Jul
Nov
Mar
Jun
Nov
Feb
Jul
Nov
C++14
C++17
C++20
File System
Networking
Reflection
Lib Fundamentals 1
Lib Fundamentals 2
Lib Fundamentals 3
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (not to merge)
Modules
Concurrency 1
Coroutines

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?

10 REAL-LIFE CASES

Case #1: 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...);
```

Case #1: 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)
// ...
```

Case #1: Variable number of function parameters

C++11

```cpp
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)...);
}
```

Case #1: Variable number of function parameters

C++17

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

Case #2: Time handing

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

Case #2: Time handing

C++98

```cpp
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();
  }
}
```

Case #2: 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);
```

Case #2: Time handing

C++98

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

```cpp
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!

Case #2: Time handing

C++11

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

Case #2: 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();
}
```

Case #2: 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));
```

Case #2: Time handing

C++14

```
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 + 300ms);
```

Case #3: Date handling

What day of the week is July 4, 2001?

Case #3: Date handling

C++98

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

```cpp
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';
```

Case #3: Date handling

C++20

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

Case #3: Date handling

C++20

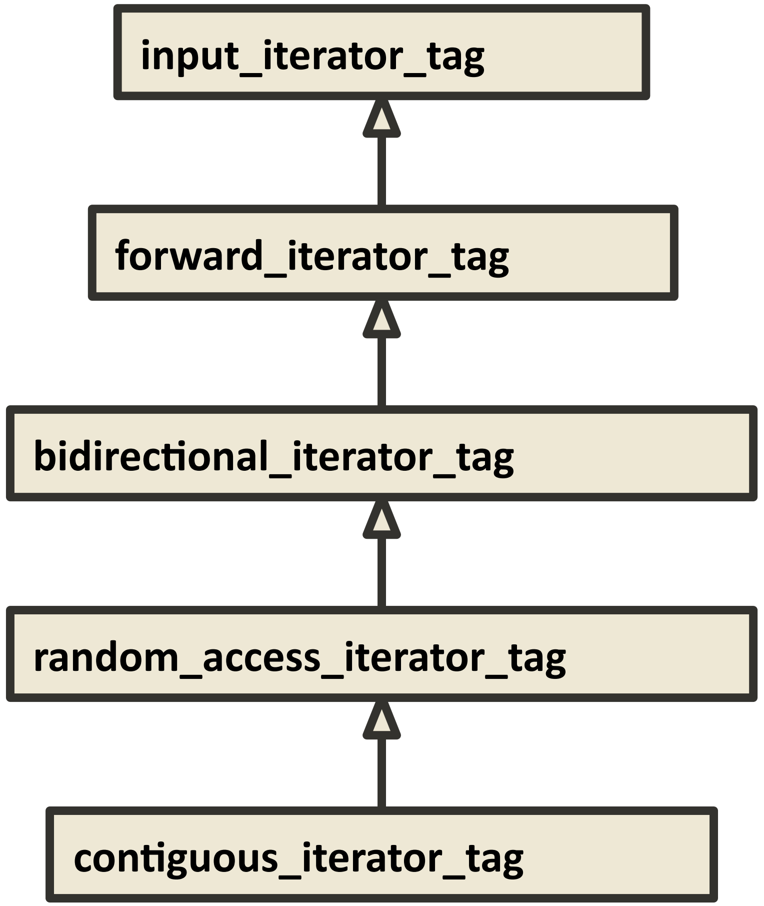
```
using namespace std::chrono;
```

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

Case #4: Compile-time dispatch

Implement efficient input iterator advance algorithm.

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



Case #4: 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());
}
```

Case #4: 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;
}
```

Case #4: 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;
}
```

Case #4: 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;
}
```

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

Case #4: 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;
}
```

Case #4: 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;
  }
}
```

Case #4: 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;
  }
}
```

Case #5: 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.

Case #5: Compile-time calculations

C++98

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

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

Case #5: Compile-time calculations

C++98

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

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

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

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

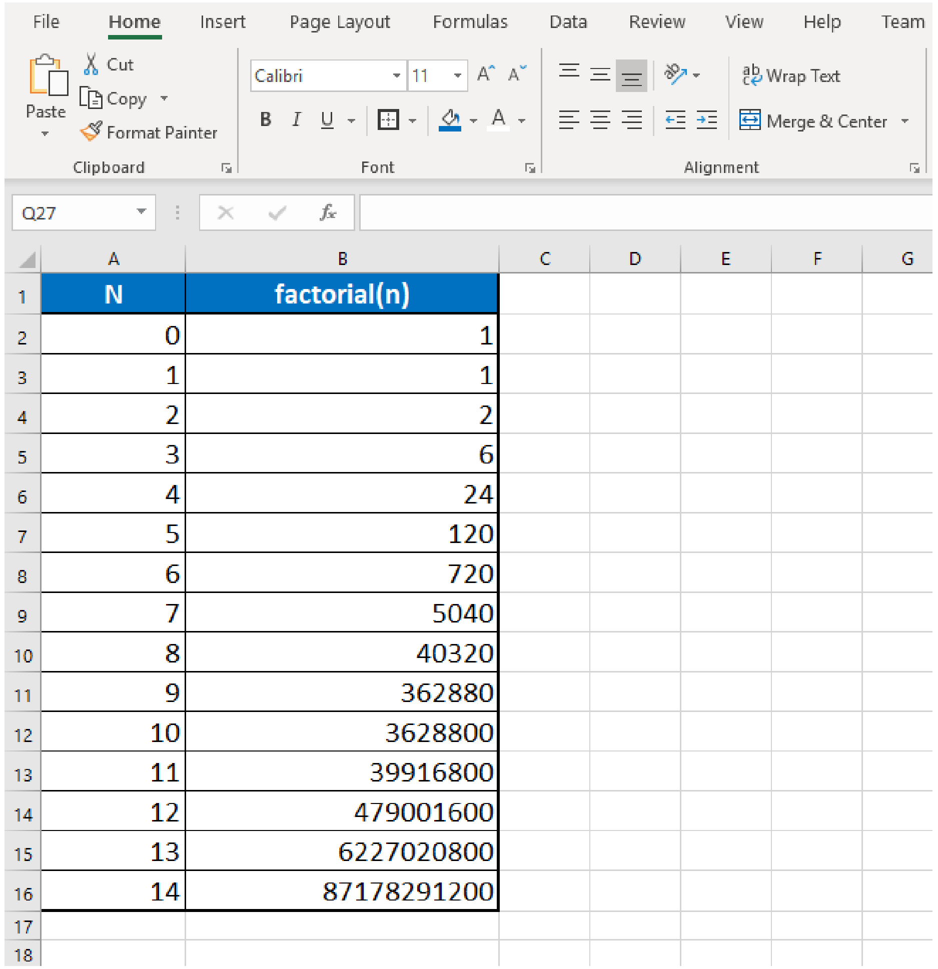
```
int array[Factorial<4>::value];
```

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

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

Case #5: 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 ;-)

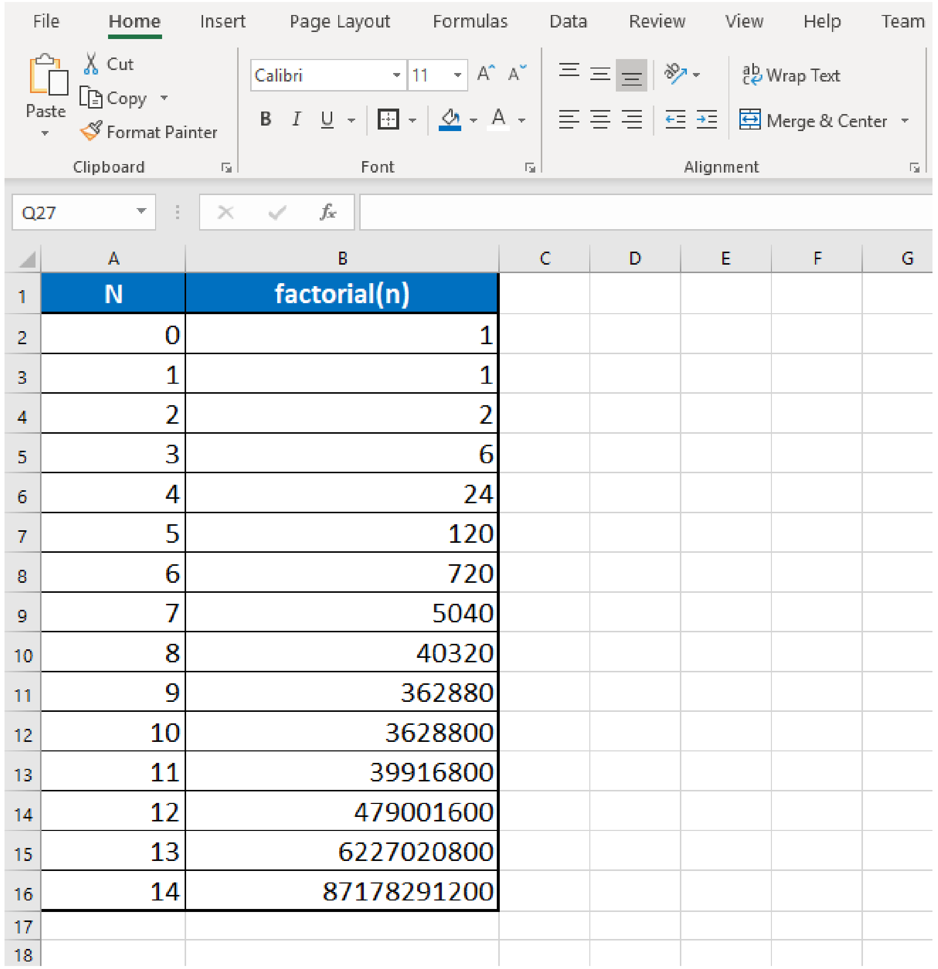


N	factorial(n)
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600
13	6227020800
14	87178291200

Case #5: 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

Case #5: Compile-time calculations

C++11

```cpp
constexpr int factorial(int n)
{
  return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

```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
```

Case #5: Compile-time calculations

C++11

```
constexpr int factorial(int n)
{
  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
```

C++14

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

```
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
```

Case #5: Compile-time calculations

C++17

```
constexpr int factorial(int n)
{
  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
```

Case #5: Compile-time calculations

C++17

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

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

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

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

C++20

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

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

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

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

Case #6: 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;
}
```

Case #6: 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);
}
```

Case #6: 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.

Case #7: 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);
```

Case #7: 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());
}
```

Case #7: 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());
}
```

Case #7: 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);
```

Case #7: 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; };
};
```

Case #7: 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);
```

Case #7: 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; };
```

Case #7: 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; };
```

Case #7: 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; };
```

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

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

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

Case #7: Evolution of algorithms

C++20

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

Case #7: 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)
{
  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?

Case #7: New family of lifetime issues

```
orders get_orders();
```

Case #7: New family of lifetime issues

```
orders get_orders();
```

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

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

Case #7: Safety included

```
orders get_orders();
```

```
template<std::ranges::ForwardRange R>
auto get_order(R&& o, client_id c)
{
  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);
```

Case #7: Safety included

```
orders get_orders();
```

```
template<std::ranges::ForwardRange R>
auto get_order(R&& o, client_id c)
{
  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);
      |       ^~~
```

Case #7: Safety included

```
orders get_orders();
```

```
template<std::ranges::ForwardRange R>
auto get_order(R&& o, client_id c)
{
  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);
```

Case #8: 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);
```

Case #8: Data range processing

C++98

```
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';
    }
  }
}
```

Case #8: Data range processing

C++98

```
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

Case #8: 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';
    }
  }
}
```

Case #8: Data range processing

C++17

```
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';
    }
  }
}
```

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

Case #8: Data range processing

C++20

```
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';
}
```

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

Case #9: Custom regular types

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

Case #9: 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); }
};
```

Case #9: 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); }
};
```

Case #10: Reflection

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

Case #10: Reflection

C++98

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

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

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

Case #10: Reflection

C++23 (???)

```
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;
}
```

```
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...