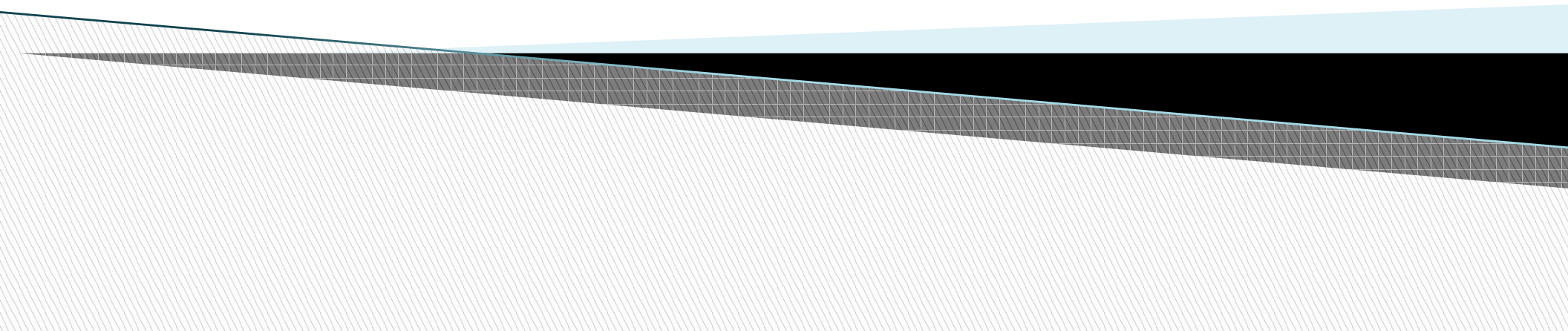
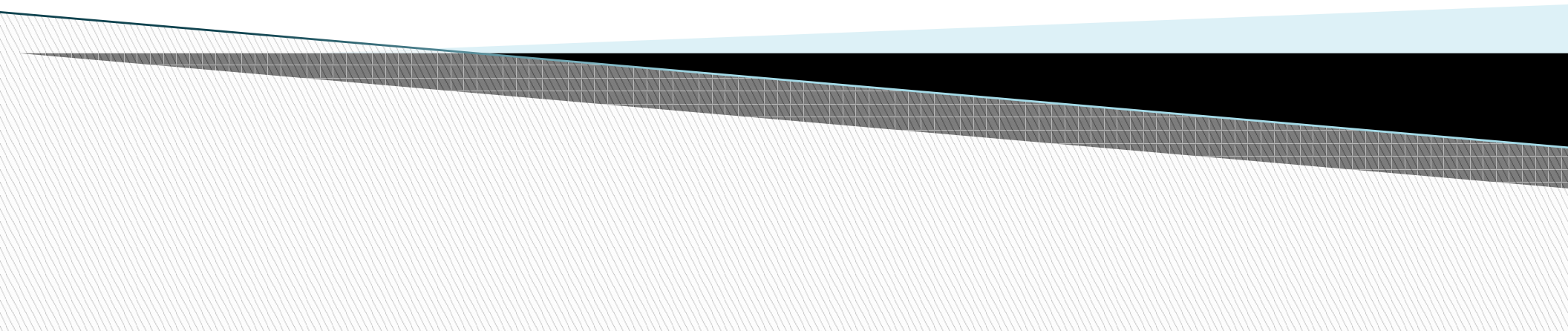
NPRG051
Pokročilé programování v C++
Advanced C++ Programming

David Bednárek
Jakub Yaghob
Filip Zavoral

http://ksi.mff.cuni.cz/lectures/NPRG051/html/nprg051.html



Types



Polymorphism in C++

- traditional view: runtime vs. compile-time

Satprem Pamudurthy
Polymorphism in C++
A Type Compatibility View
Overload Journal 141
accu.org/index.php/journals/2424

- runtime
  - dynamic polymorphism
    - inheritance + virtual functions
  - flexibility
  - runtime overhead
  - complex type hierarchies
    - enforced even for semantically unrelated types

VyVyVyS

- compile-time
  - static polymorphism
    - templates
    - generic lambdas
  - no runtime overhead
  - type specification at compile-time
  - restricted use of multiple types in one container

Type compatibility

- compatibility
  - type of expression
  - type expected in a context (i.e., parameters)
- **nominative** typing
  - variables are compatible if their declaration use the same type
    - using / typedef
- nominative subtyping
  - inheritance
    - *Liskov substitution principle*
  - relation explicitly declared
    - is-a relationship
- **structural** typing
  - compatibility by actual structure
    - no need for explicit inheritance
  - subtyping by a superset of properties
  - context-based compatibility
    - two types may be compatible in one context while incompatible in another one

```
class Dog {
  string say() { return "haf"; }
  void attack( Dog& enemy);
};
class Cat {
  string say() { return "mnau";}
  void purr();
};
```

```
f( Dog& d);
f( Cat{});
```

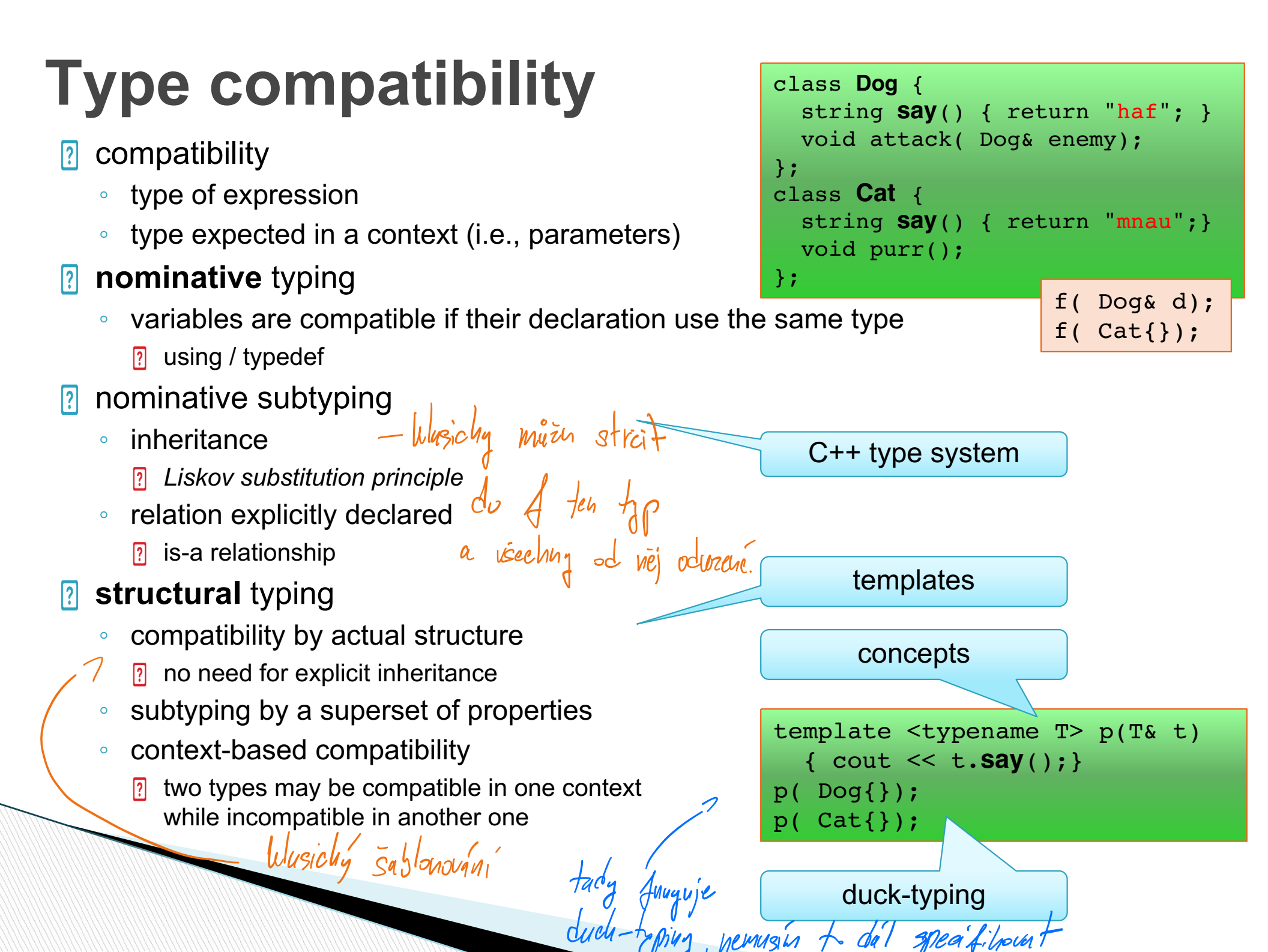
C++ type system

templates

concepts

```
template <typename T> p(T& t)
  { cout << t.say();}
p( Dog{});
p( Cat{});
```

duck-typing



Polymorphism vs. type compatibility

	compile-type polymorphism	runtime polymorphism
nominative typing		inheritance, virt fnc
structural typing	templates	

Compile time polymorphism

	compile-type polymorphism	runtime polymorphism
nominative typing	**overloaded functions**	inheritance, virt fnc
structural typing	templates	

- compile-time nominative typing
  - selection of implementation based on static types

- ad-hoc static polymorphism
  - overloaded functions
  - nominative compatibility or implicit convertibility
  - possible code duplication when applying to distinct types

```
int add( int x, int y);
double add( double x, double y);
```

```
{ return x + y; }
```

  - parametric polymorphism ⇝ templates ⇝ structural typing

```
template< typename T> ....
{ return x + y; }
```

Baarle-Nassau

Baarle - Nassau

Runtime structural typing

	compile-type polymorphism	runtime polymorphism
nominative typing	overloaded functions	inheritance, virt fnc
structural typing	templates	**type erasure**

- **structural** compatibility of **runtime** types
  - **type erasure**
    - adapter implemented using **structural** properties of the adapted object
  - polymorphic functionality of **unrelated** types
    - no need of type hierarchies!

- used in std::C++
  - std::function
  - std::any, std::variant

C++17

tohle už známe z C#!

unrelated

```
class Dog { auto&& say() { return "haf"s; }};
class Cat { auto&& say() { return "mnau"s;}};

vector< Animal> zoo;
zoo.emplace_back( Dog{});
zoo.emplace_back( Cat{});
for (auto&& a : zoo)
  cout << a.say();
```

what is Animal?

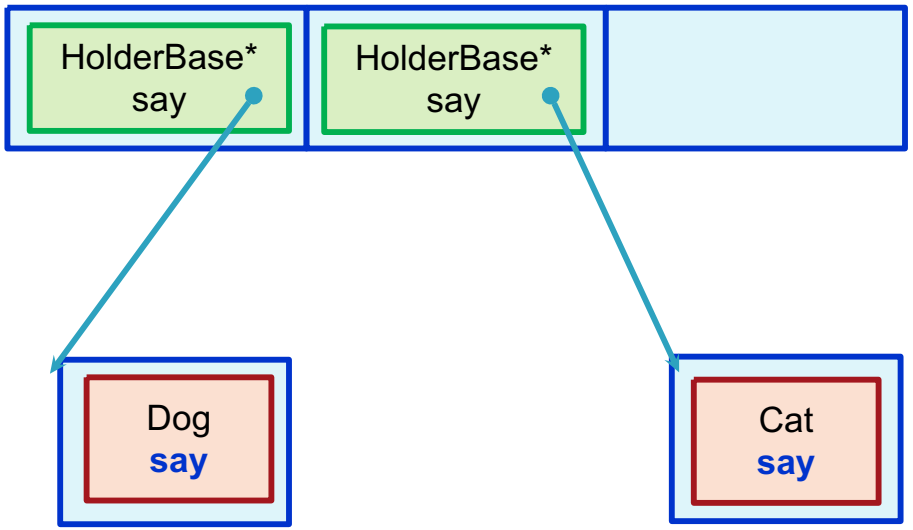
polymorphic container

common property

Type erasure

Rozhraní Dog ani Cat nedědí od Animal, takže si musím vytvořit Holder, který bude referencovat jednotlivá zvířata, zatímco ve vektoru je HolderBase

`vector< Animal>`



`Holder<Dog>:HolderBase`

`Holder<Cat>:HolderBase`

inheritance
runtime polymorphism

template
static typing

Type erasure

class! no template - polymorphism

```
class Animal {
  template<typename T> Animal(const T& obj)
    : pet_(make_unique<Holder<T>>(obj)) {}
  string say() { return pet_->say(); }

  struct HolderBase {
    virtual ~HolderBase() {}
    virtual string say() =0;
  };
  template<typename T> struct Holder : public HolderBase {
    Holder(const T& obj) : obj_(obj) {}
    string say() override { return obj_.say(); }
    T obj_;
  };

  unique_ptr<HolderBase> pet_;
};
```

set of constructors

delegation

interface - required common properties

not contained in Animal

```
class Dog { auto&& say() { return "haf"s; } };
class Cat { auto&& say() { return "mnau"s;} };

vector< Animal> zoo;
zoo.emplace_back( Dog{});
zoo.emplace_back( Cat{});
for (auto&& a : zoo)
  cout << a.say();
```

common structural property

container of unrelated types

haf mnau

any

- container for values of any type
  - must be **CopyConstructible**
  - internally uses type erasure
  - check of value presence
    - returns **std::type_info**
    - **typeid(void)** for empty value

```
any x;
x = "hello";
x = 1;
if( x.has_value())
  cout << any_cast<int>( x);
if( x.type() == typeid( string))
  ....
x.reset();

int *p = any_cast<int>(&x);  // !!
```

- **any_cast** - type-safe access
  - **bad_any_cast** if not compatible
    - runtime checking
  - pointer access possible
    - confusing syntax

```
vector<any> y{ 1, 2.9, "ahoj" };
cout << any_cast<int>(y[0]);
cout << any_cast<double>(y[1]);
cout << any_cast<string>(y[2]);
```

throw bad_any_cast

- more methods
  - emplace, swap, make_any

Je to víc resource-hungry a nepříjemné na práci

variant

- type-safe union
  - value of one of its alternative types
    - references and arrays not allowed
    - dynamic allocation not allowed
  - access by index or type
  - 'no variant': **variant<monostate>**
  - **bad_variant_access**
  - conditional pointer access
- index
  - 0-based index of current alternative
- compile-time methods:
  - variant_size, variant_alternative
- more methods:
  - emplace, swap, operator <=>

```
variant<int, float, string> v, w;
v = 12;
auto x = get<int>(v);
v = "abcd";
auto y = get<2>(v);
w = get<string>(v);
w = v;

cout << v.index(); // 2
if( holds_alternative<string>(v))
  ....

if( auto pv = get_if<int>(&v))
  cout << *pv;
else
  cout << "not an integer";
```

int*
null on error

variant & visit

- std::visit
  - polymorphic visitor
  - parameter: any **Callable**
    - overloaded function, functor, lambda
  - each type (value) is **dynamically** dispatched to the matching overload
    - runtime!

```
for( auto&& v : vec)
  cout << v;
```

overload
compile time

```
using myvar = variant<int,  double, string>;
vector< myvar> vec{ 1, 2.1, "tri" };
for( auto&& v : vec) {
  visit([](auto&& arg) { cout << arg; }, v);
}
```

polymorphic code
or overloads

must accept
all variants

auto lambda

must accept every existing type

```
  myvar w = visit([] (auto&& arg) -> myvar
    { return arg + arg; }, v);
```

return another variant
common idiom

- C++17
  - std::variant
- C++26/29 ?
  - language variant, pattern matching *later*

variant, visit & overload

```
variant< int, float, string> ifs { 3.14};
struct myVisitor {
    void operator()(int)    const { .... }
    void operator()(float)  const { .... }
    void operator()(string) const { .... }
};
visit( myVisitor(), ifs);
```

overloaded code for each type

```
variant< int, float, string> ifs { 3.14};
visit( overload {
  [](int)    { .... },
  [](float)  { .... },
  [](string) { .... },
  [](auto&&) { "default" },
  }, ifs );
```

3x C++17 → 2x C++17, 1x C++20, 1x C++26

- C++17/20 features that compose the pattern
  - extension to aggregate initialization
  - pack expansions in using declarations
  - CTAD, custom template argument deduction guides
  - CTAD and aggregates extension [C++20]

struct that inherits from several lambdas and uses their op()

```
template<class... Ts> struct overload : Ts... { using Ts::operator()... ; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;
```

overload

- pack expansions in using declarations
  - using
    - direct access to a method in a base class
    - overloading
- variadic templates
  - C++14 - recursive syntax
  - C++17 - pack expansions

```
struct b1 { void f(int); };
struct b2 { void f(string); };
struct derived : public b1, b2 {
  using b1::f;
  using b2::f;
};
derived d;
d.f('1');     // b1::f(int)
```

f to the scope of derived

request for member f is ambiguous

f must be in the same scope

```
template <typename T, typename... Ts>
struct ovld : T, ovld<Ts...> {
  using T::operator();
  using ovld<Ts...>::operator();
};
template <typename T>
struct ovld<T> : T {
  using T::operator();
};
```

C++14

```
template <typename... Ts>
struct ovld : Ts... {
  using Ts::operator()...;
};
```

C++17

overload

- extension to aggregate initialization
  - initialization of derived aggregates
    - no need for explicit constructors
  - aggregate - simple array / struct / class

```
struct b1 { int x1; };
struct b2 { int x2; };
struct derived : b1, b2 {};
derived d { 1, 2};
```

```
template <class F1, class F2>
struct overload2 : F1, F2 {
  overload2(F1 const& f1, F2 const& f2) : F1{f1}, F2{f2}
{}
  using F1::operator();
  using F2::operator();
};
```

overload resolution set
in the derived scope

aggregate initialization
no need for
explicit constructors

```
template <class ...Fs>
struct overload : Fs... {
  overload(Fs const&... fs) : Fs{fs}... {}
  using Fs::operator()...;
};
```

pack expansion

overload

C++17

- C++17
  - CTAD
  - custom template argument deduction guides
    - types of initialization expressions ⇝ template parameters

```
overload {
  [](int i)   {},
  [](float f) {},
};
```

error: cannot deduce template arguments

  - C++17 CTAD limitations

```
template<class... Ts> overload(Ts...) -> overload<Ts...>;
```

types of init expr
-> template parameters

overload

C++20

- C++20
  - CTAD and aggregates extension

```
template <typename T, typename U, typename V>
struct Triple { T t; U u; V v; };
Triple t{ 10.0f, 90, "hello"s};
```

C++20 CTAD and aggregates

- overload
  - custom deduction guide no more needed

tohle je definice

```
template<class... Ts> struct overload : Ts... { using Ts::operator()... ; };
```

- visit as a member of variant [C++26]

tohle je využití toho stmctu

```
variant< int, float, string> ifs {
3.14};
ifs.visit( overload {
  [](int)    { .... },
  [](float)  { .... },
  [](string) { .... },
  [](auto&&) { "default" },
});
```

Bartek Filipek:
2 Lines Of Code and 3 C++17 Features - The Overload Pattern
www.bfilipek.com/2019/02/2lines3featuresoverload.html

variant & multiple dispatch

- single dispatch
  - virtual class::method, object->method
- double/multiple dynamic dispatch
  - *virtual [class1,class2]::m, [obj1,obj2]->m*
- visit, variant & overload
  - visit can accept more variants
  - default overload using generic lambdas (auto)

```
class Ufo;
class Blaster : Ufo;
class Bumper : Ufo;
class Neutrino : Ufo;
vector< Ufo*> Army;
```
[Bumper,Blaster]::
crash(){..}

```
template <class Visitor, class... Variants>
constexpr RetT visit( Visitor&& vis, Variants&&... vars
);
```

```
variant<int, double, string> v1, v2;
visit( overload{
  [](int a1, int a2) {..},
  [](int a1, double a2) {..},
  [](int a1, const string& a2) {..},
  [](double a1, int a2) {..},
  [](double a1, double a2) {..},
  [](double a1, const string& a2) {..},
  [](const string& a1, auto&& a2) {..},
}, v1, v2);
```

Bartek Filipek:
How To Use std::visit With
Multiple Variants and Parameters
https://www.cppstories.com/
2018/09/visit-variants/

half-default, že pokud je první string,
tak ať je druhé cokoliv, pravděpodobně

polymorphism - inheritance vs. variant

- variant
  - faster
    - no dynamic allocation
  - shorter
  - value semantics
  - structural typing
    - unrelated classes
  - each operation requires a visitor

inh

```
class base {
public:
  virtual ~base() = default;
  virtual void foo() = 0;
};
class x : public base {
public: void foo() override;
};
class y : public base {
public: void foo() override;
};
unique_ptr<base> b = make_unique<x>();
b->foo();
```

var

```
struct x { void foo(); };
struct y { void foo(); };
variant<x, y> b;
b = x{};
visit([](auto&& v){ v.foo(); }, b);
```

variant	inheritance
structural typing	nominative typing
value semantics	pointer semantics
faster	slower
no dynamic alloc	dynamic alloc
contiguous memory	scattered memory
simple	complex
single-level	multi-level

Bartek Filipek:
Runtime Polymorphism with std::variant
www.cppstories.com/2020/04/variant-virtual-polymorphism.html

variant use cases

- error handling
  - ⇝ expected [C++23]

```
variant<string, ErrorCode> fnc();
```

- multiple return types
  - $ax^2+bx+c=0$

```
using EqRoots = variant<complex, double>;
EqRoots Roots(double a, double b, double c) {
  return cond ? complex{..} : (-1*b)/(2*a);
```

- parsing
  - parameters, config file, ...

```
using Arg = variant<bool, int, string>;
map<string, Arg> mParsedArgs;
```

- multiple dispatch
- polymorphism of unrelated types

```
vector< variant< Tria, Poly, Sph>> objects;
auto CallR = [](auto& obj) { obj.Render(); };
for( auto& obj : objects)
  visit( CallR, obj);
```

optional

- a value may or may not be present
  - useful for functions that may fail
    - or NULL-able data - db
  - default conversion to **bool**
  - **bad_optional_access**
  - convenient **value-or**-whatever access
- pointer and reference
  - *** ->**
  - behavior undefined
    if value is not present
- more methods
  - **emplace, swap, reset, make_optional**
  - **operator<=>**
- use cases
  - *"maybe I have an object, maybe I don't"*
  - database NULL, parsing, input, ...
  - deferred initialization
    - e.g., required default constructibility

```
optional<string> f() {
  return 🎲 ? "Godzilla" : {};
}
auto x = f();
if( x)      // ≡ x.has_value()
  cout << x.value();
cout << x.value_or("empty");
```

```
optional<int> y{ 1};
cout << *y;
*y = 2;
optional<string> z{ "ahoj"};
z->size() ....
```

no more magic values

```
0, -1, (void*)0, EOF,
0xFFFFFFFF, nullptr,
container.end(),
string::npos, ...
```

optional - monadic operations

C++23

- chaining optional functions
  - functional style programming

```
optional<string> get_s();
optional<int> convert_i(optional<string> so);
```

```
auto so = get_s();
auto io = so ? convert_i( so) : nullopt;
```

A monad is a monoid in the category of endofunctors

applying the function to the previous result

```
auto io = get_s()
   .and_then( convert_i)
   .transform( [](int x){....})
   .or_else( optional( -1));
```

and_then	returns the result of the given function on the contained value if it exists, or an empty optional otherwise
transform	returns an optional containing the transformed contained value if it exists, or an empty optional otherwise
or_else	returns the optional itself if it contains a value, or the result of the given function otherwise

value_or

expected

C++23

- how to return a value or an error
  - optional - no error code possible
  - additional output parameter
  - tuple / structured binding
  - variant - complex syntax
  - exceptions - too heavyweight
- expected ☻
  - Rust: result, Haskell: either
  - convenient syntax
    - **value error**
    - *** -> bool value_or error swap**
  - monadic operators
    - **and_then transform or_else**

```
expected<double, errc>
safe_div(double i, double j)
{ if (j==0)
    return unexpected(div_by_zero);
  else
    return i / j;
}
auto val = safe_div( 1,0);
if( val)
  cout << *val;  // val.value()
else
  .... val.error()
```

```
expected<image,fail_reason> get_cute_cat( image& img) {
  return crop_to_cat(img)
    .and_then(add_bow_tie)
    .and_then(make_eyes_sparkle);
}
```

expected + variant / visit

C++23

```
enum Errc { LexE, SyntaxE
  , RuntimeE
};

expected< string, Errc> parse();

auto s = parse();
if( ! s) {
  switch( s.error()) {
    case LexE: ....
    case SyntaxE: ....
    default: ....
  }
}
```

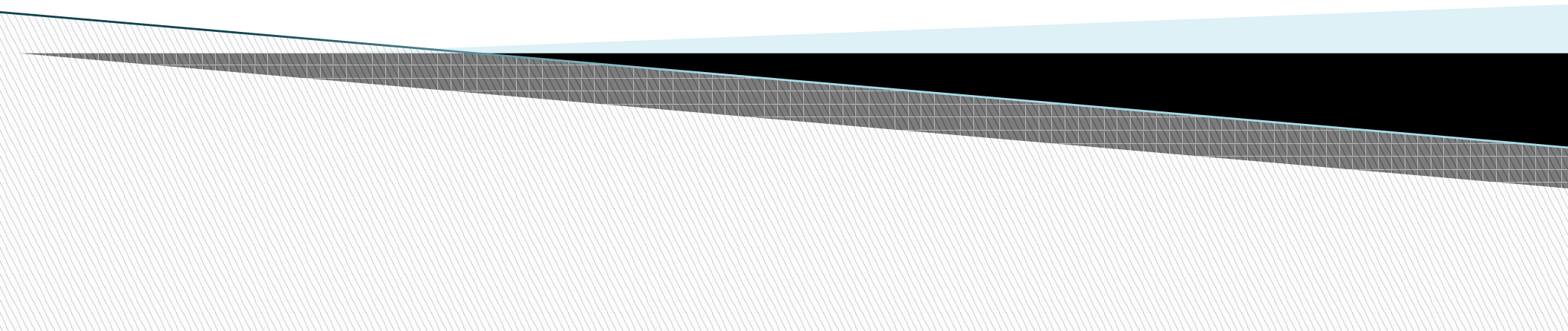
foooo

```
using Errc = variant
  <LexE, SyntaxE, RunmtimeE>;


expected< string, Errc> parse();

auto s = parse();
if( ! s) {
  visit( overload{
    [](LexE& err) {},
    [](SyntaxE& err) {},
  }, s.error());
}
```

Error: visit requires the visitor to be exhaustive

- always prefer compile-time checking

Ranges



Ranges

C++20⇝23

- stl
  - iterator based algorithms - verbosity

```
sort( v.begin(), v.end());
set_difference( v2.begin(), v2.end(), v3.begin(), v3.end(), back_inserter(v4));
```

  - no orthogonal composition

```
copy_if( input.begin(), input.end(), back_inserter(output), p);
transform( input.begin(), input.end(), back_inserter(output), f );
transform_if
```

efficiency - copy

- ranges
  - (it,it), (it,count), (it,predicate)
    - encapsulation of the limits, enhanced safety and readability
  - std:: containers, generators, virtual containers, ...
  - composability, lazy evaluation

constrained algorithm

```
ranges::sort( v);
v | views::filter([](){}) | views::transform([](){});
```

range adaptor

pipeline-like composition

Evolution of range based loops

- Using indices
  - danger of out-of-bounds UBs

```
vector<int> xs({ 1, 2, 3, 4, 5 });

for( size_t i=0; i<xs.size(); ++i)
    print("{} ", xs[i]);
```

- Using iterators
  - better
  - but iterators are hidden pointers

```
for( auto it = xs.begin();
  it != xs.end(); ++it)
    print("{} ", *it );
```

- C++11 range-based for loop
  - only the entire container
  - no other possibilities like reverse, etc.

```
for(auto&& x : xs)
    print("{} ", x);
```

- C++20 ranges
  - many possible views
  - lazy evaluation
  - piping operator

```
for(auto&& x : xs | views::reverse)
    print("{} ", x);
```

Ranges and range adaptors

- terminology
  - **range** ≈ something that can be iterated
  - **range algorithm** ≈ algorithm that takes range
  - **view** ≈ cheap to copy lazy range
  - **range adaptor** ≈ make a range into a view
- range adaptor syntax equivalence

```
adaptor( range, args...)
adaptor( args...) (range)
range | adaptor( args...)
```

```
take_view<
  transform_view<
    filter_view<
      iota_view<int, int>,
      lambda [](int n)->bool
    >,
    lambda [](int n)->int
  >
>
```

```
#include <ranges>

void f( const string& s) { cout << s; }

auto r = view::iota( 1, 1000)
  | view::filter( [](int n) { f( "o"); return n%2 == 0; })
  | view::transform( [](int n) { f( "x"); return n*2; })
  | view::take( 2)
  ;
for( auto&& i : r)
  cout << i;
```

sentinel

not a container

48

ooxoox48

oox4oox8

pipeline-like composition

lazy evaluation

C++17 vs. C++20

- reverse access to odd items

```
for_each( v.crbegin(), v.crend(),
   [](auto x) {
      if(x % 2 == 1)  f(x);
   }
);
```

```
for( auto&& x : v | views::reverse
   | views::filter(
     [](auto x){ return x%2==1; }))
  f(x);
```

- word count

```
istringstream iss(text);
vector<string> words(

istream_iterator<string>{iss},
    istream_iterator<string>{});
auto count = words.size();
```

copy

range ≈ (it,it)

```
auto count = ranges::distance(
  views::split( text, " "sv));
```

- sorted container without two greatest and two smallest values

```
auto first = v.begin();
advance( first, 2);
auto last = first;
advance( last, v.size() - 2);
v.erase( last, v.end());
v.erase( v.begin(), first);
```

```
v | views::drop(2)
  | views::take( v.size()-4);
```

Ranges - eager and lazy evaluation

```
int sum_of_squares( int count) {
  vector<int> numbers( count);
  iota( numbers.begin(), numbers.end(), 1);
  transform( numbers.begin(), numbers.end(), numbers.begin(), [](int x){ return x*x;});
  return accumulate( numbers.begin(), numbers.end(), 0);
}
```

classic STL

```
int sum_of_squares( int count) {
  vector<int> numbers( count);
  iota( numbers.begin(), numbers.end(), 1);
  ranges::transform( numbers, numbers.begin(), [](int x){ return x*x;});
  return accumulate( numbers.begin(), numbers.end(), 0);
}
```

evaluated immediately

```
int sum_of_squares( int count) {
  auto v = views::transform( views::iota(1, count+1),
    [](int x) { return x*x; } );
  return accumulate( v.begin(), v.end(), 0);
}
```

lazy evaluation

efficiency

evaluation at the point of dereference

```
int sum_of_squares( int count) {
  auto v = views::iota( 1, count+1)
    | views::transform( [](int x){ return x*x;});
    | views::common;
  return accumulate( v.begin(), v.end(), 0);
}
```

wrapper not needed *in this case*

C++23 fold functions

std:: algorithms require equal types

Creating a range-based algorithm

```
template <typename I, typename S, typename T>
T accumulate( I first, S last, T init) {
  while( first != last)
    init = move(init) + *first++;
  return init;
}
```

```
accumulate( v);
accumulate(
  v.begin(),
  v.end());
```

acc(1, "x", true);

```
template <typename I, typename S, typename T, typename Op=plus<>>
T accumulate( I first, S last, T init, Op op=Op{}) {
  while( first != last)
    init = op( move( init), *first++);
  return init;
}
```

user-defined operator

std:: concepts

```
template <input_iterator I, sentinel_for<I> S,
  typename T=iter_value_t<I>, typename Op=plus<>>
T accumulate( I first, S last, T init=T{}, Op op=Op{})
```

range overload

```
template <r::input_range R, typename T=r::range_value_t<R>, typename
Op=plus<>>
T accumulate( R&& rng, T init=T{}, Op op=Op{}) {
  return accumulate( r::begin(rng), r::end(rng), move(init), move(op));
}
```

```
accumulate( iota( 1, count+1) | transform( [](int x){ return x*x;}));
```

Projections

- C++20 rangified algorithms take extra optional parameter - projection
  - modification of the data just before use
  - default ≈ identity, common non-default ≈ lambda

```
template <input_iterator I, sentinel_for<I> S, typename T=iter_value_t<I>,
  typename Op=plus<>, Proj=identity>
T accumulate( I first, S last, T init=T{}, Op op=Op{}, Proj proj=Proj{}) {
  while( first != last)
    init = invoke( op, move(init), invoke( proj, *first++) );
  return init;
}
```

```
template <r::input_range R, typename T=r::range_value_t<R>,
  typename Op=plus<>, Proj=identity>
T accumulate( R&& rng, T init=T{}, Op op=Op{}, Proj proj=Proj{}) {
  return accumulate( r::begin(rng), r::end(rng), move(init), move(op),
    move(proj));
}
```

```
accumulate( iota( 1, count+1), {}, {}, [](int x) { return x*x; });
```

C++17

```
int sum_of_squares( int count) {
  vector<int> numbers( count);
  iota( numbers.begin(), numbers.end(), 1);
  transform( numbers.begin(), numbers.end(), numbers.begin(), [](int x){ return x*x;});
  return accumulate( numbers.begin(), numbers.end(), 0);
}
```

Projections & invoke

```
pair<int, string_view> pairs[] = { {2, "foo"}, {1, "bar"}, {0, "baz"} };
ranges::sort( pairs, less{}, &pair<int,string_view>::first);
ranges::sort( pairs, less{}, [](auto const& p) { return p.first; });
```

```
struct Box { string name; int w, h, d;
  constexpr int volume() const { return w*h*d; }
};

ranges::sort( box, {}, &Box::name);
ranges::sort( box, {}, &Box::volume);
```

r-t specified sort criteria

- projections in non-std::library functions

```
void PrintEx( const ranges::range auto& container, auto proj) {
  for( auto&& elem : container)
    cout << invoke( proj, elem);
};

vector< pair<int, char>> pairs { {0, 'A'}, {1, 'B'}, {2, 'C'}};
PrintEx( pairs, &pair<int, char>::first);
PrintEx( pairs, &pair<int, char>::second);
```

fold

C++23

- fold
  - generalization of accumulate
  - takes an algorithm and optional initial value
  - **fold_left** / **fold_right**
    - forward / backward direction
    - f(f(f(init, x_1), x_2), ..., x_n)
  - **fold_left_first** / **fold_right_last**
    - first / last element as an initial value
  - supports projections

```
constexpr auto ranges::fold_left( R&&, T init, F f, Proj proj={});

vector v = { 25, 75};
auto result = fold_left( v, 0, std::plus());    //100
```

Fibonacci

- Fibonacci sequence: 1 1 2 3 5 8 13 ....
  - first **n** elements divisible by **k**

C++17

```
vector<size_t> fd( int k, int n) {
   vector<size_t> v;
   size_t val = 1;
   size_t next_val = 1;
   while (n > 0) {
      if (val % k == 0) {
         --n;
         v.push_back(val);
      }
      val = exchange( next_val, val + next_val);
   }
   return v;
}
```

- Single responsibility principle?
  1. calculating a fibonacci sequence
  2. checking divisibility
  3. counting found numbers
  4. filling a container

⇝ decomposition !
(efficiency)

Fibonacci

```
class Fib {
public:
   size_t operator()() {
      auto result = val;
      val = exchange( next_val, val + next_val);
      return result;
   }
private:
   size_t val = 1;
   size_t next_val = 1;
};
```

functor

```
vector<size_t> fd(int k, int n) {
   vector<size_t> v;
   Fib fib;
   while (n > 0) {
      auto val = fib();
      if (val % k == 0) {
         --n;
         v.push_back(val);
      }
   }
   return v;
}
```

saving the result

single responsibility

? next value
? return value

bad interface

Fibonacci

decomposition of "functor" calls

```
class Fib {
public:
  const size_t& get() { return val; }
  void next() { val = exchange(next_val, val + next_val);
}
private:
  size_t val = 1;
  size_t next_val = 1;
};
```

??

```
vector<size_t> fd(int k, int n) {
  vector<size_t> v;
  Fib fib;
  while (n > 0) {
    if( fib.get() % k == 0) {
      --n;
      v.push_back( fib.get());
    }
    fib.next();
  }
  return v;
}
```

Fibonacci

```
class Fib {
public:
  using value_type = size_t;
  using difference_type = ptrdiff_t;
  using iterator_category = forward_iterator_tag;
  const size_t& operator*() const { return val; }
  Fib& operator++() { val = exchange(next_val, val + next_val); return *this; }
  Fib operator++(int) { auto temp = *this; ++*this; return temp; }
  bool operator==(const Fib&) const = default;
private:
  size_t val = 1;
  size_t next_val = 1;
};
```

C++26?
precooked interface

```
static_assert(forward_iterator<Fib>);
```

the concept
input_or_output_iterator
evaluated to false

the concept
weakly_incrementable
evaluated to false

```
vector<size_t> fd(int k, int n) {
  vector<size_t> v;
  Fib fib;
  while (n > 0) {
    if( *fib % k == 0) {
      --n;
      v.push_back( *fib);
    }
    ++fib;
  }
  return v;
}
```

iterator-like
behavior

Fibonacci

```
class Fib {
public:
    using value_type = size_t;
    using difference_type = ptrdiff_t;
    using iterator_category = forward_iterator_tag;
  const size_t& operator*() const { return val; }
  Fib& operator++() { val = exchange(next_val, val + next_val); return *this; }
    Fib operator++(int) { auto temp = *this; ++*this; return temp; }
    bool operator==(const Fib&) const = default;
private:
  size_t val = 1;
  size_t next_val = 1;
};
```

```
take_view<
  remove_cv<
    filter_view<
      subrange<Fib, unreachable_sentinel_t, r::subrange_kind::unsized>,
      lambda[](auto i)->auto
    >
  >::type
>
```

```
constexpr auto fib_view =
  ranges::subrange< Fib, std::unreachable_sentinel_t>{};
```

```
auto fd(int k, int n) {
  return fib_view
    | views::filter([k](auto i) { return i % k == 0; })
    | views::take(n);
}
```

decomposition
readability
single responsibility
efficiency

Sieve of Eratosthenes

- ranges/view compatible
  - lazy evaluation
  - do not forget to set your compiler to C++23 (/std:c++latest)

```
sieve_view | views::take(n) |
to<vector>();
for( auto i : sieve_view) ....
```

- basic algorithm
  - everyone is familiar
  - no optimized algorithm needed
- efficient implementation
  - memory
    - O(current_value)
  - processor
    - each computation should fully reuse the data from previous computations

Dangling protection

```
vector<int> x = get_input();
auto it = std::min_element( x.begin(), x.end());
```

dangling iterator ?!

```
vector<int> x = get_input();
auto it = ranges::min_element( x);
```

```
auto it = ranges::min_element( get_input());
```

all ranges algorithms

```
iterator ranges::min_element( range&);
ranges::dangling ranges::min_element( range&&);
```

dereferencing compiler error

```
auto it = ranges::min_element( get_input());
cout << *it;
```

compile-time protection

```
string s = "Hello world";
auto it = string_view{s}.begin(); // dtto: span
cout << *it;                      // OK
```

borrowed range

protection not needed

- dangling protection can be disabled by specializing the **enable_borrowed_range** trait

```
template <> inline constexpr bool ranges::
enable_borrowed_range<MyStrRef> = true;
```

View definition

C++23⇝20

'new' C++23 definition, retroactively changed the 'old' C++20 definition

- view is a range that
  - its **move** operations are **constant**-time
    - don't depend on the view size
  - its **copy** operations are **constant**-time
    - if it is copyable
  - **destruction** of a **moved**-from object is **constant**-time
    - otherwise, destruction is O(n)
  - usually non-owning, lazy evaluation
  - expected to be passed by value
    - no **const&** - caching
- these properties cannot be determined by compilers
- user-defined type can be specified as a view
  - **enable_view** trait
  - or inherited from **ranges::view_base**
  - or **ranges::view_interface**

```
template <> inline constexpr bool
ranges::enable_view<MyType> = true;
```

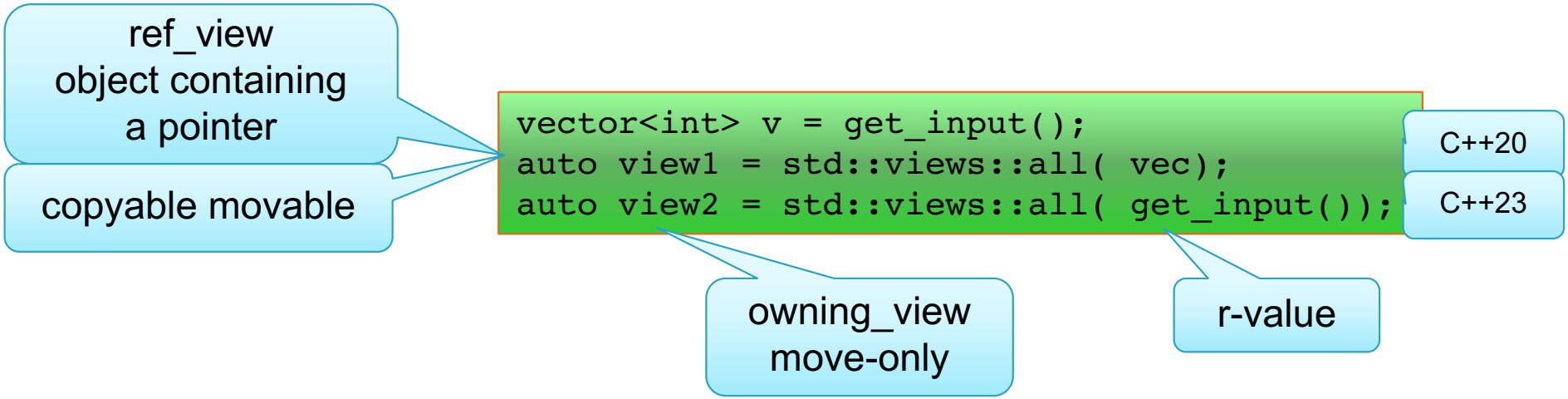
```
class MyType
  : public ranges::view_base { .. };
```

Tristan Brindle:
C++20 Ranges in Practice
www.youtube.com/watch?v=L0bhZp6HMDM

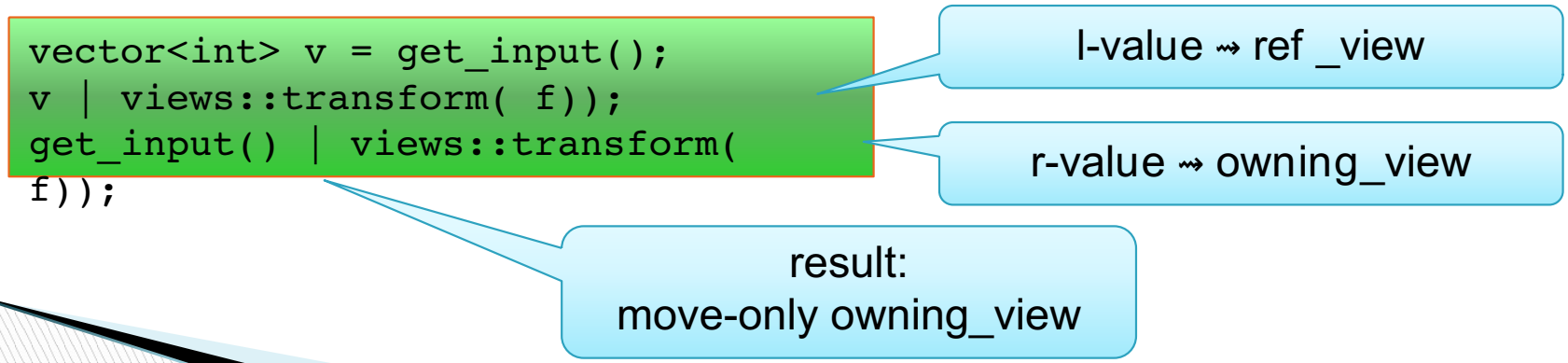
Conquering C++20 Ranges
www.youtube.com/watch?v=3MBtLeyJKg0

Conversion of ranges and views

- any movable range can be turned into a view using **std::views::all**



- view adaptors
  - implicit conversion of arguments to views
  - pipes



Range adaptors (views)

C++20

all	all elements of its range argument
filter	elements of an underlying sequence that satisfy a predicate
transform	underlying sequence after applying a transformation function to each element
take	the first N elements from another view
take_while	a range [begin(r), find_if_not(r, pred))
drop	excluding the first N elements from another view
drop_while	a range [find_if_not(r, pred), end(r)).
join	flattens a view of ranges into a view
split	splits the view into subranges; the delimiter can be a single element or a view of elements
counted	the elements of the counted range ([iterator]) i+[0, n).
common	a view of the same elements with an iterator and sentinel of the same type
reverse	a view that iterates the same elements in reverse order
elements	a view with the Nth element (tuple)
keys	a view with a value-type of the first element
values	a view with a value-type of the second element

zip zip_transform adjacent adjacent_transform join_with slide chunk as_const as_rvalue stride

C++23

trim string

- trim whitespaces
  - **" \n \t \r I love C++ \n"**

    ↓

    "I love C++"
  - no loop, if/else hell or other old-school manual error-prone hard work

static life-time function object

specialization of the std view

```
inline constexpr auto trim_front = views::drop_while(::isspace);
inline constexpr auto trim_back =
      views::reverse | trim_front | views::reverse;
inline constexpr auto trim = trim_front | trim_back;

string trim_str( const string& str) {
      return str | trim | ranges::to<string>();
}
```

no copy
lazy evaluation

materialization
copy chars

Views and range algorithms

```
#include <ranges>
namespace std {
  namespace ranges {...};
  namespace ranges::views {...};
  namespace views = ranges::views;
}
```

- miscellaneous views
  - iota repeat subrange single empty ref_view owning_view
- std:: algorithms adapted to ranges
  - namespace **ranges::**
  - for_each, count, find, copy, move, transform, remove, sort, ..., ..., ..., ...
  - **to**
- range concepts
  - range borrowed_range sized_range view
  - _range: input_ output_ forward_ bidirectional_ random_access_ contiguous_ common_ viewable_ constant_

C++23 materialization

hackingcpp.com/cpp/cheat_sheets.html

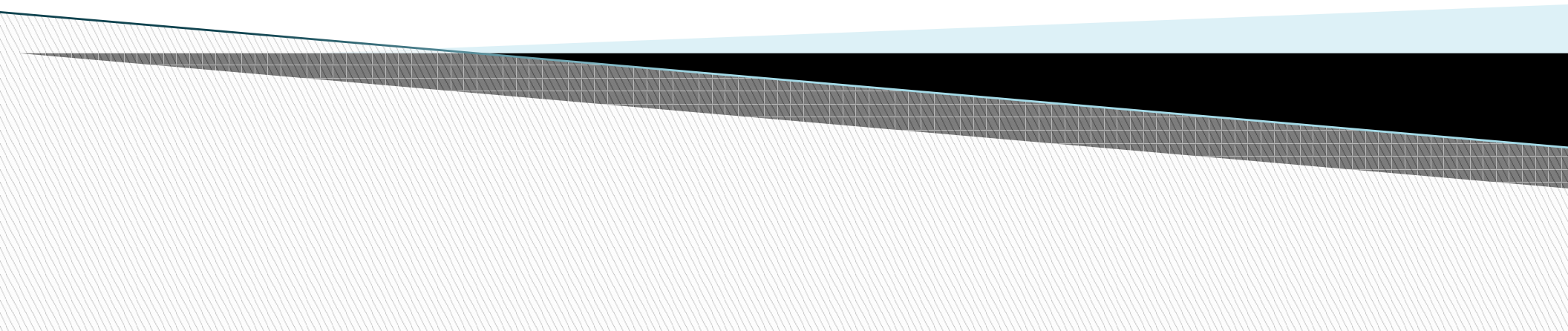
Erich Niebler: Range v.3
std:: proposal based on this library
ericniebler.github.io/range-v3

Bartek Filipek: C++20 Ranges Algorithms
www.cppstories.com/2022/
ranges-alg-part-one
ranges-alg-part-two
ranges-alg-part-three
ranges-composition

C++26 proposal

- algorithms
  - reduce, sum, product
- views
  - concat, cycle, getlines
  - drop_last, take_last, drop_exactly, take_exactly
  - split_when
  - remove, remove_if, replace, replace_if
  - transform_join
- ....

Chars & strings



char types & unicode literals

C++03	char	string	"abc"	implementationd efined
	wchar_t	wstring	L"abc"	
C++11	**char16_t**	**u16string**	**u**"abc**u**FFFF"	UTF16/UCS-2
	char32_t	**u32string**	**U**"abc**U**F0F0F0F0"	UTF32/UCS-4
C++20	**char8_t**	**u8string**	**u8**"abc\xFF\xFFFF"	UTF8

- fixed platform-independent width
- Unicode support
  - u8"" - always UTF8
    - "žhář" ≠ u8"žhář" (def ault codepage / UTF8, 5/8 bytes!)
  - UCS-2 older, fixed 2-byte
  - UTF16 newer, superset, possibly 4-byte

???

```
f = open( "C:\temp\new.txt");
```

```
('(?:[^\\']|\\.)*'|"(?:[^\\"]|\\.)*")|
```

Raw string

```
f = open( "C:\t emp\n ew.txt");
```

who **never** made such a bug ??

```
('(?:[^\\' ]|\\ .)*' |"(?:[^\\" ]|\\ .)*")|
```

\\ ? \\\\
" ?\"

- raw string literal
  - escape chars does not apply
  - all chars valid (*newline*, *tab*, *"*, ...)
  - user-defined delimiter
  - applicable to uR"", u8R"", ...

R" *dch** (*ch**) *dch** "

```
R"#( )" )#"
```

```
R"(('(?:[^\\']|\\.)*'|"(?:[^\\"]|\\.)*")|)"
```

```
"(\'(?:[^\\\\']|\\\\.)*\'|\"(?:[^\\\\"]|\\\\.)*\")|"
```

```
R""(A    \b
C)""  "\0 "  R"raw(GHI)raw";
```

⟺

```
"A\t\\b\nC\0 GHI";
```

strings using different syntax concatenated

string conversions

C++17

- many std:: functions
  - **sprintf snprintf sscanf atol strtol strstream stringstream num_put num_get to_string stoi ...**
  - locale, memory allocation, whitespaces, exceptions, ...
  - extra functionality not needed, insufficient error reporting, safety
- **hi-perf** string conversions
  - low-level
  - non-throwing, non-allocating, no locale, memory safety, format not needed
  - error reporting about the conversion outcome
  - speedup factor 6 - 250

```
#include <charconv>
from_chars_result from_chars( const char* first, const char* last,
        INT_TYPE& value, int base = 10);
from_chars_result from_chars( const char* first, const char* last,
        FLOAT_TYPE& value, chars_format fmt = chars_format::general);
struct from_chars_result { const char* ptr; errc ec; };
```

structural bindings

```
to_chars_result to_chars( char* first, char* last,
        INT_TYPE_ value, int base = 10);
to_chars_result to_chars( char* first, char* last,
        FLOAT_TYPE value, [opt] chars_format fmt, [opt] int precision);
```

string_view

C++17

- **string_view**
  - **non-owning** (sub-)string read-only reference
    - does not manage the object lifetime
    - borrow type
  - pointer + size
  - easy to create/pass
- methods
  - **remove_prefix**
  - **remove_suffix**
  - **start_with** / **end_with** [C++20]
  - **contains** [C++23]
- automatic conversions
  - **const char * ⇝ string** - constructor (*not explicit*)
  - **const char * ⇝ string_view** - constructor (*not explicit*)
  - **string ⇝ string_view** - conversion operator
- explicit construction
  - **string_view ⇝ string** - explicit constructor

```
#include <string_view>
bool cs( const string& s1, const string& s2);
bool sv( string_view s1, string_view s2);
string str { "my string" };
cs( str, "string test");
sv( str, "string_view test");
```

allocation copy

```
string f( string_view sv) {
  return string{sv} + ".";
}
```

allocation copy

string_view pros & cons

- good for
  - efficient substrings without alloc/copying
  - efficient passing of string literals
- not so good for
  - using **string_view** for not-(yet)-supported functions
  - uncontrolled object lifetime
  - exact type needed

no string_view!
ptr to temp!!

```
string operator+( string_view x, string_view y)
{ return string{ x} + string{ y}; }
string ccat( string_view x, string_view y)
{  return x + y; }
string a, b;
auto c = ccat( a, b);
```

string

```
bool f( string_view sv) {
  map<string,whatever> m; ...
  // m.find( sv);
  m.find( string{ sv});
}
```

allocation
copy

⚠ don't mix string and string_view

⚠ don't return (newly created) string_view

string ⇢ string_view ⇝ string
⇢ string_view ⇝ string ...

allocation
copy

span

C++20

- **span**
  - non-owing bounds-safe view
    - contiguous objects
    - **vector, array, string**
    - ranges
  - alà string_view
    - value type semantics
    - { ptr, count } ⇝ pass-by-value
  - dynamic (known at runtime)
  - static/fixed-size (compile-time)

- subviews
  - **first last**
  - **subspan**

- conversions
  - **as_bytes**
  - **as_writable_bytes**

```
T* ptr = new int[len];
T arr[6];
span<T> sd1{ ptr, len};
span<T> sd2{ arr};  // size deduction
span<T,4> sf3{ arr+1};
span<T,8> sf4{ arr}; // error
```

```
auto span_static  = ss.first<3>();
auto span_dynamic = sd.first(3);
```

```
auto subspan = sd2.subspan(3,2);
```

```
float data[] { 3.141592f, 2.718281f };
auto const const_bytes
          = as_bytes( span{data});
```

span<byte>

span, mdspan

C++20/23

- typical use-case

```
void read_into( int* buffer, size_t buffer_size);

int buffer[BUFFER_SIZE];
read_into( buffer, BUFFER_SIZE);
```

runtime checks

compile-time checks

```
void read_into( span<int> buffer);

int buffer[BUFFER_SIZE];
read_into( buffer);
```

  - array is passed to a function as a pointer
    - the size is lost
    - frequent reason for errors in C/C++
  - avoid free-standing pointers
  - prefer compile-time checking to run-time checking

- multidimensional span [C++23]
  - **mdspan**
  - row-wise, column-wise
  - user-defined layout
  - requires multidimensional **operator[]**

view::iota

```
vector v = {0,1,2,3,4,5,6,7,8,9,10,11};

auto ms2 = mdspan( v.data(), 3, 4);
auto ms3 = mdspan( v.data(), 2, 3, 2);

for(size_t i=0; i != ms3.extent(0); i++)
  for(size_t j=0; j != ms3.extent(1); j++)
    for(size_t k=0; j != ms3.extent(2); k++)
      ms3[i,j,k] = i*100 + j*10 + k;
```

User-defined literals

- user-defined suffix
  - integral number, floating-point number, character, string
  - originally: STL extensibility
  - useful, e.g., for computations using different units

(CNN) NASA lost a **$125000000** Mars orbiter because one engineering team used **English** units of measurement while the other team used the **metric** system

```
Kilograms w = 200.5_lb + 100.1_kg;
Time t = 2_h + 23_m + 17_s;
Year y = "MCMLXVIII"_r + 48;
```

```
SomeType operator "" _suffix( const char *);
SomeType operator "" _suffix( unsigned long long);
SomeType operator "" _suffix( long double);
```

user-defined suffixes must begin with _

User-defined literals

```
Kilograms w = 200.5_lb + 100.1_kg;
Kilograms x = 200.5;      // error
```

```
class Kilograms {
  double rawWeightKg;
public:
  class DoubleIsKilos{};     // a tag
  explicit constexpr Kilograms(DoubleIsKilos, double wgt) : rawWeightKg(wgt) {}
};
```

only specialized constructor
wieght = 200.5 not allowed

```
constexpr Kilograms operator "" _kg( long double wgt) {
  return Kilograms{Kilograms::DoubleIsKilos{}, static_cast<double>(wgt)};
}
constexpr Kilograms operator "" _lb( long double wgt) {
  return Kilograms{Kilograms::DoubleIsKilos{}, static_cast<double>(wgt*0.4535)};
}
```

conversion

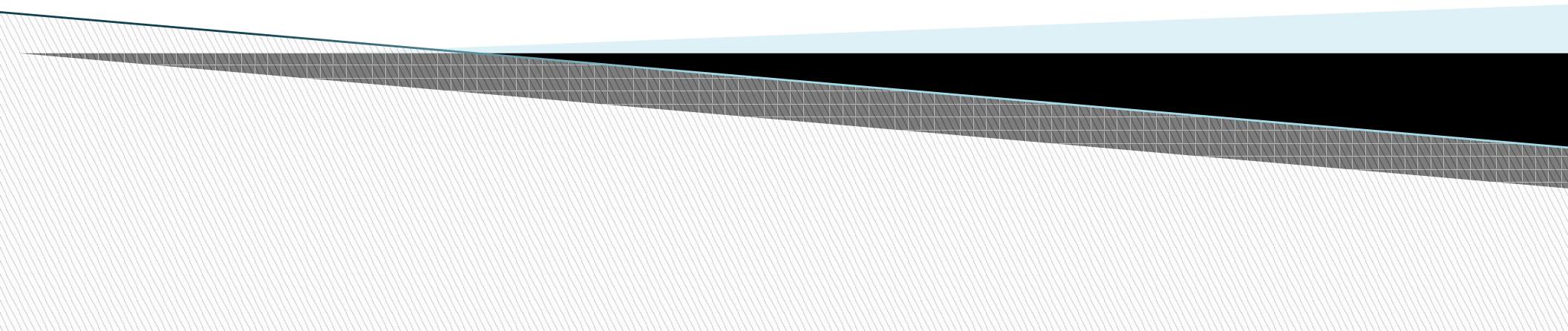
User-defined literals and strings

```
using namespace std::literals;

pair<string,int> p1( "A",1);     // C++98
auto p2 = make_pair( "A", 1);    // C++11 -> pair<const char*,int>
auto p3 = make_pair( "A"s, 1);   // C++14 -> pair<string,int>
pair p4{ "A"s, 1};               // C++17 CTAD
```

```
string_view s1 = "abc\0\0def";
string_view s2 = "abc\0\0def"sv;
cout << "s1: " << s1.size() << " '" << s1 << "'";
cout << "s2: " << s2.size() << " '" << s2 << "'";
```

```
literals::string_literals::operator""s( const char*) ➟ string
literals::chrono_literals::operator""s( unsigned long long) ➟ second
```

Other Libraries



format

C++20

- printf/python-like formatting
  - automatic parameters
  - indexed parameters
  - named parameters
    - mismatched names ⇝ exception
  - chrono integration
- user defined types
- print

C++23

  - streams no more needed
  - Unicode support
  - ranges formatting
  - performance (3x)

string formatting

```
string s = fmt::format( "{}-{}", "a", 1);
```

```
format( "{1:*>3}-{0}", 8 ,"b");  // **b-8
```

```
int width = 10;
fmt::format( "{num:{width}.{prec}f}",
   fmt::arg( "width", width),
   fmt::arg( "prec", 3)),
   fmt::arg( "num", 12.34567));
// "    12.345"
```

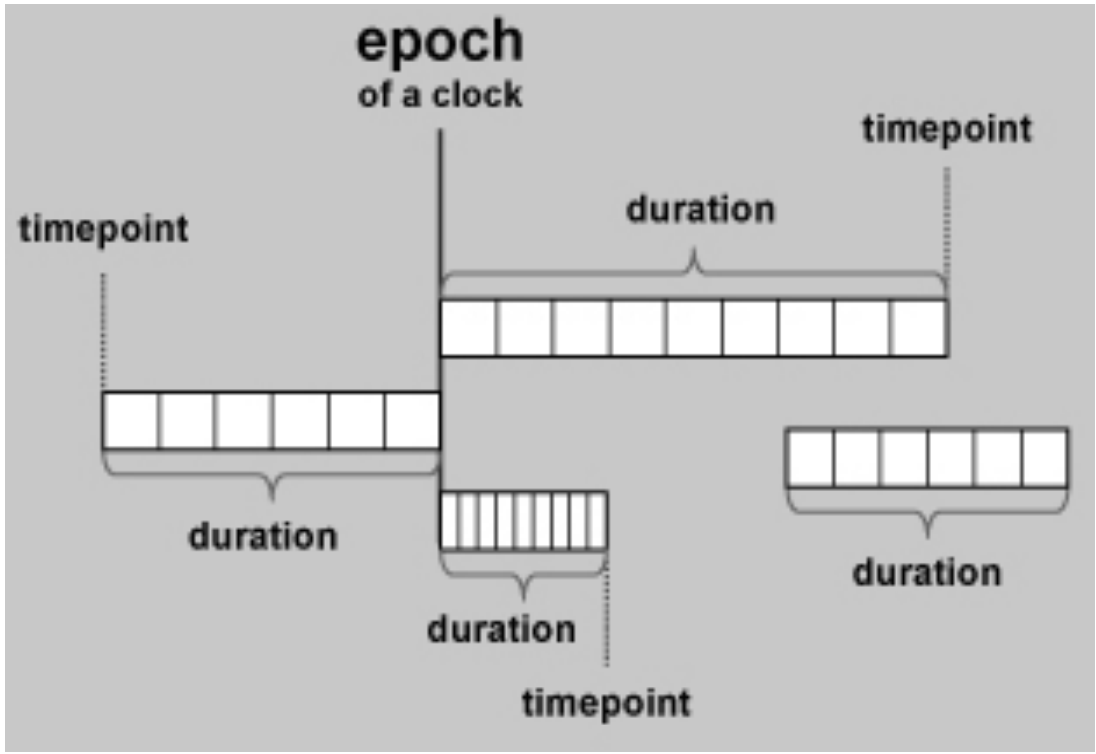
float formatting

```
enum class clr { red, green, blue };
clr c = clr::blue;
string s = format( "{}", c);
```

```
template ..... // 3 lines of templatish boilerplate
  string_view name = "???";
  switch( c) {
    case clr::red:   name = "red";   break;
    case clr::green: name = "green"; break;
    case clr::blue:  name = "blue";  break;
  }
  return formatter<string_view>::format( name, ctx);
}
```

```
cout << format( "{}{}", "a", 1);
print( "{}{}", "a", 1);
```

chrono



- epoch
- duration
  - various time units
  - time arithmetic
  - strong type checking
- clock
  - system_clock
    - system **real-time** clock
    - to_time_t(), from_time_t() - conversion from/to time_t
  - steady_clock
    - monotonic clock, **never** decreasing
    - not related to wall clock time, suitable for measuring
  - high_resolution_clock
    - the clock with the shortest tick period available
- timepoint - time interval from the start of the clock's epoch

! leap seconds
daylight saving time

Nicolai Jossutis: The C++ Standard Library: Utilities
www.informit.com/articles/article.aspx?p=1881386&seqNum=2

chrono

```
#include <iostream>
#include <chrono>
#include <thread>
using namespace chrono;

void sleep_ms( int ms)
{
  auto t0 = high_resolution_clock::now();
  this_thread::sleep_for( milliseconds(ms));
  auto t1 = high_resolution_clock::now();
  milliseconds total_ms = duration_cast<milliseconds>(t1 - t0);
  cout << total_ms.count();
}
```

various time units
strong type checking

h min s ms us ns
y ~~month~~ d

```
chrono::seconds twentySeconds(20);
chrono::hours   aDay(24);
chrono::milliseconds ms;
using namespace chrono_literals;

auto tm = 1h + 23min + 45s;
ms = tm + twentySeconds + aDay;
--ms;
ms *= 2;
```

chrono calendars & timezones

C++20

- C++20 - significant chrono extension
  - **calendar support**
    - **time_of_day, day, month, year, weekday, month_day, year_month_day, ...**

```
year_month_day ymd = 14d/11/2019;
sys_days d{ ymd};
d += weeks{ 1};
cout << ymd << d
     << format( "{:%d.%m.%Y}", ymd);
auto d2 = Thursday[2] /November/2019;
```

  - **time zone**
    - **tzdb, locate_zone, current_zone, time_zone, sys_info, zone_time, ...**
  - various real-world / IT clocks
    - **utc_clock, tai_clock, gps_clock, file_clock, local_t**
  - conversions
    - **clock_time_conversion, clock_cast**
  - input/output
    - **format, parse**

```
auto zt = chrono::zoned_time{..}
cout << format( locale{ "cs_CZ"},
        "Local time: {:%c}", zt)
     << format( "{:%d.%m.%Y %T}", zt);
```

compiler support:
en.cppreference.com/w/cpp/compiler_support

Howard Hinnant: Design Rationale for Chrono
www.youtube.com/watch?v=adSAN282YIw

regex

- templates
  - **regex** regular expression, various syntax
  - **match_results** container for regex_search results
    - **cmatch = match_results<const char*>** **wcmatch**
    - **smatch = match_results<string::const_iterator>** **wsmatch**
- functions
  - matching
    - **bool regex_match(string, regex)**
    - determines if **regexp** matches the entire **string**
  - searching
    - **bool regex_search(string, smatch, regex)**
    - searches for **regex** as any part of **string**
  - replacing
    - **string regex_replace(string src, regex, string fmt)**
    - searches for all appearances of **regex** in **src**; replaces them using **fmt**
- supported regex syntax
  - **ECMAScript**, basic, extended, awk, grep, egrep

regex

```
#include <regex>
const regex patt( R""((\+|-)?[[:digit:]]+)"");
string inp{ "-1234"};
if( regex_match( inp, patt)) ....
```

raw string - escape

```
string days{ "Saturday and Sunday, some Fridays."};
smatch match;
if( regex_search( days, match, regex{ R"(\w+day)"})) {
    cout << match[0];
}
cout << regex_replace( days, regex{ "day"}, "tag" );
```

Saturday

Saturtag and Suntag,
some Fritags.

random

- C: stdlib ~~rand~~
- generators
  - generate uniformly distributed values
  - *linear_congruential, mersenne_twister, substract_with_carry, ...*
- distributors
  - transform a generated sequence to a particular distribution
  - *uniform, normal, poisson, student, exponential, ...*
- usage: distributor(generator)

10 generators
20 distributors

```
#include <random>
default_random_engine generator;
default_random_engine generator(time(0));
uniform_int_distribution<int> distrib(1,6);
int dice_roll = distrib(generator);
```

equal sequence

seed

generates 1..6

generator is called by distributor

```
auto dice = bind( distrib, mt19937_64);
auto dice = []{ return distrib(mt19937_64); };
int wisdom = dice()+dice()+dice();
```

functor binding generator and distributor

filesystem

C++17

- platform independent operations on file systems and their components
  - paths, regular files, directories, symlinks/hardlinks, attributes, rights, ...
- iterable filesystem
  - possibly recursively
  - copying, deleting, renaming, ...
- iterable paths and filenames
  - parts of a path
  - normalized concatenation

```
fs::path p{ "temp/"};
p += "user" + "data";
```

zřetězení
temp\userdata

```
fs::path p{ "temp"};
p /= "user" / "data";
```

část path
temp\user\data

```
dir( const string& tree)
{
  fs::path treep{ tree};
  for( auto&& de : fs::recursive_directory_iterator( treep)) {
    if( is_directory( de.status()))
      cout << ">> ";
    fs::path p{ de};
    cout << p.root_path() << ":" << p.parent_path() << ":" << p.filename
()
         << ":" << p.stem() << ":" << p.extension() << endl;
    for(auto&& pi : p) cout << *pi;
  }
}
```

path parts

More C++20

- if/switch (init; condition)
- apply
  - invoke the callable object with tuple as arg
  - tuple, pair, array
- flat_map/ _multimap
  - implemented within a vector
  - faster lookup, slower insert/delete
- spaceship operator <=>
- endian checking
- container.contains()
- synchronization library
  - atomics, shared_ptr, floating point, atomic_ref

... and lots of minor extensions

```
if(auto [it,cc] = mmap.insert(val); cc)
  use(it);
```

```
int f( int x, int y) {..}
auto t = make_tuple( 1, 2);
std::apply( f, t);
```

```
if( endian::native == endian::little)
```

```
if( c.contains( "val")) ....
```

More C++23

- ranges
  - **to,** ...
- modularized stl
  - **import std**
- **expected**
- **mdspan**
- **print println**
- explicit **this** parameter
- **string::contains**

```
void log( source_location loc =
    source_location::current()) {
cout << loc.file_name() << "("
    << loc.line() << ":" << loc.column() << ")\n"
    << loc.function_name();
}

int main(int, char*[]) {
    log();
}
```

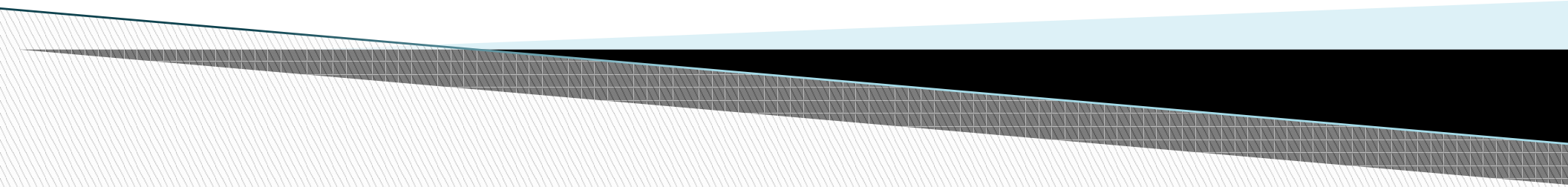
main.cpp(9:2)
int main(int, char**)

- stack trace
- **to_underlying(enum)**
- visit extension
  - classes derived from **variant**
- **spanstream**
  - full control over the memory management
- **out_ptr inout_ptr**
  - interoperability between C and **unique_ptr**

cppreference.com/w/cpp/compiler_support

Upcoming Features

beyond C++23



C++20/23

- ISO C++ Committee
  - C++20 - February 2020 - Prague
    - many major features
    - not all proposals approved
  - C++23 - February 2023
    - only minor features & corrections

feature	status	target
Concepts	**approved**	C++20
Coroutines	**approved**	C++20
Modules	**approved (incomplete)**	C++20 / 23
Ranges	**approved (incomplete)**	C++20 / 23
Executors	compromise design submitted for C++23	C++~~23~~ ⇝26?
Networking	TS based on Executors	C++26?
Contracts	TS published	C++~~23~~ ⇝26
Reflection	TS published	C++26?

C++26?

- contracts
  - preconditions, postconditions, assertions
  - documentation, rt-exception

```
T& operator[](size_t i)
[[expects: i < size()]] ;
class MyVector {
  void MyVector::push_back(Elem e)
  [[ensures: data != nullptr]];
  Data* data;
};
```

- executors
  - TS too large
  - another proposal - more practical
- networking
  - sockets, buffers, timers, aio, protocols, ...
  - depends on executors
  - TS 400 pages!
    - no consensus
    - redesign?

```
tcp::acceptor my_acceptor = ..      // listening socket
auto my_thread_pool_executor = ..   // ex for a thread pool

acceptor.async_accept(              // new connection
  bind_executor( my_thread_pool_executor,
    [](errc ec, tcp::socket new_connection) {..})
  );
```

C++26?

```
tuple t{ 0, 'a', 3.14};
apply([](auto&&... args) {((cout << args), ...);}, t);
```

- template for
  - tuples, variadic templates, ...

```
tuple t { 0, 'a', 3.14};
template for( auto e : t)
    cout << e;
```

```
template <typename ...Ts> min( Ts ...args) {
  auto min = head( args...);
  template for(auto x : tail(args...))
    if( x < min) min = x;
  return min;
}
```

- compile-time reflection
  - GUI generation, frameworks
  - RPC, network, db, serialization
  - documentation, ...

```
void f(int);
void f(string);
using f_call_m = reflexpr(f(123));
using f_m = get_callable_t<f_call_m>;
using param0_m = get_element_t
    <0, get_parameters_t<f_m>>;
cout << get_name_v<get_type_t<param0_m>>;
```

Beyond C++26

- pattern matching and language variants
  - std::variant - inconvenient syntax
  - inspection - generalized switch
    - pattern matching
    - variant selection

www.open-std.org
/jtc1/sc22/wg21/docs/papers
/2016/p0095r1.html

```
struct sum {
  expr* lhs, rhs;
};
lvariant expr {
  sum sum_exp;
  int lit;
  string var;
};

expr simplify( const expr& exp) {
return inspect(exp) {
  sum {*(lit 0),     *rhs} => simplify(rhs)
  sum {*lhs    , *(lit 0)} => simplify(lhs)
  sum {*lhs    ,     *rhs} => sum{....}
  _ => exp
};
```

```
lvariant tree {
  int leaf;
  pair< tree*, tree* > branch;
}
int sum_of_leaves( const tree & t ) {
  return inspect( t) {
    leaf i => i
    branch b => sum_of_leaves(*b.first) + sum_of_leaves(*b.second)
  };
}
```

```
lvariant command {
  size_t set_score;
  monostate fire_missile;
  unsigned fire_laser;
  double rotate;
};
```

```
struct set_score { size_t value; };
struct fire_missile {};
struct fire_laser { unsigned intensity; };
struct rotate { double amount; };

struct command {
  variant< set_score, fire_missile,
    fire_laser, rotate > value;
};
```

Beyond C++26

- metaclass
  - abstraction authoring mechanism for encapsulating behavior
  - defining categories of classes with common defaults and generated functions

Applying a reusable abstraction with defaults, generated functions, and **custom semantics** = XL improvement

```
template <class T1, class T2>
struct pair {
  using first_type  = T1;
  using second_type = T2;
  T1 first;
  T2 second;
  template <class... Args1, class... Args2>
    pair(piecewise_construct_t,
         tuple<Args1...> args1,
         tuple<Args2...> args2);
  constexpr pair();
  pair(const pair&) = default;
  pair(pair&&) = default;
  pair& operator=(const pair& p);
  pair& operator=(pair&& p) noexcept(see below);
  void swap(pair& p) noexcept(see below);
  explicit constexpr pair(const T1& x, const T2& y);
  template<class U, class V>
    explicit constexpr pair(U&& x, V&& y);
  template<class U, class V>
    explicit constexpr pair(const pair<U, V>& p);
  template<class U, class V>
    explicit constexpr pair(pair<U, V>&& p);
  template<class U, class V>
    pair& operator=(const pair<U, V>& p);
  template<class U, class V>
    pair& operator=(pair<U, V>&& p);
};

template <class T1, class T2>
  constexpr bool operator==
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator<
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator!=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator>
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator>=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator<=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template<class T1, class T2>
  void swap(pair<T1, T2>& x, pair<T1, T2>& y)
    noexcept(noexcept(x.swap(y)));
template <class T1, class T2>
  constexpr pair<V1, V2>
    make_pair(T1&& x, T2&& y);
```

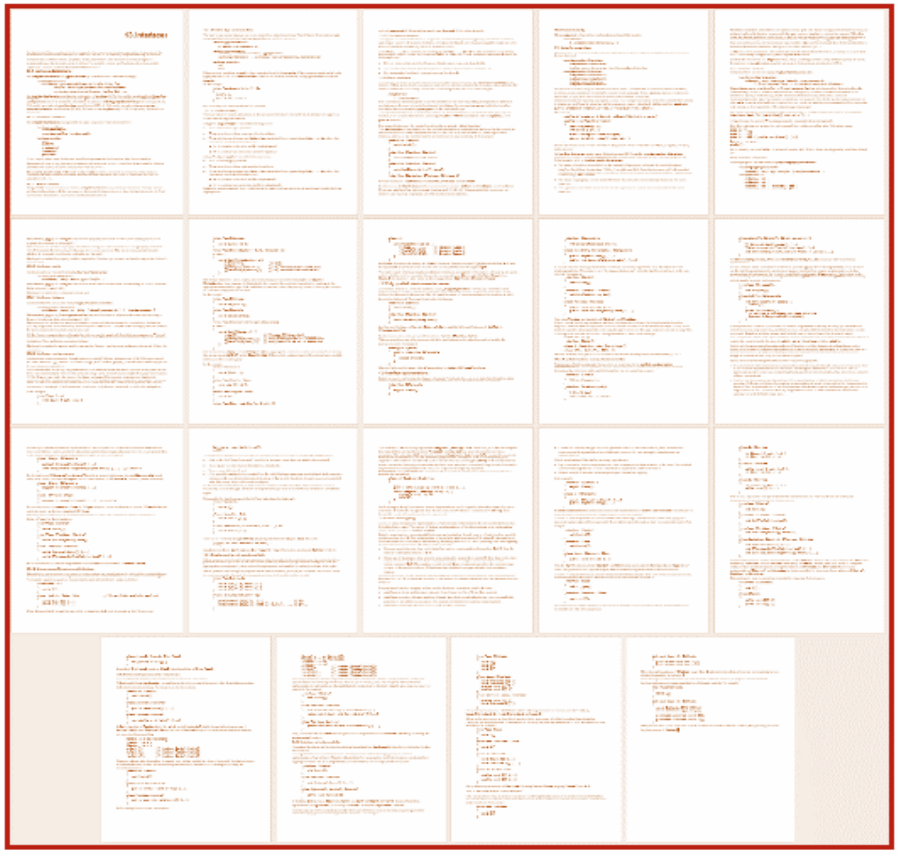
```
template<class T1, class T2>
aggregate pair {
    T1 first;
    T2 second;
};

// note: section 3 shows code for
// all metaclasses mentioned in the
// paper except for aggregate
```

Andrew Sutton, cppcon 2017
www.youtube.com
/watch?v=29IqPeKL_QY

Beyond C++26

Writing **as-if a new 'language' feature** using compile-time code + adding expressive power = XXL improvement

```
// C# language spec: ~20 pages of nontestable English
```



```
// User code (today's Java or C#)
interface Shape {
    int area();
    void scale_by(double factor);
}
```

```
// (Proposed) C++ library: ~10 lines of testable code
constexpr void
interface(meta::type target, const meta::type source) {
   compiler.require(source.variables().empty(),
      "interfaces may not contain data");
   for (auto f : source.functions()) {
      compiler.require(!f.is_copy() && !f.is_move(),
         "interfaces may not copy or move; consider"
         " a virtual clone() instead");
      if (!f.has_access()) f.make_public();
      compiler.require(f.is_public(),
         "interface functions must be public");
      f.make_pure_virtual();
      ->(target) f;
   }
   ->(target) { virtual ~(source.name()$)() noexcept {} }
};
```

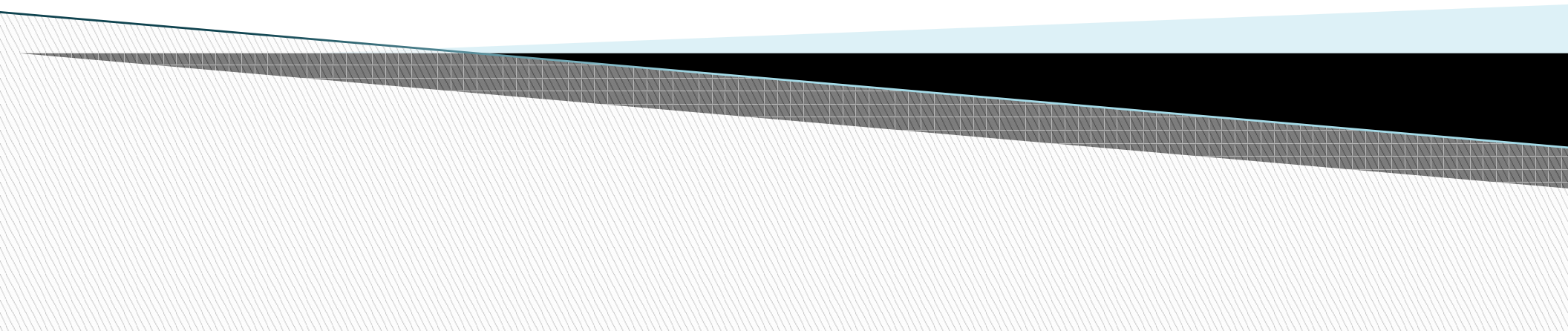
```
// User code (proposed C++)
interface Shape {
    int area() const;
    void scale_by(double factor);
};
```

... stay tuned:

concurrency & parallelism

metaprogramming

... but may be useful



New keywords: `char8_t`, `co_await`, `co_return`, `co_yield`, `concept`, `consteval`, `constinit`, `import*`, `module*`, `requires`
* identifiers with a special meaning

Concepts

Constrains on the template parameters and meaningful compiler messages in case on an error. Can also reduce the compilation time.

```
template <class T>
concept SignedIntegral = std::is_integral_v<T> &&
                         std::is_signed_v<T>;
template <SignedIntegral T> // no SFINAE here!
void signedIntsOnly(T val) { }
```

Modules

The replacement of the header files! With modules you can divide your program into logical parts.

```
import helloworld; // contains the hello() function
int main() {
   hello(); // imported from the "helloworld" module!
}
```

Coroutines

Functions that can suspend their execution and can be resumed later, also asynchronously. They are associated with a promise object and might be allocated on the heap. C++20 gives language support. Use libs like cppcoro for full functionality (generators objects).

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

operator<=>

New operator that can define all other comparison operators.

`R operator<=>(T, T);` where R is the Ordering category

```
(a <=> b) < 0 if a < b
(a <=> b) > 0 if a > b
(a <=> b) == 0 if a and b are equal/equivalent.
```

Designated Initializers

Explicit member names in the initializer expression:

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

Range-based for with initializer

Create another variable in the scope of the for loop:

```
for (int i = 0; const auto& x : get_collection()) {
    doSomething(x, i);
    ++i;
}
```

char8_t

Separate type for UTF-8 character representation, the underlying type is `unsigned char`, but they are both distinct. The Library also defines now `std::u8string`.

Attributes

`[[likely]]` - guides the compiler about more likely code path
`[[unlikely]]` - guides the compiler about uncommon code path
`[[no_unique_address]]` - useful for optimisations, like EBO
`[[nodiscard]]` for constructors – allows us to declare the constructor with the attribute. Useful for ctors with side effects, or RAII.
`[[nodiscard("with message")]]` – provide extra info
`[[nodiscard]]` is also applied in many places in the Standard Library

Structured Bindings Updates

Structured bindings since C++20 are more like regular variables, you can apply `static`, `thread_storage` or capture in a lambda.

Class non type template parameters

Before C++20 only integral types, enums, pointer and reference types could be used in non type template parameters. In C++20 it's extended to classes that are Literal Types and have "*structural equality*".

```
struct S { int i; };
template <S par> int foo() { return par.i + 10; }
auto result = foo<S{42}>();
```

explicit(bool)

Cleaner way to express if a constructor or a conversion function should be `explicit`. Useful for wrapper classes. Reduces the code duplication and SFINAE.

```
explicit(!is_convertible_v<T, int>) ...
```

constexpr Updates

`constexpr` is more relaxed you can use it for `union`, `try` and `catch`, `dynamic_cast`, memory allocations, `typeid`. The updates allows us to create `constexpr std::vector` and `std::string` (also part of C++ Standard Library changes)! There are also `constexpr` algorithms like `std::sort`, `std::rotate`, `std::reverse` and many more.

consteval

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

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

constinit

Applied on variables with static or thread storage duration, ensures that the variable is initialized at compile-time. Solves the problem of static order initialisation fiasco for non-dynamic initialisation. Later the value of the variable can change.

std::format

Python like formatting library in the Standard Library!

```
auto s = std::format("{:-^5}, {:-<5}", 7, 9);
```

`s` has a value of „`--7--, 9----`" centred, and then left aligned
Also supports the Chrono library and can print dates

Ranges

A radical change how we work with collections! Rather than use two iterators, we can work with a sequence represented by a single object.

```
std::vector v { 2, 8, 4, 1, 9, 3, 7, 5, 4 };
std::ranges::sort(v);
for (auto& i: v | ranges:view::reverse) cout << i;
```

With Ranges we also get new algorithms, views and adapters

Chrono Calendar And Timezone

Heavily updated with Calendar and Timezones

```
auto now = system_clock::now();
auto cy = year_month_day{floor<days>(now)}.year();
cout << "The current year is " << cy << '\n';
```

Multithreading and Concurrency

- `jthread` - automatically joins on destruction. Stop tokens allows more control over the thread execution.
- More atomics: floats, `shared_ptr`, `weak_ptr`, `atomic_ref`
- Latches, semaphores and barriers – new synchronisation primitives

std::span

Non owning contiguous sequence of elements. Unlike `string_view`, `span` is mutable and can change the elements that it points to.

```
vector<int> vec = {1, 2, 3, 4};
span<int> spanVec (vec);
for(auto && v : spanVec) v *= v;
```

Other

- Class Template Argument Deduction for aliases and aggregates, and more CTAD in the Standard Library
- template-parameter-list for generic lambdas
- Make `typename` optional in more places
- Signed integers are two's complement
- `using enum` – less typing for long `enum class` names
- Deprecating `volatile`
- Pack expansion in lambda init-capture
- `std::bind_front()` - replacement for `std::bind()`
- String prefix and suffix checking
- `std::bit_cast()` and bit operations
- Heterogeneous lookup for unordered containers
- `std::lerp()` and `std::midpoint()`, Math constants
- `std::source_location()` – get file/line pos without macros
- Efficient sized delete for variable sized classes
- Feature test macros and the `<version>` header
- `erase/erase_if` non-member functions for most of containers!

www.bfilipek.com/2020/01/cpp20refcard.html

Concepts

- named predicates on template parameters
  - evaluated at **compile** time
  - limits the set of template arguments
  - no runtime overhead, negligible compilation overhead
  - standard concepts
  - user-defined concepts

```
template <typename T>
concept bool equality_comparable() =
  requires(T a, T b) {
    {a == b} -> bool;
    {a != b} -> bool;
  };

template<typename T>
requires equality_comparable<T>
bool compare(const T& x, const T& y) {
  return x == y;
}
```

usable in boolean context

```
template<typename T>
requires
  equality_comparable<T> && integral<T>
bool compare(const T& x, const T& y) {
  return (x+1) == (y-1);
}
```

```
template<typename T>
requires
  equality_comparable<T> || same<T, void>
bool compare(const T& x, const T& y) {
  ...
}
```

```
template<equality_comparable T>
bool compare(const T& x, const T& y) {
  return x == y;
}
```

Concepts - constrained type deduction

- placeholders
  - constrained automatic type deduction
    - **auto** ≈ unconstrained placeholder
    - **concept auto** ≈ constrained placeholder
  - **constrained** placeholders may be used anywhere **auto** may be used
  - auto & concept may appear in parameters
    - they turn function declarations into template declarations

```
template<equality_comparable T>
bool compare(const T& x, const T& y) {
  return x == y;
}
```

terse syntax

```
sortable auto x = f(y);
```

```
bool compare(const equality_comparable auto& x,
     const equality_comparable auto& y) {
  return x == x && y == y;
}
```

constrained placeholders
⇝ template

```
auto f( container auto) -> sortable;
```

```
template< typename T>
requires container<T>
auto f( T) -> sortable;
```

Concepts - overload resolution

- concept ordering
  - the **most constrained** concept is chosen
    - when the argument satisfies **more** concepts
  - applicable to
    - function template overloads
    - class template specializations

```
void f( auto x);
void f( integral auto x);
void f( unsigned long x);

f(3.14);     // auto
f(2010);     // integral
f(2020ul);   // ul
```

Standard Concepts

- foundational concepts
  - same<T,U>, derived_from<T,U>, convertible_to<T,U>
  - movable<T>, copyable<T>, assignable<T>, swappable<T>
  - constructible<T>, default_c.<T>, move_c.<T>, destructible<T>
  - integral<T>, boolean<T>
  - equality_comparable<T>, equality_comparable<T,U>
  - totally_ordered<T>, totally_ordered<T,U>, ordered<T>
  - semiregular<T>, regular<T>
- functions
  - function<F, Args...>, predicate<P, Args...>, relation<P,T>
  - unary_operation<F,T>, binary_operation<F,T>
- iterators and ranges
  - input_iterator<I>, output_iterator<I,T>
  - forward_iterator<I>, bidirectional_iterator<I,T>
  - random_access_iterator<I,T>
  - range<R>

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

```
template <random_access_iterator T>
void sort( T, T);
```

concept 'random_access_iterator' not satisfied

Coroutines

- execution state ╳ thread of execution

thread / state	single state	multiple states
single thread	sequential code	**coroutines**
multiple threads	SIMD	multithreading

- non-preemptive multitasking
  - can suspend execution
    - and return control to its caller
  - can resume execution
- new keywords: **co_yield**, **co_await**, **co_return**
- cooperative tasks, event loops, iterators, infinite lists, pipes

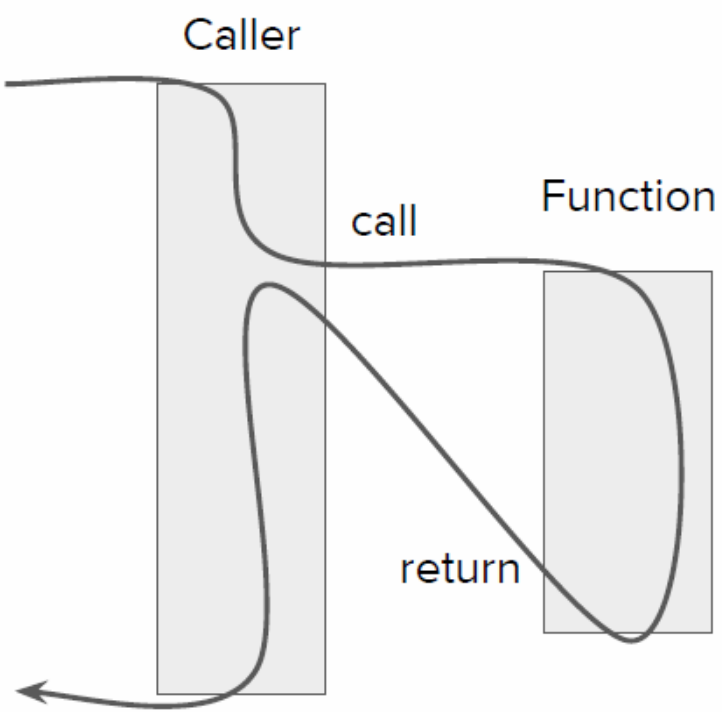
```
generator<int> integers( int start=0, int step=1 ) {
  for (int n=start; ; n+=step)
    co_yield n;
}
```

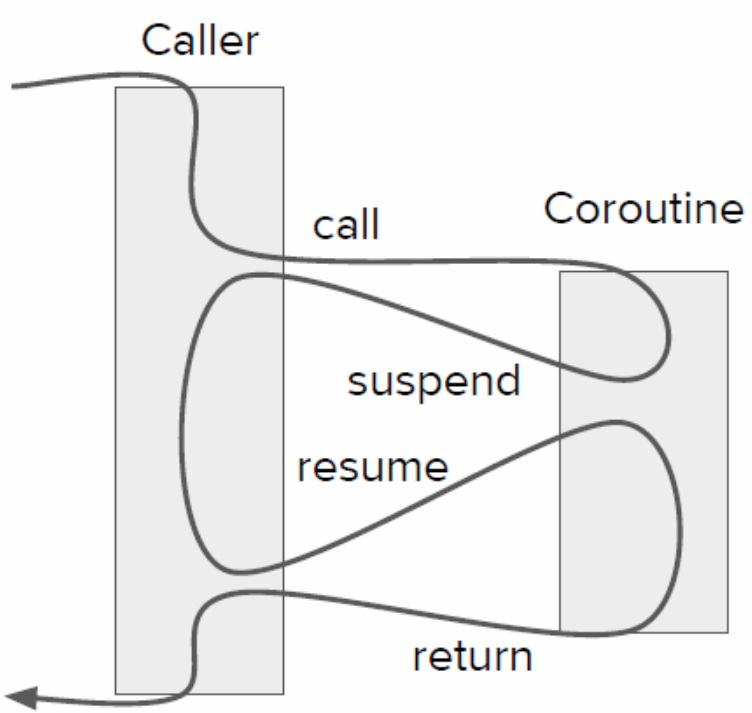
suspend / resume

```
for( auto&& i : integers( 5, 10))
  ....
```

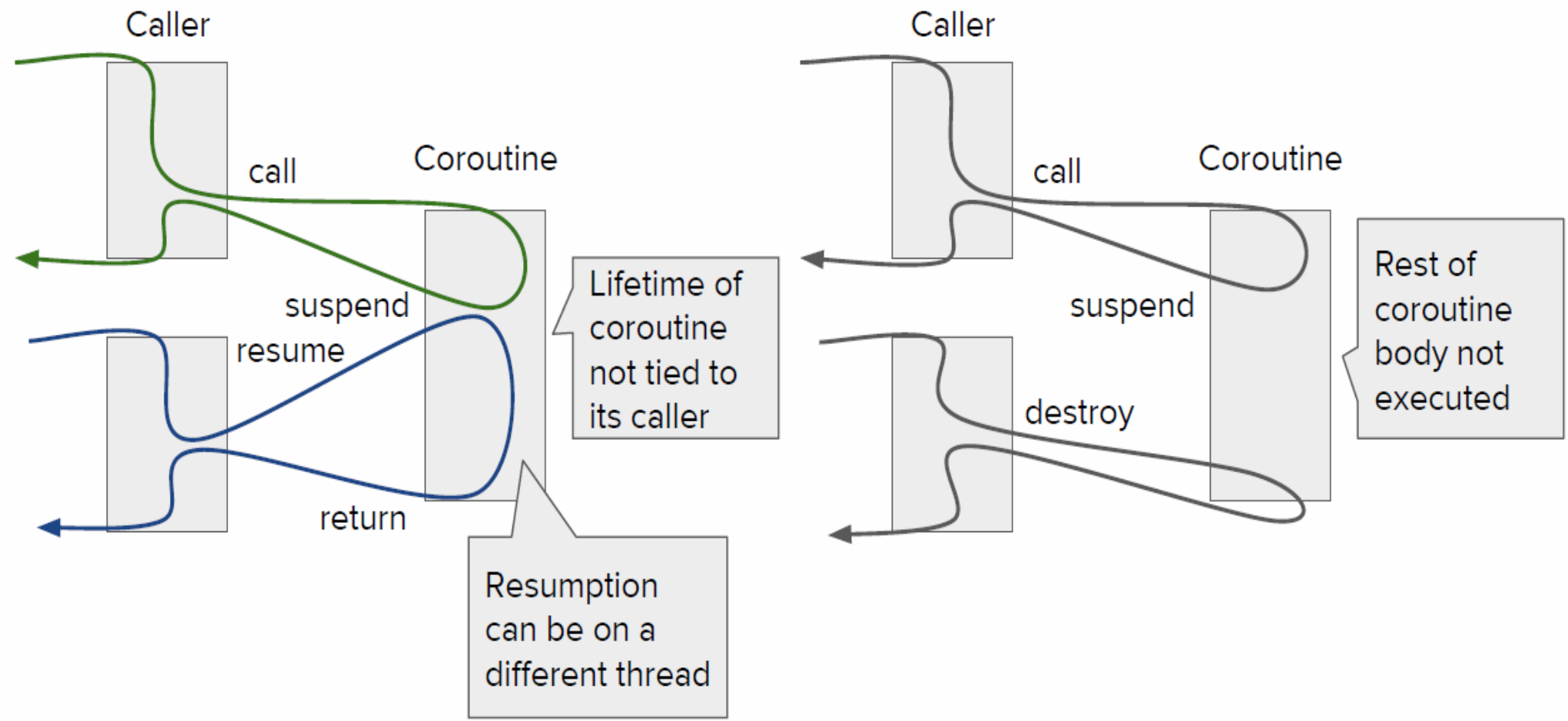
- co_yield
  - suspends the execution
  - stores that state in the generator<int>
  - returns the value through the generator<int>

Coroutines


Caller
call
Function
return

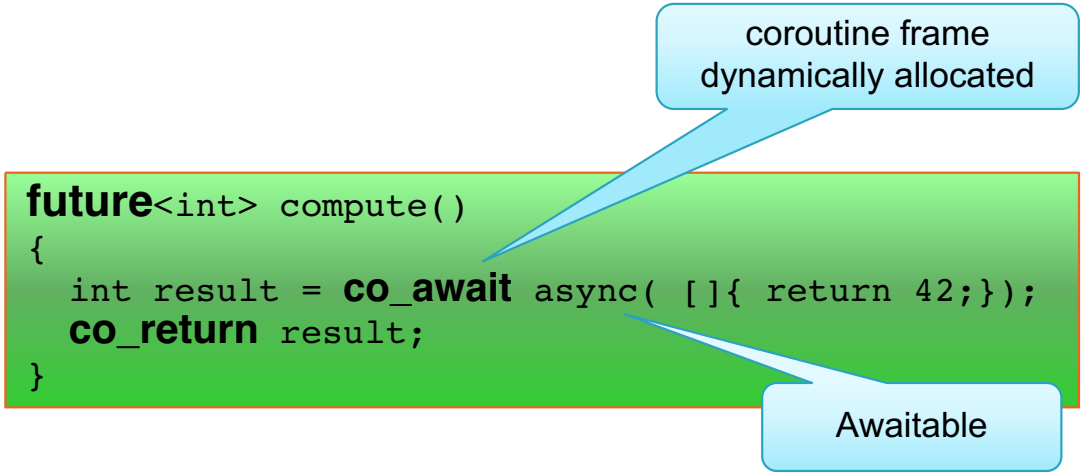

Caller
call
Coroutine
suspend
resume
return

Coroutines


Caller
call
Coroutine
suspend
resume
return
Lifetime of coroutine not tied to its caller
Resumption can be on a different thread
Caller
call
Coroutine
suspend
destroy
Rest of coroutine body not executed

Coroutines

- **co_await**
  - waiting (or not) for an awaitable thing
    - class, several methods required
  - usually another (async) coroutine
- **co_return**
  - return type ≠ coroutine type



- low-level constructs
  - intended for library writers

- design principles
  - scalable to **b**illions of concurrent coroutines
  - efficient: suspend/resume overhead comparable to function call
  - open-ended: libraries exposing high-level semantics (generators, tasks, ...)

Coroutines

```
resumable_thing named_counter( string name)
{
  cout << "cnt " << name << " was called\n";
  for (unsigned i = 1; ; ++i) {
    co_await suspend_always{};
    cout << "cnt " << name << " resumed #" << i << '\n';
  }
}

int main() {
  resumable_thing counter_a = named_counter("a");
  resumable_thing counter_b = named_counter("b");
  counter_a.resume();
  counter_b.resume();
  counter_b.resume();
  counter_a.resume();
}
```

Awaitable

```
cnt a was called
cnt b was called
cnt a resumed #1
cnt b resumed #1
cnt b resumed #2
cnt a resumed #2
```

Coroutines

Statement/Expression...	Equivalent to...
`co_return x;`	`__promise.return_value(x);` `goto __final_suspend_label;`

`co_await y`	`auto&& __awaitable = y;`
	`if (__awaitable.await_ready())`
	`{`
	`  __awaitable.await_suspend();`
	`  // ...suspend/resume point...`
	`}`
	
	`__awaitable.await_resume();`

`co_yield z`	`co_await __promise.yield_value(z)`

James McNellis: Introduction to C++ Coroutines
www.youtube.com/watch?v=ZTqHjjm86Bw
Toby Allsopp: Coroutines What Can't They Do
www.youtube.com/watch?v=mlP1MKP8d_Q

auto declaration

- original C[1974] semantics
  - local variable - storage class specifier
  - redundant
- problem
  - unnecessary verbosity, code

```
std::vector<int> v = { .. };
std::vector<int>::const_iterator i = v.begin();
```

  - explicit declaration is used only for compiler checks
  - may enforce automatic conversions
    - sometimes not intended
- C++11..14..17
  - inference of a variable type from the type of initializers
  - strongly typed language ➟ **compiler !**
    - not a polymorphic type

```
std::vector<int> v = { .. };
auto i = v.cbegin();
```

```
{
  auto int x;
  ...
}
```

Thomas Becker:
C++ auto and
decltype Explained

thbecker.net/articles/auto_and_decltype/section_01.html

auto - be careful

- exact type
  - C++03

```
typedef vector<int> vecType;
vecType v;
vecType::const_iterator it = v.begin();
```

  - C++11

```
vector<int> v;
auto it = v.begin();
```

What's the difference?

auto - be careful

- exact type
  - C++03

```
typedef vector<int> vecType;
vecType v;
vecType::const_iterator it = v.begin();
```

const_iterator

  - C++11

```
vector<int> v;
auto it = v.begin();
const vector<int> v2;
auto it2 = v2.begin();
```

iterator

const_iterator

  - deduced type is silently changed
  - cbegin(), cend()

```
vector<int> v;
auto it = v.cbegin();
```

auto - be careful!

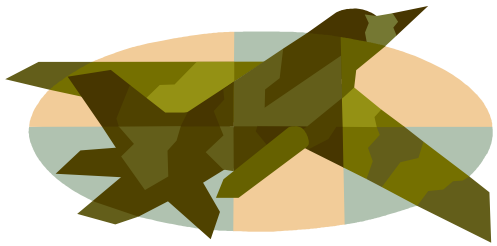
- exact type
  - C++03

```
std::deque<int> v = { .. };
for( int i = v.size()-1; i>0; i-=2) {
    f( v[i], v[i-1]);
}
```

any problem?

  - C++11

```
std::deque<int> v = { .. };
for( auto i = v.size()-1; i>0; i-=2) {
    f( v[i], v[i-1]);
}
```



... why?? where??

auto - be careful!

- exact type
  - C++03

```
std::deque<int> v = { .. };
for( int i = v.size()-1; i>0; i-=2) {
    f( v[i], v[i-1]);
}
```

size_t : unsigned

int i: 1-2 = -1

  - C++11

```
std::deque<int> v = { .. };
for( auto i = v.size()-1; i>0; i-=2) {
    f( v[i], v[i-1]);
}
```

unsigned int i:
1-2 = 4294967295

- use carefully when:
  - the interface depends on an exact type
  - inference from a complex expression
  - computation/correctness is based on an exact type

auto - be careful!

- exact type, size, ..., ...

```
for( auto i = 0; i < v.size(); ++i) ....
```

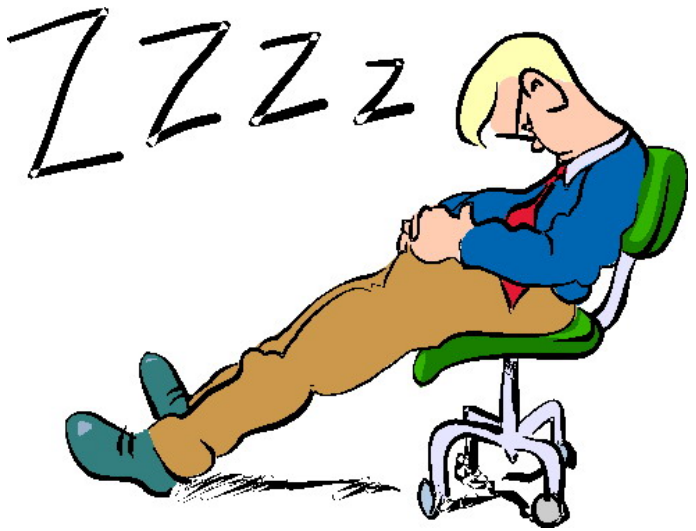
int i

wide type comparison

size_t
64-bit architectures:
unsigned long long

narrow type arithmetics

any problem?



auto - why to use

- Correctness automatically guaranteed
  - const_iterator ↚ const container& k.begin()
- Maintenance
  - type changes (containers, templates, ...)
- Performance
  - no implicit conversions
- Usability
  - lambdas, binders, template parameters, template expressions
- Readability
  - traditionally the motivation
- ... more use cases - *templates, lambdas, ...* ↬ later

auto in templates

```
template <typename T> void srt( T& x)
{
  ?? tmp = x[0];
```

```
vector< string> v;
srt( v);
```
tmp - what type?

- without auto

```
template < typename T> void srt( T& x)
{
  typename T::value_type tmp = x[0];
```

must be supported
```
template<typename T>
class vector {
  typedef T value_type;
```

```
template <typnm V, typnm T> void srt( T& x)
{
  V tmp = x[0];
```

user must specify
```
vector< string> v;
srt< string>( v);
```

- using auto

```
template <typename T> void srt( T& x)
{
  auto tmp = x[0];
```

no problem

Function parameters in templates

- functions/methods in templates
  - known container type
  - unknown element type

```
template <typename T> class Srt {
  sort( T& x) { auto tmp = x[0]; if( lt ( x[0], x[1])) ... }
  bool lt ( ?? a, ?? b);
```

what type of parameters?

```
bool lt( auto a, auto b);
```

error

```
using V = auto;
bool lt( V a, V b);
```

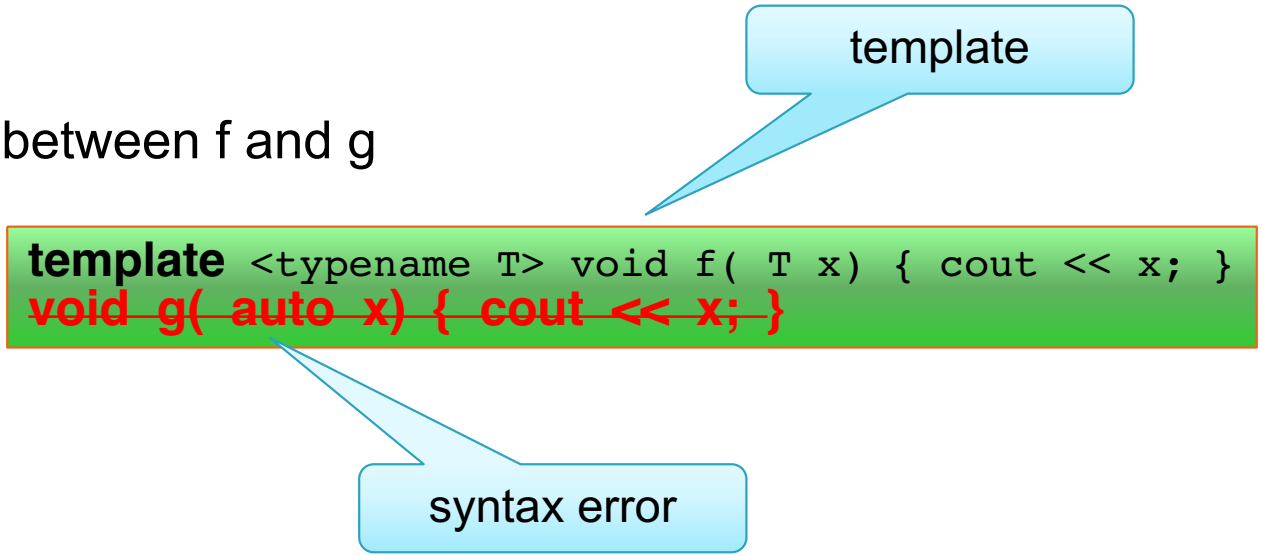
Templates vs. auto

- relation between f and g

```
template <typename T> void f( T x) { cout << x; }
void g( auto x) { cout << x; }
```

Templates vs. auto

- relation between f and g



  - f ? function template
    - all functions with all applicable types of the parameter
    - compiler deduces the type
  - g ? syntax error
    - compiler needs to deduce **one particular** type
    - ⇝ Concepts ⇝ C++20/23

auto in templates

- external template

```
template<typename V> bool lt (V& a,V& b) {}

template <typename T> class Srt {
  sort( T& x) { if( lt ( x[0], x[1])) ... }
```

is this correct?

auto in templates

- external template

```
template<typename V> bool lt (V& a,V& b) {}

template <typename T> class Srt {
  sort( T& x) { if( lt ( x[0], x[1])) ... }
```

global function
no access to members
namespace pollution

- nested template

```
template <typename T> class Srt {
  template<typename V> bool lt (V& a,V& b) {}
  sort( T& x) { if( lt ( x[0], x[1])) ... }
```

method template

  - method templates nested in class templates
  - accessibility and visibility rules similar to non-template classes

- *... or even better?*

```
template <typename T> class Srt {
  bool lt ( ??(T[0]) &a, ??(T[0]) &b) {}
```

idea: an expression
of the required type

decltype

```
decltype( dhl.list()) list;
tav::dh_list::bob_hell_type<tav::dh_list::bob_hell_list_trait
>
```

- **decltype**
  - type deduced from any expression
  - limited use for non-template programming
  - very useful when used in templates and metaprogramming

```
int i;
decltype(i) j = 5;
int f( decltype(i) x) { .. }
```

- usage

```
template <typename T> class Srt {
  bool lt ( decltype(*&T()[0])& a,
            decltype(*&T()[0])& b )
{}
```

expression not evaluated

declval<T>()[0]
can be used if no default constructor available

- or better

```
template <typename T> class Srt {
  using V = decltype(*&T()[0]);
  bool lt ( V& a, V& b ) {}
```

compare

```
using V = auto;
bool lt( V a, V b);
```

Comparison

- external template

```
template<typename V> bool lt (V& a,V& b) {}
template <typename T> class Srt {
  sort( T& x) { if( lt ( x[0], x[1])) ... }
```

global function
no access to members
namespace pollution

- nested template

```
template <typename T> class Srt {
  template<typename V> bool lt (V& a,V& b) {}
  sort( T& x) { if( lt ( x[0], x[1])) ... }
```

method template

- decltype

```
template <typename T> class Srt {
  bool lt ( decltype(*&T()[0])& a,
            decltype(*&T()[0])& b ) {}
```

type inference
expression not evaluated

- decltype using typedef

```
template <typename T> class Srt {
  using V = decltype(*&T()[0]);
  bool lt ( V& a, V& b ) {}
```

better readability

Lambda as parameter

- common use case
  - parameter in an algorithm

```
auto it = find_if( k.begin(), k.end(), [=](int x){ x < n; });
for_each( k.begin(), k.end(), [](int& x){ ++x; });
```

- how to write user-defined function?

```
vector<int> v;
void each_even( lambada l )
{
  auto i = v.begin();
  while( i < v.end()) {
    l( *i);
    if( ++i < v.end()) ++i;
  }
}

each_even( [](int& x){ ++x; });
```

parameter ? ? ?

parameterized code execution

argument: user-defined statement(s)

Lambda type

```
?? sum = [](int x, int y) { return x+y; };
```

- C++11: type of lambda expression:
  - *is a **unique**, unnamed nonunion class type*
  - the type cannot be named
  - decltype cannot be used for variable declaration

- std::function class
  - std::function<int(int,int)> sum
  - subtle rt overhead - lambda as a member of std::function object

```
std::function<int(int,int)> sum = [](int x, int y) { return x+y; };
```

- auto - no overhead

```
auto sum = [](int x, int y) { return x+y; };
```

different types

```
auto s1 = [](int x, int y) { return x+y; };
auto s2 = [](int x, int y) { return x+y; };
assert( typeid(s1) == typeid(s2));
```

Lambda as parameter

- auto
  - not allowed in parameters
- template
  - compiler deduces a particular type
  - usage must match the argument
- C++20
  - concepts - Function, Unary_operation, ...

```
vector<int> v;
template <typename lambada> void each_even( lambada&& l )
{
  auto i = v.begin();
  while( i < v.end()) {
    l( *i );
    if( ++i < v.end()) ++i;
  }
}

each_even( [](int& x){ ++x; });

each_even( [](int& x){ ++x; });
```

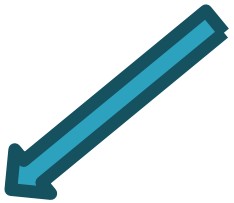
usage must be applicable to the argument

different types template - OK

Lambda - auto

- C++11 - declaration of parametr types

```
vector<int> v;
vector<unique_ptr<T>> ix;
```



C++11

```
for_each( v.begin(), v.end(),
  [](  decltype(*v.begin())& x)
  { cout << x; }
);

sort( ix.begin(), ix.end(),
  [](  const shared_ptr<T> & p1,
       const shared_ptr<T> & p2)
  { return *p1 < *p2; }
);

auto get_size = [](
  unordered_multimap<wstring,
     list<string>> & m )
  { return m.size();
};
```

template - unknown type

complex type

Lambda - auto

- C++11 - declaration of parametr types
- C++14 - auto

```
vector<int> v;
vector<unique_ptr<T>> ix;
```

C++11

```
for_each( v.begin(), v.end(),
  [](  decltype(*v.begin())& x)
  { cout << x; }
);

sort( ix.begin(), ix.end(),
  [](  const shared_ptr<T> & p1,
       const shared_ptr<T> & p2)
  { return *p1 < *p2; }
);

auto get_size = [](
  unordered_multimap<wstring,
     list<string>> & m )
  { return m.size();
};
```

C++14

```
for_each( v.begin(), v.end(),
  [](  auto&& x)
  { cout << x; }
);

sort( ix.begin(), ix.end(),
  [](  auto&& p1,
       auto&& p2)
  { return *p1 < *p2; }
);

auto get_size = [](  auto&& m )
{ return m.size(); };
```

bonus: any container supporting size()

Lambda - motivation

- find x < parameter...
  - C++03

```
class ftor {
public:
   ftor(int y) :y_(y) { }
   bool operator()(int x) const { return x<y_; }
private:
   int y_;
};
xxx::iterator i=find_if( v.begin(), v.end(), ftor( n));
```

  - C++11

```
auto i=find_if(v.begin(), v.end(), [=](int x){ return x<n; }
);
```

lambda

Lambda - syntax

- Syntax

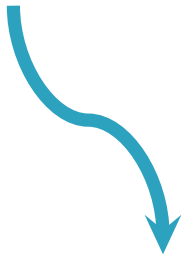
[captures] (params)$_{opt}$ mutable$_{opt}$ -> rettype$_{opt}$ { statements; }

  - **[captures]**
    - access to external local variables - **initialization**
    - explicit/implicit, by-value/by-reference, generalized
  - **(params)**
    - call parameters
    - optional but usual
  - **mutable**
    - local copy of external variable can be modified
  - **-> rettype**
    - return type, *"new"* syntax
    - optional - compiler deduces type from expression
    - template type deduction
  - **{ statements; }**
    - lambda body

what's the difference?

Lambda = funktor

```
[ captures ] ( params ) -> rettype { statements; }
```



```
class ftor {
private:
  CaptTypes captures_;
public:
  ftor( CaptTypes captures ) : captures_( captures ) {}
  rettype operator() ( params ) { statements; }
};
```

Lambda – capture

- capture
  - defines which external variables are accessible and how
    - local variables / this in the enclosing function
  - determines the data members and constructor parameters of the functor
- explicit capture
  - list of all enclosing *local* variables
  - variables marked & passed by reference, the others by value
    - value **copy** in time of **definition!**
  - this - pointer to the enclosing object
  - C++17 *this - copy of the enclosing - *parallelism*
- implicit capture
  - external variables determined automatically by the compiler
  - capture defines the mode of passing
    - can be complemented by explicit capture
- generalized capture
  - any expression
  - move semantics

[a,&b]

[this,*this]

[=] **[=,&b]**

[&] **[&,a]**

~~**[&,&a]**~~

ambiguities not allowed

[p=move(ptr)]

Lambda - nested functions

- an attempt to nest a function

```
int f( int i) {
  int j = i*2;
  int g( int k) { return j+k; }
  j += 4;
  return g( 3);
}
```

"nested function"
not possible in C++

- using lambda

```
int f( int i) {
  int j = i*2;
  auto g = [ ?]( int k) { return j +k; };
  j += 4;
  return g( 3);
}
```

capture	
=	(i*2)+3
&	(i*2+4)+3

capture variables here

Lambda - capture default

- *What-if …* default capture by reference??
  - [] [?] [?] [?]

```
function<void()> f() {
  int i = 0;
  auto fn = []{ DoWork( i); };
  DoOtherWork( i);
  return fn;
}
```

what's wrong?

function type:
void function / lambda

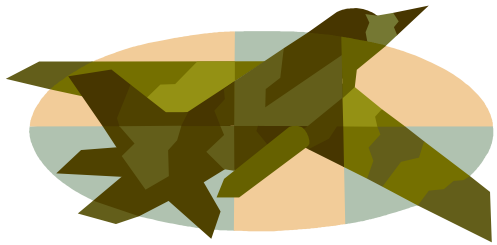
Lambda - capture default

- *What-if …* default capture by reference??
  - [] [?] [?] [?]

```
function<void()> f() {
  int i = 0;
  auto fn = [] { DoWork( i ); };
  DoOtherWork( i);
  return fn;
}
```

what's wrong?

- variable lifetime
  - calling **fn** after return from **f**
  - access to a nonexisting variable [?]



Lambda - capture default

- *What-if …* default capture by value??
  - [] ? ? ?

```
vector<int> v = ReadBigVectorFromDisk();
auto first = find( v.begin(), v.end(), 42);
auto lambda = []{ FindNext( v, first ); };
```

what's wrong?

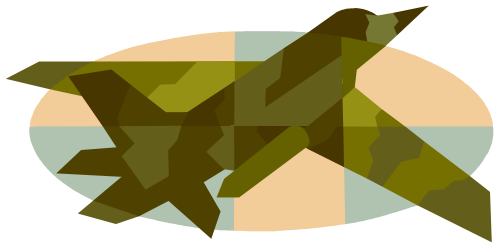
Lambda - capture default

- *What-if …* default capture by value??
  - [] ? ? ?

```
vector<int> v = ReadBigVectorFromDisk();
auto first = find( v.begin(), v.end(), 42);
auto lambda = [] { FindNext( v, first ); };
```

what's wrong?

- efficiency
  - deep copy of the vector
- correctness
  - **first** not an iterator of **v** ?



Lambda - no implicit default

- good decision
  - [] capture **nothing**!
  - the programmer **must** specify the capture mode
    - than *maybe* ...

```
function<void()> f() {
  int i = 0;
  auto fn = [=]{ DoWork( i); };
  DoOtherWork( i);
  return fn;
}
```

```
vector<int> v = ReadBigVectorFromDisk();
auto first = find( v.begin(), v.end(), 42);
auto lambda = [&v,first]{ FindNext( v, first ); };
```

Lambda – ugly ducklings

where and why mutable?

```
int a = 1, b = 1, c = 1;
auto m1 = [a, &b, &c]() {
  auto m2 = [a, b, &c]() {
    cout << a << b << c;
    a = 4; b = 4; c = 4;
  };
  a = 3; b = 3; c = 3;
  m2();
};
a = 2; b = 2; c = 2;
m1();
cout << a << b << c;
```

where and how are variables captured

```
std::function<int(int)> fact;
fact = [&fact](int n)->int {
  if(n==0) return 1;
  else return (n * fact(n-1));
};
cout << "factorial(4) : " << fact(4);
```

try to describe/draw the created data structures

Delegating constructors

- common functionality of constructor overloads
  - C++03

```
class X {
  X( int x) { init( x); }
  X( string& s) { init( stoi(s)); }
  init( int x) { a = x<0 ? 0 : x; }
  const int a;
};
```

unnecessary verbose
not in an init-list const,
performance

  - C++11

```
class X {
  X( int x) : a( x<0 ? 0 : x) {}
  X( string& s) : X( stoi(s)) {}
  const int a;
};
```

other constructor
can be called directly

{ X::X(a); }

Delegating constructors

- complex computations in a constructor
  - C++03

```
class Y {
public:
  Y( int x) : a(f(x)+1), b(f(x)+2) {}
private:
  int f( int x ) { return x*x; }
  int a, b;
};
```

complex computation, side-effects, ...

  - C++11

```
class Y {
public:
  Y( int x) : Y( x, tmp( x)) {}
private:
  struct tmp { int fx; tmp( int x) : fx( x*x) {}};
  Y( int x, tmp t) : a(t.fx+1), b(t.fx+2) {}
  int a, b;
};
```

intermediate result computed first

enum class

- C++03 enum
  - *"unscoped enum"*
  - weakly typed
  - value identifiers globally unique
  - variable size decided by a compiler
    - size/performance optimizations

```
enum Color { Red, Blue, Green } x;
enum Feelings { Happy, Sad } y;
int Blue;
if( x < y || y < 1) ...
```

ColorBlue
FeelingBlue

```
enum Color { Red, Blue, Green };
enum Feelings { Happy, Blue };
```

- C++11 enum class
  - *"scoped enum"*
  - strongly typed
  - local identifiers
  - exactly defined variable size
    - default: int

```
enum class E1 { Red, Blue } x;
if( x == 1) ...
int a = static_cast<int>(x);
enum class E2 : uint8_t { Happy, Blue };
E2 y { E2::Blue };
auto z = E2::Happy;
y = blue;
y = E1::blue;
```

scoped enum

C++11 scoped & unscoped enum

- C++11 scoped & unscoped enum
  - declaration of an **underlying type**

```
enum E3 : unsigned short { Xxx, Yyy };
enum class E4 : uint32_t { Xxx, Yyy };
```

  - forward declaration
    - elimination of the entire project recompilation

```
enum E5 : int;
enum class E6;
void f( E6 e);
```

u.t.

header - declaration

```
enum class E6 { Arsenal, Liverpool };
void f( E6 e) {
  if( e == E6::Arsenal) ....
```

values needed for implementation

  -

- usefulness of unscoped enum
  - implicit conversion to an integral type
    - bitmask, tuple, ...
  - redundant verbosity
    - enum in a class

```
auto x = get<static_cast<size_t>(Ids::idName)>(r);
```

```
enum Ids { idName, idAddress, idYear };
using Row = tuple<string, string, int>;
Row r;
auto x = get<idName>(r);
```

default, delete

- C++03
  - object copying prevention - private constr, op=
  - automatically generated methods
    - X::X(), X::X(const& X), X::operator=(const& X), X::~X()
    - default constructor is not generated if an explicit constructor defined
- C++11
  - better control
  - delete: compiler error instead of linker error
  - default: generated constructor explicitly added

```
class X {
  X() = default;
  X( int i) { .... }
  X( const& X) = delete;
  X& operator=( const& X) = delete;
};
```

Type alias

- std::unique_ptr<std::unordered_map<std::string, std::string>>
- 40 years old solution: typedef
  - **typedef** `up<um<s,s>> MapPtr;`
  - **typedef** `void (*FP)(int, const string&);`
- C++11: using
  - **using** `MapPtr = up<um<s,s>>;`
  - **using** `FP = void (*)(int, const string&);`

worse readability for many users

- main difference: templates / alias templates
  - C++03: encapsulation to class/struct necessary

```
template<typename T> using MyAllocList = std::list<T, MyAlloc<T>>;
MyAllocList<T> x;
```

```
template<typename T> struct MyAllocList {
  typedef std::list<T, MyAlloc<T>> type;
};
MyAllocList<T>::type x;
```

Type alias

- usage in templates
  - C++03: dependent name ⇒ typename
  - C++11: template alias ⇒ must be a type

```
template<typename T> using MyAllocList = std::list<T, MyAlloc<T>>;

template<typename T> class C {
  MyAllocList<T> x;        // no typename, no ::type
}
```

dependent name ⇝ typename

```
template<typename T> struct MyAllocList {
  typedef std::list<T, MyAlloc<T>> type;
};

template<typename T> class C {
  typename MyAllocList<T>::type x;
}
```

- std:: type transformations - type traits

C++11	C++14	
`remove_const<T>::type`	`remove_const_t<T>`	const T → T
`remove_reference<T>::type`	`remove_reference_t<T>`	T&/T&& → T
`add_lvalue_reference<T>::type`	`add_lvalue_reference_t<T>`	T → T&

Static assertions

- C/C++03 - runtime checks
  - expected state condition
    - f alse ⇝ error message and abort
  - programmers bugs protection
  - debug mode only *(#define NDEBUG)*
    - inactive in a release mode

```
int f( char* p) {
  assert( p != 0);
  *p == ...
```

if(false)
message; abort;

- C++11 - compile checks

```
enum color { Blue, Red, Yellow, COLORS };
string color_names[] = { "Blue", "Red", "Yellow" };
static_assert( (sizeof(color_names) / sizeof(string)) == COLORS,
  "colors don't match");
```

  - compile-time checks, user-defined compiler error
  - template parameters, data size, array range, constant values

⇝ concepts

```
template<class Intgr>
Integral f( Intgr x, Intgr y) {
  static_assert( is_integral<Intgr>::value,
    "parameter must be an integral type");
}
```

override, final

- C++03
  - undesired overloading
  - overloading ╳ overriding
  - *př etí žení ╳ .... "př eplácnutí "*

Переопределение

Bednárek 2017

```
class Base {
  virtual void f( double);
};

class Derived : Base {
  virtual void f( int );
};

Base* b = new Derived;
f(0);
```

  - non-intuitive use of **virtual** in a derived class

use carefully!
prevention of overriding

- C++11
  - override
  - base class virtual method must be defined

context keyword
can be in identifier
in other context

```
class Base {
  virtual void f( double);
};

class Derived : Base {
  virtual void f( int ) override;
};
```

syntax error

  - final

```
class FBase final {};
class FDerived : FBase {};
```

```
class Derived : Base {
  virtual void f( double) override final;
};
```

Fold expressions

+ - * / % ^ & | << >>
= @= <=> && || , .* ->*

- variadic templates
  - C++14 - recursion

C++14

```
auto old_sum(){ return 0; }
template<typename T1, typename... T>
auto old_sum(T1 s, T... ts){ return s + old_sum(ts...); }
```

  - C++17 - simplification

( ) required

```
template<typename... T> ????( T... pack)
( ... op pack )                 // ((P1 op P2) op ... Pn-1) op Pn
( pack op ... )                 // P1 op (P2 op ... (Pn-1 op Pn))
( init op ... op pack )         // ((init op P1) op ... Pn-1) op Pn
( pack op ... op init )         // P1 op (P2 op ... (Pn op init))
```

```
template<typename... T>
auto fold_sum(T... s){
    return (... + s);
}
```

C++17

```
template<typename... T>
auto fold_sum1(T... s){
    return (0 + ... + s);
}
```

first param

remaining parameters

```
template<typename ...Args> void prt(Args&&... args)
{ (cout << ... << args) << '\n';  }
prt( 1, "2", 3.14);
```

string conversions

- conversion string ⇝ numeric type
- old C libraries

```
#include <cstring>
string s = "1234";
int i = atoi( s.c_str());
```

- C++03

```
istringstream ss(s);
ss >> i;
```

int i = ... ?

- C++11

```
int i = stoi( s);
```

```
int stoi( const string& s);
unsigned long stoul
double stod
stoull, stof, stold
```

- cannot convert ⇝ throw
  - invalid_argument
  - out_of_range

```
string to_string( int i);
string to_string( double d i);
```

cstdint

signed	unsigned	
intmax_t	uintmax_t	maximum width integer type
int8_t	uint8_t / byte	width **exactly** 8/16/32/64 bits **Optional** byte - C++17
int16_t	uint16_t	
int32_t	uint32_t	
int64_t	uint64_t	
int_least8_t	uint_least8_t	**smallest** i. type with width of at least 8/16/32/64 bits
int_least16_t	uint_least16_t	
int_least32_t	uint_least32_t	
int_least64_t	uint_least64_t	
int_fast8_t	uint_fast8_t	**fastest** i. type with width of at least 8/16/32/64 bits
int_fast16_t	uint_fast16_t	
int_fast32_t	uint_fast32_t	
int_fast64_t	uint_fast64_t	
intptr_t	uintptr_t	integer type capable of holding a **pointer** **Optional**

numeric_limits

- template
  - standardized way to query various properties of arithmetic types
  - integer, real

```
#include <limits>
numeric_limits<int>::min()
```

```
min  max  lowest  digits  digits10  is_signed  is_integer  is_exact
is_specialized  radix  epsilon  round_error  min_exponent
min_exponent10  max_exponent  max_exponent10  has_infinity
has_quiet_NaN  has_signaling_NaN  has_denorm  has_denorm_loss
infinity quiet_NaN  signaling_NaN  denorm_min  is_iec559  is_bounded
is_modulo  traps  tinyness_before  round_style
```

bitset

- fixed-size set of bits
  - optimalized
  - access to individual bits
  - various forms of inicialization
  - `set, reset` - set/reset of some/all bits
  - `bool all(), any(), none()` - test

```
unsigned x = 0xDEADBEEF;
x &= 0xC0C0 << 2;
```

```
#include <bitset>
bitset<6>  a(42);
bitset<6>  b(0x1B);
bitset<18> c("1001001011101001");
string s = "BaaBBaBaaBBaB";
bitset<4>  d( s, 3, 4, 'a', 'B');
a ^= b;
a[2] = (~b << 2)[3];
a.set(1);
if( a.any()) ...
```

from, count,
what is 0/1

locale

- internacionalization support
  - locale - set of facets
  - facet [fæsit] - *ploška, stránka, aspekt, charakteristika tř í dy*

```
use_facet < numpunct<char> > (mylocale).decimal_point();
```

category	facet	member functions
collate	collate	compare, hash, transform
ctype	ctype	is, narrow, scan_is, scan_not, tolower, toupper, widen
	codecvt	always_noconv, encoding, in, length, max_length, out, unshift
monetary	moneypunct	curr_symbol, decimal_point, frac_digits, grouping, negative_sign, neg_format, positive_sign, pos_format, thousands_sep
	money_get	get
	money_put	put
numeric	numpunct	decimal_point, falsename, grouping, thousands_sep, truename
	num_get	get
	num_put	put
time	time_get	date_order, get_date, get_monthname, get_time, get_weekday, get_year
	time_put	put
messages	messages	close, get, open

string comparison

locale

- C++03

```
void print_date( const Date& d) {
  switch( loc) {
    default:
    case ISO: cout << d.year() << "-" << d.month() << "/" << d.day(); break;
    case US:  cout << d.month() << "/" << d.day() << "/" << d.year(); break;
    case CZ:  cout << d.day() << "." << d.month() << "." << d.year(); break;
}
```

- C++11

```
void print_date( const Date& d) {
  cout.imbue( locale{"en_US.UTF-8"});
  cout << d;
}
```

stream localization

```
template<class C>
inline bool isalpha( C c, const locale& loc) {
  return use_facet< ctype<C> >(loc).is ( alpha, c));
}
```

localization of char properties

sorting localization

z < å

```
locale dk{"da_DK"};
sort( v.begin(), v.end(), dk);
```