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

David Bednárek
Jakub Yaghob
Filip Zavoral

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

C++11/14/17/20/23/26

- Syntax
  - auto, structured bindings, type deduction
  - initialization, attributes, modules
- Types
  - type compatibility, type erasure
  - any, variant, optional, expected
  - char & string types, userdef literals
- Libraries
  - ranges, format, chrono, ...
- Future
  - C++23/26

cppreference.com
cppcon.org
herbsutter.com
stroustrup.com
cppstories.com
www.youtube.com/user/CppCon/

auto
range-based for

Range-based for

cyklus řízený rozsahem
Kruliš 2014:

```
map<string,int> m;
```

- C++03

```
for( map<string,int>::const_iterator i = m.begin(); i != m.end(); ++i)
 if( i->first == "Me")
       cout << i->second;
```

what is more readable?

- C++11

```
for( auto&& i : m) // a lot of space for my own valuable comments :-)))
 if( i.first == "Me")
       cout << i.second;
```

😐 ?

- C++17
  - structured bindings

Structured return values

- functions returning multiple values – C++03

```
void read( handle h, string& rs, bool& valid);
read( db, s, valid);
if( valid)
  doit( s);
```

not a function-like syntax

non-intuitive in/out par

```
bool read( handle h, string& rs);
if( read( db, s))
  doit( s);
```

non-intuitive separation

```
pair<string, bool> read( handle h);
pair<string, bool> rs = read( db);
if( rs.second)
  doit( rs.first);
```

unnecessary type verbose

non-descriptive fields

```
const string& read( handle h);
const string& rs = read( db);
if( result_ok( db))
  doit( rs);
```

querying state

additional fnc call

static internal state !!!

Structured return values

- C++11

```
pair<string, bool> read( handle h);

auto rs = read( db);
if( rs.second)
  doit( rs.first);
```

more convenient

😐 ?

bž

```
ææ read( handle h);

ææ(s,valid) = read( db);
if( valid)
  doit( s);
```

? ? ?

already defined variables

rozdělí strukturu na jednotlivé proměnné

Structured return values

- tuple, tie

```
tuple<string,bool> read( handle h)
{
  return make_tuple(s,b);
}

string s; bool valid;
tie(s,valid) = read( db);
if( valid)
  doit( s);
```

C++14

```
auto read( handle h)
{
  return make_tuple(s,b);
}
```

automaticky podle typu

must be visible for type deduction

unpacking

```
auto read( handle h);

auto rs = read( db);
f( rs);
if( get<1>(rs)) ....
```

result forwarding

element access

structured bindings

no declaration needed

```
auto f(....) { .... return {a,b,c}; }
auto [x,y,z] = f();
auto [a,b,c] = { 1, 2, 3};
```

C++17

Structured bindings

C++17

- structured binding declaration
  - like `tie()` without having to have variables already defined
  - applicable to `tuple`, `pair`, `array`, user defined
- declared variables bound to initialized subobjects
  - uniquely-named variable
    - referencable
  - copy elision
    - RVO / NRVO

```
tuple_size<>
get<N>()
tuple_element
```

```
auto    [ identifier-list ]
auto &  [ identifier-list ]
auto && [ identifier-list ]
```

↔

```
= expression ;
{ expression } ;
( expression ) ;
```

- array, vector, ..., pair, tuple, struct

```
auto ω = f();
using x = ω[0];
using y = ω[1];
```

alias
no indirection

initialized variable ω
x ≡ ω[0], y ≡ ω[1]

```
int a[] {1,2};
auto f() -> int(&)[2]
   { return a; }

auto [x,y] = f();
auto& [xr, yr] = f();
```

xr ≡ a[0]
yr ≡ a[1]

```
for( auto&& [key,value]:mymap)
```

postupně nabývá hodnot toho kontejneru

Structured bindings

tohle se zavolá už na místě volání funkce f.

	auto [x,y] =	**auto&** [x,y] =	**auto&&** [x,y] =
auto f() { **return** **tuple** {1,2}; }	direct constructor	✗	direct constructor
auto g() { tuple a {1,2}; **return a**; }	local constructor **move constructor** local destructor	✗	local constructor **move constructor** local destructor
a[]{1,2}; � { a };	copy-constructor	∅	∅
a[] {1,2}; auto h() -> int(**&**)[2] { return **a**; }	copy-constructor	∅	∅
for(� [x,y] : **mmap**)	copy-constructor	∅	∅

C++17: RVO mandatory

NRVO non-mandatory

do not copy redundantly

Range-based for

```
map<string,int> m;
```

- C++03

```
for (map<string,int>::const_iterator i = m.begin(); i != m.end(); ++i)
 if( i->first == "Me")
       cout << i->second;
```

what is more readable?

- C++11

```
for (auto&& i : m) // a lot of space for my own valuable comments :-)))
 if( i.first == "Me")
       cout << i.second;
```

what is more readable?

- C++17

```
for (auto&& [key, value] : m)                    // Structured Binding
    if( key == "Me")
        cout << value;
```

~~x.second~~
~~x.get<3>~~

Range-based for

Idiom – use it!

auto&& i : x	applicable everywhere

r-value, l-value, const, proxy, ...

!! e.g. proxy
vector<bool>
temporal object

auto i : x	copy of each object!
auto& i : x	read/write
const auto& i : x	read-only
explicittype i : x	cast *really* needed, copy

-> pokud je to char/int, tak je to jedno. Třeba ale už std::string<> by to celé zpomalil...

- supported data structures
  - x[], std:: containers
  - user-defined containers
    - to be implemented:
      begin(), end(), iterator: * != ++

Range-based for

proxy
temporal object

```
vector<bool> v;
for( auto& b : vb)
    b = true;
```

cannot bind non-const
r-value ref to l-value

```
class bitfield {
    class proxy {
        operator bool() const { return (base_ >> i_) & 1; }
        proxy& operator=(bool b) const { base_ = ...; }
    private:
        uint_least64_t & base_;
        size_t i_;
    };
    class iterator {
        proxy operator*() { return proxy( base_, i_); }
    };
    proxy operator[](size_t i) { return proxy{ base_, i }; }
    iterator begin() { return iterator(base_, 0); }
    iterator end() { return iterator(base_, size_); }
private:
    uint_least64_t base_ = 0;
};
```

```
vector<bool> v;
for( auto&& b : vb)
    b = true;
```

r-value ref
ok

Generalized range-based for

- C++ 11/14
  - same type for begin_expr and end_expr

- C++17 – generalized range-based for
  - **different** types of begin_expr / end_expr
  - end_expr does not have to be an iterator
  - begin_expr / end_expr must be **comparable**
    - e.g. **predicate** as a range delimiter
  - support for ranges [C++20]

Můžou být jiného typu

```
for( range_decl : range_expr) {..}
```

C++11,14

```
auto begin = begin_expr ,
     end = end_expr;
```

C++17

```
auto && range = range_expr;
auto begin = begin_expr ;
auto end = end_expr ;
for( ; begin!=end; ++begin) {..}
```

predicate

jen musí implementovat operátor !=

– můžu si to specificky vyrobit

Generalized range-based for

```
class sentence {
public:
    struct const_iterator {
        char operator*() const { return sentence_[index_]; }
        void operator++() { ++index_; }
    private:
        const sentence& sentence_;
        size_t index_;
    };
    struct end_iterator {
        char sep_;
    };

    sentence( const string& s, char sep) : s_(s), sep_( sep) {}
    char operator[](size_t i) const { return s_[i]; }
    const_iterator begin() { return const_iterator{ *this}; }
    end_iterator end() { return end_iterator( sep_); }
private:
    string s_;
    char sep_;
};
```

iterátor musí umět iterovat a přistupovat na položku

```
sentence x{ "Sunny afternoon. I ♥ C++", '.' };
for (auto&& c : x)
    cout << c;
```

Tohle je to jediné, co potřebuji umět...

```
bool operator!=(const const_iterator& lhs, const end_iterator& rhs)
{ return *lhs != rhs.sep_; }
```

Range-based for

- when not to use
  - iteration other than begin⇝end

→ tady bych mohl vyskočit

```
for( auto it = k.begin(); it != k.end(); it += 2) ;
```

  - access to more elements

```
for( auto it = k.begin() + 1; it != k.end(); ++it)
    diff = *it - *(it - 1);
```

  - simultaneous iteration of multiple containers

```
auto it1 = k1.begin();
auto it2 = k2.begin();
for( ; it1 != k1.end() && it2 != k2.end(); ++it1, ++it2)
    sum = *it1 + *it2;
```

→ nevytvořím nějakou strukturu pro více iterátorů najednou

- when to use
  - common cases – 95%
  - idiomatic
  - structured bindings

auto – be careful!

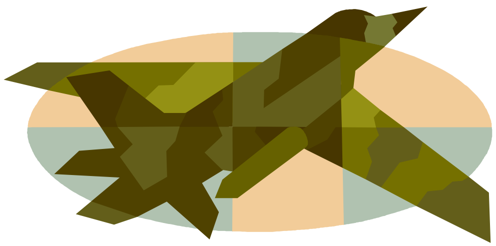
- exact type
  - C++03

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

  - C++11

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

any problem?



... why?? where??

auto – be careful!

deque umí chodit do záporu.

- exact type
  - C++03

size_t : unsigned

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

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

tady si hrnu na druhý konec vesmíru...

- 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!

-> tohle ale umí být 64-bit (můžu mít v kolekci více indexů než co se vejde jen do 32 bit)

-> podle definice je 0 ≐ int

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

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

int i

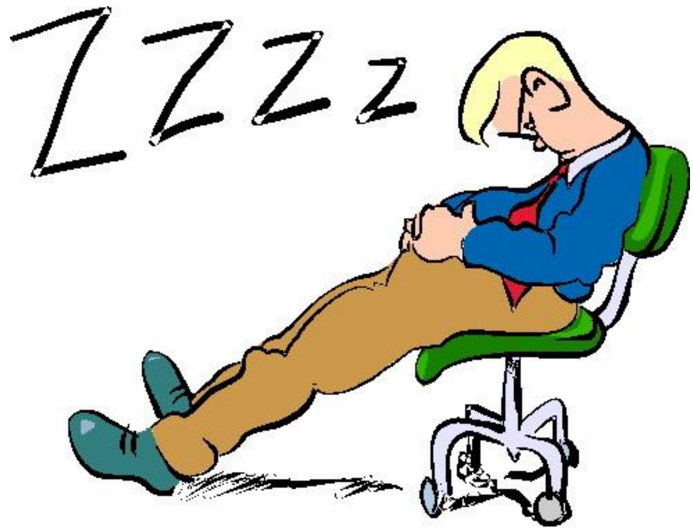
wide type comparison

size_t
64-bit architectures:
unsigned long long

narrow type arithmetics

any problem?

takže ten cyklus poběží donekonečna, protože ten int bude vždy menší, když je size_t větší jak 32-bit.



Type deduction

Type inference

decltype

—> je to součástí jazyka, runtimově nedělá nic, umí pouze typ za kompilace

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

- **decltype** Nejlepší pro šablonované programování...
  - 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) { .. }
```

compile time

- usage

-> pokud třídím kolekci, tak T je typ kolekce, třeba std::vector

ošetření proxy

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

no code generated

T::value_type

template ...

expression not evaluated

vytvořím si kontejner a sáhnu si na něj.

Nevyhodnívá se!

actually kolekce často mají funkci value_type

Return value syntax

- return type specification to the end of declaration

```
auto add( int x, int y) -> int;
```

uhmm .. what's it good for ?? 😐

- inside class scope

```
class Srt {
  enum Type { TA, TB };
  Type getType();
};
Srt::Type Srt::getType() {}
```

redundant scope duplication

```
class Srt {
  enum Type { TA, TB };
  Type getType();
};
auto Srt::getType() -> Type {}
```

tady už vím, že jsem součástí Srt namespace

```
tav::dh_list::bob_hell_type<tav::dh_list::bob_hell_list_trait>::Type
tav::dh_list::bob_hell_type<tav::dh_list::bob_hell_list_trait>::getType(){}
```

- templates

```
template <typename Builder>
auto doit( const Builder& b) -> decltype( b.create())
{ auto val = b.create();
  return val;
}
```

type deduced from parameters

Return value syntax

- templates – combinations of parameters

```
template <typename T, typename U>
?? add( T t, U u);
```

T = signed char
U = unsigned char

→ **int** add

```
template <typename T, typename U>
decltype((*(T*)0)+(*(U*)0)) add( T t, U u);
```

what do you prefer?

```
template <typename T, typename U>
auto add( T t, U u) -> decltype(t+u);
```

Function return type

- automatic deduction of a return type from return expression
  - C++11 – limited: only simple lambdas
  - C++14/17 – extensive: all lambdas, all functions

```
auto add( int x, int y);
decltype(auto) add( int x, int y);
```

- auto – use **template** type deduction rule
- decltype(auto) – use **decltype** deduction rules

different rules!

→ do tohohle můžu zapisovat

```
auto f() {
  vector<int> v;
  return v[i];
}
```

int

```
decltype(auto) g() {
  vector<int> v;
  return v[i];
}
```

int&

```
vector<int> v;
decltype(auto) g( size_t i) {
  return v[i];
}

f(0) = 1;
```

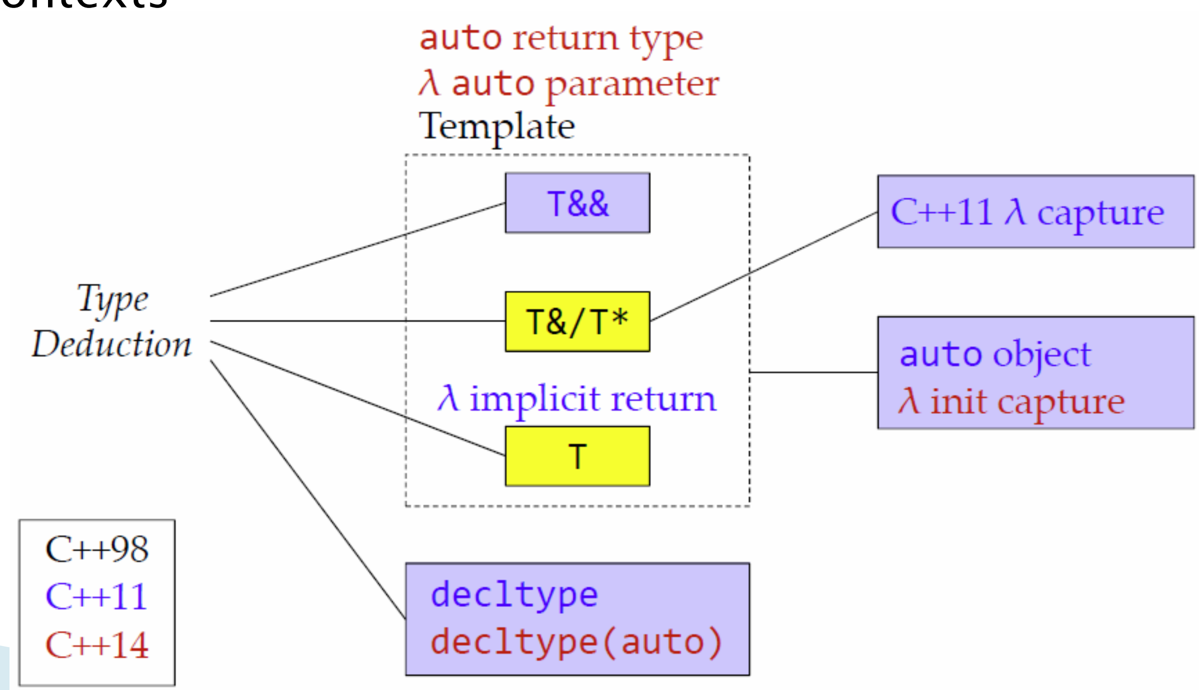
- ... wtf ... why ???
  - ⇝ type deduction rules!

Type deduction

expected knowledge:
forwarding references

Klaus Iglberger:
Back to Basics: Move Semantics (part 2)
www.youtube.com/watch?v=pIzaZbKUw2s

- C++98/03
  - **FTAD,** used only for templates
  - *just works*
  - detailed understanding rarely needed
- C++11 – scope expands
  - **auto** variables, universal references, lambdas, **decltype**
  - no more *just works*, different sets of rules!
- C++14/17/20 – scope expands further
  - **CTAD, function return types,** lambda init captures
  - same rulesets, more usage contexts
- type deduction rules
  - **template** – T, T&/T*, T&&
    - auto objects
    - CTAD
  - **decltype**



Template type deduction

```
template< typename T>
void f( PT p);

f( expr);   // type: ET
```

function template
C++98 – FTAD

PT může být T s ozdobami (const, & ...)

Scott Meyers:
C++ Type Deduction
and Why You Care
vimeo.com/97344493

- **function** template
  - PT is often different from T (e.g., **const** T&)
- **implicit** instantiation
  - ET – type of expr
  - deduction of **PT** and **T** from **ET**
- cases of PT:
  - T&/T* – reference or pointer
  - T&& – universal reference
  - T – by-value

C++17: CTAD ↬

```
pair p{2, 4.5};
tuple t{4, 3, 2.5};
```

```
auto p = make_pair(2, 4.5);
auto t = make_tuple(4, 3, 2.5);
```

Reference/pointer deduction

```
template< typename T>
void f( T& p);

f( expr);   // type: ET
```

PT

- rules:
  - if ET is a reference ⇒ **discard** reference
  - match ET to PT and deduce T

typování šablony zahrnuje &

```
int x = 1;
const int c = x;
const int& r = x;

f(x);   // ET: int        ⇝ T: int,       PT: int&
f(c);   // ET: const int  ⇝ T: const int, PT: const int&
f(r);   // ET: const int& ⇝ T: const int, PT: const int&
```

keep constness of f(T&)
const part of T

no &

<= tohle je vzaný z definice f,
nikoliv z ET.

- behavior of pointers as PT is similar

```
..... void f( T* p);
const int* p = &x;
f(&x);   // int*       ⇝ T: int,       PT: int*
f(p);    // const int* ⇝ T: const int, PT: const int*
```

Reference/pointer deduction

```
template< typename T>
void f( const T& p);

f( expr);   // type: ET
```

- rules:
  - if ET is a reference ⇒ **discard** reference
  - match ET to PT and deduce T

```
int x = 1;
const int c = x;
const int& r = x;

f(x);   // ET: int          ⇝ T: int, PT: const int&
f(c);   // ET: const int    ⇝ T: int, PT: const int&
f(r);   // ET: const int&   ⇝ T: int, PT: const int&
```

no const deduced

no surprise so far

Universal (forwarding) references

```
template< typename T>
void f( T&& p);

f( expr);   // type: ET
```

able to capture lvalue or rvalue

idea:
– lvalue -> param is lvalue ref
– rvalue -> param is rvalue ref

- rules:
  - the same behavior as references, except:
  - if **expr** is lvalue with type **ET**, **T** deduced as **ET&**

→ pak se přidá referenčního

the only case where T is deduced to be &

```
int x = 1;
const int c = x;
const int& r = x;

f(x);     // lv - T: int&,           PT: int&
f(c);     // lv - T: const int&,     PT: const int&
f(r);     // lv - T: const int&,     PT: const int&
f(1);     // rvalue - T: int,        PT: int&&
```

& && ⇒ &
perfect forwarding
reference collapsing

PT declared &&
deduced &

rvalue – no special handling

By-value deduction

```
template< typename T>
void f( T p);

f( expr);   // type: ET
```

```
int x = 1;
const int c = x;
const int& r = x;

f(x);     // T,PT: int
f(c);     // T,PT: int
f(r);     // T,PT: int
```

zahodím const

no const deduced

což nevadí, protože tam dochází k vytváření nové proměnné...

- rules:
  - if type of expr is a reference ⇒ discard reference
  - if expr is const or volatile ⇒ discard const/volatile
  - T is (stripped) ET

new object created

- template type deduction
  - general rules

> - reference-ness of arguments is ignored
> - universal reference parameters:
>   lvalue arguments get special treatment
> - by-value parameters:
>   const / volatile is dropped

auto type deduction

- **auto** behaves as **T**
  - except braced initializers ⇝ *later*
- **full declaration** behaves as **PT**
- **auto declaration never deduced to be a reference**
  - & or && have to be added

většinou to „r" je funkce, resp. její return

tohle je důležité !

```
int x = 1;
const int c = x;
const int& r = x;

auto y = 0;          // int
auto y = x;          // int
auto y = c;          // int
auto y = r;          // int

auto& y = 0;         // error - nelze zapisovat
auto& y = x;         // int&
auto& y = c;         // const int& (auto: const int)
auto& y = r;         // const int&

const auto& y = 0;  // const int& - konstanta
const auto& y = x;  // const int& (auto: int)
const auto& y = c;  // const int& (auto: const int)
const auto& y = r;  // const int&

auto&& y = 0;        // int&& - rvalue
auto&& y = r;        // const int& - lvalue
```

by-value
drop &, const

```
class C {
  C() { very heavy }
};

C& f() {
  C c; ....
  return c;
}

auto c = f();
```

!

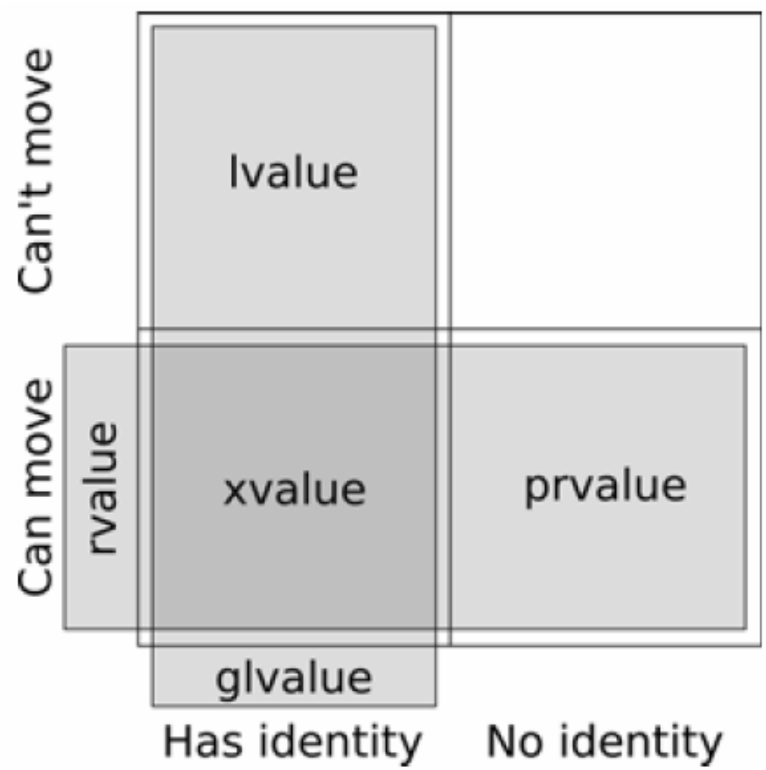
const auto& ≈ PT
auto ≈ T

1. discard &
2. lvalue ⇒ add &
3. && & ⇒ &

value categories

bydlí v paměti

výrazy typu ...= „1+2"



- prvalue
  - pure rvalue
  - pre-C++11 "rvalue"
  - no identity
- lvalue
  - assignable
  - ... ...
- xvalue
  - has identity (*"address"*)
  - expiring value, movable
  - std::move(x), anonymous obj, ...
- rvalue (C++11)
  - prvalue + xvalue
- glvalue
  - generalized lvalue
  - lvalue + xvalue

decltype deduction

- rules:
  - decltype(**name** of type T) ⇒ T
    - **does not** discard const/volatile/&
  - decltype(**prvalue**) ⇒ T
  - decltype(**xvalue_expr** of type T) ⇒ T&&
  - decltype(**lvalue_expr** of type T) ⇒ T&
  - ...

pure rvalue | eXpiring value | assignable

rvalue

```
int x = 1;
const int& r = x;
int a[1];

decltype(1+2);       // int - prv
decltype(move(x));   // int&& - xv
decltype(x);         // int - name
decltype((x));       // !! int& - lv
decltype(r);         // const int&
decltype(a[0]);      // int& - lv
```

decltype vhodný hlavně u šablonového programování...

- **auto** always **discards** an outermost reference
- **decltype** either **preserves** it or even adds one

Function return type deduction

... again

- different rules
  - auto – use **template** type deduction rule
  - decltype(auto) – use **decltype** deduction rules

```
auto f() {
  vector<int> v;
  return v[i];
}
```

int

```
decltype(auto) g() {
  vector<int> v;
  return v[i];
}
```

int&

- lvalue int expression
  - auto ⇒ int
  - decltype(auto) ⇒ int&

```
vector<int> v;
decltype(auto) g( size_t i) {
  return v[i];
}

f(0) = 1;
```

- Beware of the dog!
  - name vs. lvalue expr

```
decltype(auto) h() {
  int x;
  return x;
}
```

name ⇝ int

```
decltype(auto) h() {
  int x;
  return (x);
}
```

lvalue expr ⇝ int&

tady by to vrátilo referenci na lokální proměnnou, co nebude žít mimo scope funkce

Class template argument deduction

- CTAD [C++17]
  - obsoletes many “make” helpers
  - more efficient compilation
  - more efficient generated code

```
pair<int,double> p( 2, 4.5);
auto p = make_pair( 2, 4.5);
```

deduces int, double

C++17

```
pair p{ 2, 4.5};
```

deduces array< string, 3>

```
array x = { "dog"s, "cat"s, "rat"s };
```

```
list<pair<int, string>> x={{0,"z"}, {1,"o"} };
vector y{ x.begin(), x.end()};
```

arg: list<pair<int,string>>::iterator
CTAD: vector<pair<int,string>>

why ?

Stephan T. Lavavej: Class Template Argument Deduction for Everyone
www.youtube.com/watch?v=-H-ut6j1BYU
Timur Doumler: Class Template Argument Deduction in C++17
www.youtube.com/watch?v=UDs90b0yjjQ

Deduction guides

- user defined deduction guides
  - useful for library implementators
  - how to translate
    a set of parameter types
    into template arguments

arg: **list**<pair<int,string>>::**iterator**
CTAD: **vector**<pair<int,string>>

```
list<pair<int, string>> x={{0,"z"}, {1,"o"} };
vector y{ x.begin(), x.end()};
```

```
template<typename T>
struct S { T t; };
S(const char *)
  -> S<string>;

S s{"A String"};
```

string

deduction guide

Ta struktura udělá, že C-string
převede na std::string
a uloží

```
array test{1, 2, 3, 4, 5}; // ⇝ array<int, 5>
```

```
template <class T, class... U>
array(T, U...) -> array<T, 1 + sizeof...(U)>;
```

```
template <typename T> struct MyVec {
  template <typename It> MyVec(It, It){}
};
template <typename It> myVec(It,It) ->
MyVec<typename iterator_traits<It>::value_type>;
set<⌘> s;
MyVec v{ s.begin(), s.end()};
```

deduces MyVec<⌘>

CTAD – What Is This New Acronym All About?
Overload Journal #143
accu.org/index.php/journals/2465

Why CTAD

- much less boilerplate
- much better signal-to-noise
- code much more like 'normal' programming

```
template< class C1, class C2>
f(C1& c1, C2& c2) {
flat_map<
  typename C1::value_type,
  typename C2::value_type,
  less<typename C1::value_type>, C1, C2>
fm(c1,c2);
}
```

```
template< class C1, class C2>
f(C1& c1, C2& c2) {
  flat_map fm(c1,c2);
}
```

⇝ improved maintainability

Deducing this

C++23

C++20

```
class X {
  T& get();
  const T& get() const;
};
```

invoked on const object – const this

&& – invoked on temporary *this

```
class X {
  T& get() &;
  const T& get() const&;
  T&& get() &&;
  const T&& get() const&&;
};
```

C++23

```
class X {
  T& get( this X&);
  const T& get( this const X&);
  T&& get( this X&&);
  const T&& get( this const X&&);
};
```

r-val ref

uni ref!

```
class X {
  template <typename Self>
  auto&& get( this Self&& self);
};
```

```
class X {
  auto&& get( this auto&& self);
};
```

pokud se používá u šablonovaného typu

- code du- (quadri-) plication
- simplifying CRTP-like idioms
  - curiously recurring template pattern
- recursive lambdas

CRTP

- Curiously Recurring Template Pattern
  - compile-time polymorphism

public interface

ugly, old-fasioned

tohle se staticky přetypuje na ten správný typ a pro ten se metoda zavolá.
Vše staticky!

```
template <class Drvd>
class Base { void name() { (static_cast<Drvd*>(this))->impl(); } };
class D1 : public Base<D1> { void impl() { "D1"; } };
class D2 : public Base<D2> { void impl() { "D2"; } };

D1 d1; d1.name();          // D1
D2 d2; d2.name();          // D2
```

implementation

no virtual methods

template

self deduced as Dx

```
class Base { void name( this auto&& self) { self.impl(); } };
class D1 : public Base { void impl() { "D1"; } };
class D2 : public Base { void impl() { "D2"; } };

D1 d1; d1.name();          // D1
D2 d2; d2.name();          // D2
```

C++23

Initializers

initializer_list

- not to be confused with **member** initializer list

inicializace hodnot před spuštěním těla konstruktoru

```
class db {
  db( int id; string nm) : id_(id), nm_(nm) {};
```

- initialization of structures and containers

```
vector<string> v = { "xyz", "plgh", "abraca" };
vector<string> v  ({ "xyz", "plgh", "abraca" });
vector<string> v   { "xyz", "plgh", "abraca" };
```

- template initializer_list
  - initialization of user-defined classes
  - ad-hoc variables or loops

```
for( auto&& x : { 6, 7, 8})
....

auto y = { 9, 15, 21 };
for( auto&& i : y)
....
```

std::initializer_list<int>

```
template <class T> class S {
  vector<T> v;
  S( initializer_list<T> l) : v(l) {}
  void append( initializer_list<T> l)
    { v.insert( v.end(), l.begin(), l.end()); }
};
S<int> s = { 1, 2, 3 };
s.append( { 4, 5});
```

user-defined class template

behives like an ordinary container

initializer_list

- initializer_list is expanded to a local unnamed array

```
class X {
  X( initializer_list<int> v);
};
X x { 1, 2, 3 };
```

→ jen LOKÁLNÍ POLE

```
const int __[] { 1, 2, 3};
X x( initializer_list<int>( __, __+3));
```

- ☺ lightweight object
  - no need to pass const &
  - passing by value doesn't copy the referenced elements

→ by value se předávají jenom začátek a konec pointery

- ☹ initializer_list as a return value
  - references a **local** array!
  - extremely dangerous ☠

Dobrý předávat dovnitř, nikoliv ven.

```
int& f() {
  int x;
  return x;
}
```

ekvivalentní

```
initializer_list<int> f() {
  return { 1, 2, 3 };
}

auto x = f();
```

!!! references local variable

Aggregate initialization

- initialization without explicit constructors

```
class Person { long id; string name; int age; };
Person p = { 87654321, "My Name", 25 };
```

- functionality

  musí se zachovat pořadí typů a argumentů

  - copy-initialization in order of declaration
  - array size deduction
  - |initializes| < |members| ➟ default inicialization of the remaining members
  - static data are ignored

- applicability → fungování jen u jednoduchých tříd...
  - array, struct, class
    - no private or protected non-static data members
    - no virtual member functions
    - no virtual, private, or protected base classes
    - no user-provided constructors
      - explicitly defaulted or deleted constructors are allowed

Designated initilizers

C++20

- struct / class

```
Point p{ .x = 3, .y = 4 };
distance({.x{}, .y{5}}, p);
```

- array

```
enum color{ red, green, blue };
string color_names[] {
   [red] = "red",
   [green] = "green", ...
```

- rules
  - only for aggregate initialization
  - only non-static data members
  - **designators must have the same order of data members in a class declaration**
  - not all data members must be specified
  - no mix of regular initialization with designaters
  - only one designator for a data member

```
struct Date { int month, day, year };
Date ourWedding { 8, 3, 2018 };
```

Oops August or March ??

```
Date ourWedding { .month=8, .day=3, ..
```

```
struct ScreenCfg {
  bool autoScale;
  bool fullscr;
  int bits;
  int planes;
};

ScreenCfg cfg { true, false, 8, 1 };

ScreenConfig playbackCfg {
   .autoScale = true, .fullscr = false,
   .bits = 8, .planes = 1
};
```

Screen ≈ skrin

https://www.cppstories.com/2021/designated-init-cpp20/

Uniform initialization

- uniform initialization – concept
- {} *(braced)* initializer – syntactic construction

- initialization syntax

```
ttt x( init);
ttt z{ init};
ttt y = init;
ttt z1 = { init};
```

direct initialization

copy initialization

  - equivalent in many contexts
  - ... but not everywhere!

- {} initializer ≈ implementation of uniform initialization
  - *"use {} in all possible contexts"*

Uniform initialization

- initial values of non-static data members

```
class T {
  int x{ 0};
  int y = 0;
  int z( 0); // error!
};
```

- uncopyable objects

() – **explicit** constructor

```
UCO x{ 0};
UCO y( 0);
UCO z = 0; // error!
```

zeštíhlující ☺

- prohibits implicit ***narrowing*** conversions
  - init expression is not expressible by the type of the initialized object

```
double a, b, c;
int x( a+b+c);
int y = a+b+c;
int z{ a+b+c}; // error!
```

nejotravnější

- immunity to *most vexing parse*
  - anything that **can** be parsed as a declaration **must** be interpreted as one

```
T x( 0);
T y(); //fnc declaration
T z{};
```

  - ➠ *Prefer braced initialization syntax*

but ...

Constructors and initializer_list

- {} initialization vs. initializer_list vs. constructor overloading
  - {} init **stricly** prefers initializer_list
  - if exists **any way** to use initializer_list, it **will** be used

```
class T {
  T(int i, bool b);
  T(int i, float f);
};

T x(10, true); // T(i,b)
T y{10, true}; // T(i,b)
T z(10, 5.0);  // T(i,f)
T u{10, 5.0};  // T(i,f)
```

```
class T {
  T(int i, bool b);
  T(int i, float f);
  T(initializer_list<float> l);
};

T x(10, true); // T(i,b)
T y{10, true}; // T(l<float>)
T z(10, 5.0);  // T(i,f)
T u{10, 5.0};  // T(l<float>)
```

T((float) 10, (float) true);

definice konverze do floatu (automatické)

```
class T {
  T( const T& t);
  T(initializer_list<double> l);
  operator float() const;
};

T x;
T v( x); // copy constructor
T w{ x}; // T(l<double>) !!!
```

T ⇝ float ⇝ {double}

Constructors and initializer_list

- {} initialization vs. initializer_list vs. constructor overloading
  - {} init **stricly** prefers initializer_list

```
class T {
  T(int i, bool b);
  T(int i, float f);
  T(initializer_list<bool> l);
};

T u{10, 5.0};  // error !!
```

→ vždycky se preferuje initializer_list, někdy je ale nefunkční, protože narrowing

1. initializer_list preferred

2. narrowing conversion prohibited!

```
class T {
  T(int i, bool b);
  T(int i, float f);
  T(initializer_list<string> l);
};

T x(10, true); // T(i,b)
T y{10, true}; // T(i,b)
T z(10, 5.0);  // T(i,f)
T v{10, 5.0};  // T(i,f)
```

→ tady ale není (ani narrowing) however, takže to projde.

OK – no applicable conversion

Empty initializer_list

- no initialization parameters ⇒ default constructor
  - empty list ⇒ initializer_list

```
class T {
  T();
  T(initializer_list<string> l);
};

T x;      // T()
T y{};    // T()
T z();    // declaration!
T u({});  // T(l)
T v{{}};  // T(l)
```

{} means 'no arguments'

most wexing parse

{} means 'empty list'

Consequences

now considered as a **bad** interface design

- vector constructors
  - vector(size_t size, T init_value)
  - vector(initializer_list<T> l)

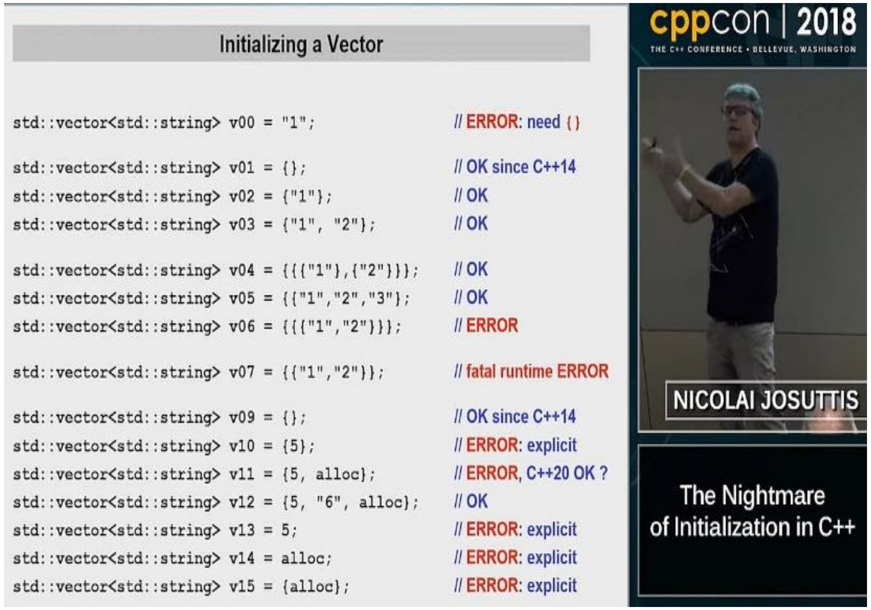
```
vector<int> x( 3, 1); // 1, 1, 1
vector<int> y{ 3, 1}; // 3, 1
```

```
string s( 33, '*'); // "******...**"
string s{ 33, '*'}; // "!*"
```

- assertions
  - lexical analysis

```
assert( c==complex(0,0));   // OK
assert( c==complex{0,0});   // error
assert((c==complex{0,0})); // OK
```

- the thing can be even more complicated
  - mixed explicit / non-explicit constructors
  - default constructor vs. initializer_list
  - initialization of aggregates (struct, array), atomics, ...



Nicolai Jossutis:
The Nightmare of Initialization in C++
www.youtube.com/watch?v=7DTlWPgX6zs

Guidances

- "uniform" is not always uniform
  - still cases of ambiguity or confusion
    - *for a human reader, not for compilers*
- syntax not always intuitive

```
template< typename T> class C {
  C( AnyType x, T y);
```

```
template< typename T> class C {
  C( AnyType x, Tag z, T y);
```

- ... for class authors
  - design your constructors carefully
    - the overload called should **not** be affected by using () or {} initializers

čistě syntaktický trik, aby se odlišily konstruktory

  - design additional overloads carefully
    - initializer_list overshadows other overloads !
    - nonintuitive behavior in many contexts
    - a new constructor overload can alter or even **invalidate** existing code !
- ... for class clients
  - prefer **{}** initialization (without =)
    - some (many) professionals prefer copy-initialization for primitive types
  - use the other kind only if necessary

```
int n = 0;
```

Ways of initialization

	Code	Result
default init	int i1;	undef
copy init	int i2 = 3;	3
direct init	int i3(3);	3
	int i4 = int();	0
	int i5 {3};	3
value init	int i7 {};	0
	int i6 = {3};	3
	int i8 = {};	0
	auto i9 = 3;	3 (int)
direct list init	auto i10 {3};	C++11: init_list<int> C++14: 3 (int)
copy list init	auto i11 = {3};	initializer_list<int>
	auto i12 = int{3};	3 (int)
	int i13();	function decl
	int i14(4,5);	error
	int i15 = (4,5);	5 (int, comma op)
	int i16 = int(4,5);	error
	auto i17(4,5);	error
	auto i18 = (4,5);	5 (int, comma op)
	auto i19 = int(4,5);	error

'=' changes the type!

Initialization

Timur Doumler:
Initialization in Modern C++
www.youtube.com/watch?v=ZfP4VAK21zc

`Type var`	**Default init** `;`	**Copy init** `= value;`	**Direct init** `(args);`	**Value init** `();`	**Empty braces** `{};  = {};`	**Direct list init** `{args};`	**Copy list init** `= {args};`
Built-in types	Uninitialised. Variables w/ static storage duration: Zero-initialised	Initialised with value (via conversion sequence)	1 arg: Init with arg >1 arg: Doesn't compile	Zero-initialised	Zero-initialised	1 arg: Init with arg >1 arg: Doesn't compile	1 arg: Init with arg >1 arg: Doesn't compile
auto	Doesn't compile	Initialised with value	Initialised with value	Doesn't compile	Doesn't compile	1 arg: Init with arg >1 arg: Doesn't compile	Object of type std::initializer_list
Aggregates	Uninitialised. Variables w/ static storage duration: Zero-initialised***	Doesn't compile	Doesn't compile (but will in C++20)	Zero-initialised***	Aggregate init**	1 arg: implicit copy/move ctor if possible. Otherwise aggregate init**	1 arg: implicit copy/move ctor if possible. Otherwise aggregate init**
Types with std::initializer_list ctor	Default ctor	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Default ctor	Default ctor if there is one, otherwise std::initializer_list ctor	std::initializer_list ctor if possible, otherwise matching ctor	std::initializer_list ctor if possible, otherwise matching ctor****
Other types with no user-provided* default ctor	Members are default-initialised	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Zero-initialised***	Zero-initialised***	Matching ctor	Matching ctor****
Other types	Default ctor	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Default ctor	Default ctor	Matching ctor	Matching ctor****

*not user-provided = not user-declared, or user-declared as `=default` *inside* the class definition
**Aggregate init copy-inits all elements with given initialiser, or value-inits them if no initialiser given
***Zero initialisation zero-initialises all elements *and initialises all padding to zero bits*
****Copy-list-initialisation considers explicit ctors, too, but doesn't compile if such a ctor is selected

const ⊗ constexpr ⊗ consteval ⊗ -init

- const
  - object that cannot be changed - logical immutability
  - **can** be evaluated at compile-time
  - integral values can be used as array size, NTTA, ...
  - cannot be applied to functions

```
const int SIZE = 99;
array<int, SIZE>;
```

```
class C {
  int f() const {}
};
```

const this

~~int f() const {}~~

- constexpr [C++11]
  - **compile-time** constant value
  - can be used in constant expressions
  - can be applied to **functions**
    - they **can** be called to produce constant expressions
    - they **can** also be called at **runtime**

```
int x = f();
const int y=x;        // OK
constexpr int z=x;    // ERROR
```

```
constexpr int SIZE = 99;
array<int, SIZE>;
```

```
constexpr int f() {}
```

c-t body

```
constexpr int n = 42;
constexpr const int* p = &n;
```

p is evaluated at c-t

*p = ... is prohibited

const ⊗ constexpr ⊗ consteval ⊗ -init

- consteval [C++20]
  - immediate function
  - applied **only** to **functions**
  - **every** call **must** produce a compile-time constant

```
constexpr int fact( int n) { return n < 2 ? 1 : n*fact(n-1); }
consteval int combination( int m, int n) { return .... }
int a = g();
fact( 4); // OK C-T
fact( a); // OK R-T
```

```
combination( 4, 8);   // OK C-T
~~combination( a, 7);   // ERROR~~
```

- constinit [C++20]
  - mutable
  - forces constant initialization of **static** or thread-local **variables**
  - it can help to limit static order initialization nondeterminism
    - precompiled values and well-defined order
    - rather than dynamic initialization and linking order

```
int f() { .... }
constexpr int g( bool p) { return p ? 42 : f(); }
constinit int x = g( true);        // OK
~~constinit int y = g( false);        // ERROR~~
```

Modules

Headers & include

- separate compilation
  - 50 years old
  - #include
    - textual processing of the source code
    - multiple/redundant compilations
    - lack of isolation
      - headers can change your code
      - your code can change headers
      - headers can change each other
    - separation of declaration and definition
    - one definition rule
    - dependencies
    - ...

```
#include "component.h"

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

...	...
<vector>	<queue>
"other_comp.h"	"mylib.h"
<algorithm>	<algorithm>
<iostream>	<iostream>
<string>	<string>
"component.h"	"component.h"
main.cpp	**component**.cpp

. . .

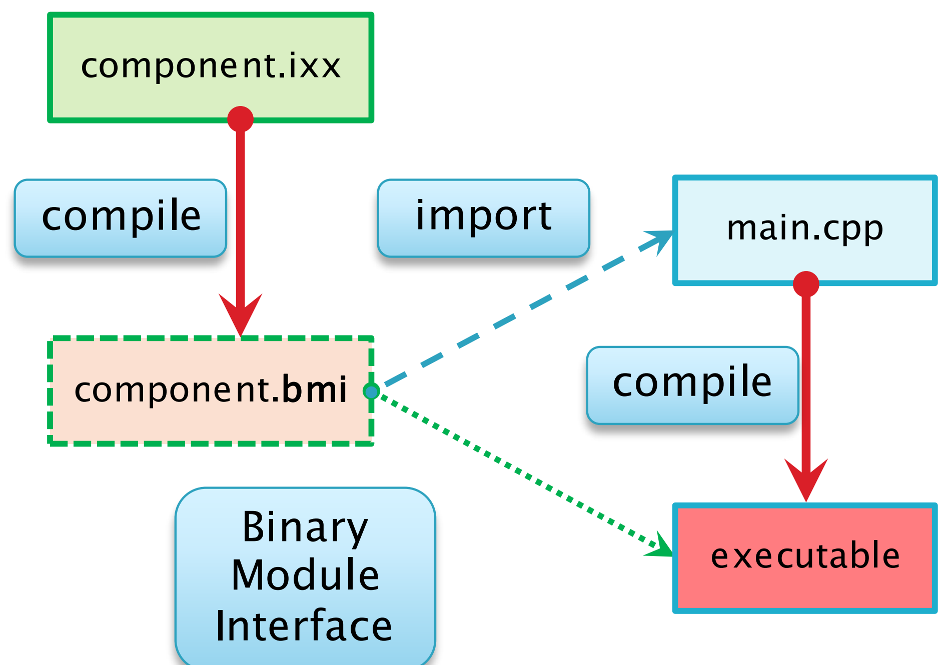
Modules & import

- C++20
  - long awaited
    - coexistence include / import
  - ☺ VS 16.8
    - Gabriel dos Reis
  - 😐 gcc / clang incomplete

```
import component;

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

- Binary Module Interface
  - compiler/platform specific
  - compiler options
- export/import
- build system integration



C++ Modules: A Brief Tour
Overload 2020/10
accu.org/journals/overload/28/159/sidwell

Boris Kolpackov: Modules
CppCon 2019
www.youtube.com/watch?v=szHV6RdQdg8

Simple export / import

mylib.ixx

```
export module mylib;

import std;

export std::string get_text()
{
  return "I am here";
}
```

module declaration

std libraries

exported symbol

context-sensitive keyword

suffix .ixx required by msvc build system

import

consumer.cpp

```
import mylib;

s = get_text();
```

mylib.ixx

```
module;
#include <....>
export module mylib;
import submodule;
export {
  std::string get_text(){..}
  class S { .... };
  ....
}
```

global module fragment

#directives only

module preamble

import only

exported symbols

Templates

```
export module temple;

export template <typename T>
struct foo
{
  T value;
  foo(T const v):value(v) {}
};

export template <typename T>
foo<T> make_foo(T const value)
{
  return foo<T>(value);
}
```

BMI
compiled code

```
import temple;

int main()
{
  auto fi = make_foo( 42);
  cout << fi.value;

  auto fs = make_foo( "modules"s);
  cout << fs.value;
}
```

redundant
compilation
not needed

Partitions

- modules can be split into separate files
- partitions
- internal partitions
  - no export

module : partition

mlc.ixx

```
export module mylib:classes;

export {
  class S { .... };
  ....
}
```

mlf.ixx

```
export module mylib:fnc;

export {
  std::string get_text() {..}
  ....
}
```

mylib.ixx

```
export module mylib;

export import :fnc;
export import :classes;
```

import partition
reexport

whatever.cpp

```
import mylib;

s = get_text();
```

import module

Private partitions

- modules can contain private fragments

```
export module imp;
struct Impl;

export class S {
public:
  void doit();
  Impl* get() { return i_.get(); }
private:
  std::unique_ptr<Impl> i_;
};

module :private;
struct Impl { .... };
```

declaration only

exported class / methods

definition not exported

```
import imp;

int main() {
  S s;
  s.doit();
  auto ip = s.get();

  auto impl = *s.get();
}
```

method – OK

pointer – OK

Error: undefined type

Modules – syntax summary

- **export**$_{(opt)}$ **module** *module-name module-partition*$_{(opt)}$ *attr*$_{(opt)}$;
  - module declaration – declares that the current translation unit is a module unit

- **export** *declaration*
- **export** { *declaration-seq*$_{(opt)}$ }
  - export declaration – export all namespace-scope declarations

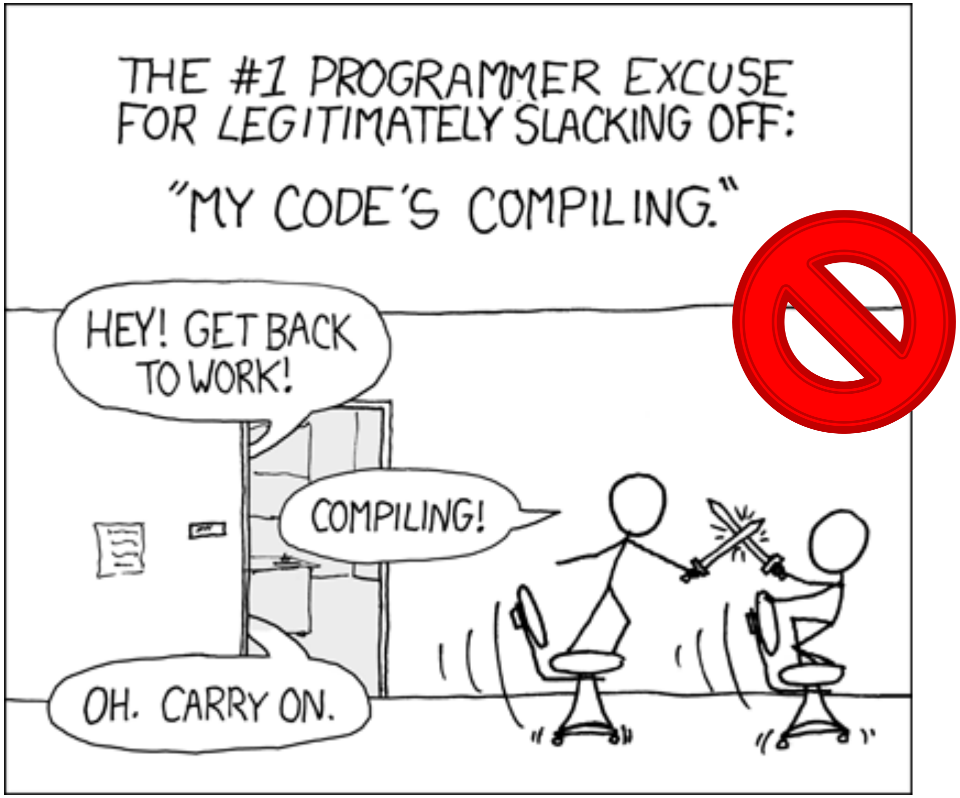
- **export**$_{(opt)}$ **import** *module-name attr*$_{(opt)}$;
- **export**$_{(opt)}$ **import** *module-partition attr*$_{(opt)}$;
- **export**$_{(opt)}$ **import** *header-name attr*$_{(opt)}$;
  - import declaration – import a module unit/module partition/header unit

- **module** ;
  - starts a global module fragment
- **module** : **private** ;
  - starts a private module fragment

Modularization of std libraries

- std:: libraries
  - C++20 not standardized ☹
    - ~~platform specific code~~
  - C++23
    - one big module
    - gcc/clang NYI
    - extension
      - std.compat
      - global namespace
      - contain std

```
import std;
import std.compat;
```



modules **will be** finally useful

	#include	import std	speedup
hello world <iostream>	0.87 s	0.08 s	10.9
mix 9 headers	2.2 s	0.44 s	5
real-world projects			5 – 20

Attributes

Attributes

- language extensions, compiler properties & hints
  - C/C++03: #pragma, __*extkeyword*
- C++11: attributes
  - [[attr]] [[ns::attr(args)]] applicable in many contexts
  - unknown attributes ignored
    - defined by compiler/extensions
    - user-defined attributes not allowed (no C#)
  - should not affect semantics

```
int [[at1]] i [[at2, at3]];
[[at4(arg1, arg2)]] if (cond) {
  [[vendor::at5]] return i;
}
```

- common use (of compiler extensions)
  - optimization, statistics, parallelism support
  - external interface (serialization, database, network, ...)

```
class [[db::object, db::table("people"))]] person
{
  unsigned long id [[db::id, db::type("INT")]];
  string first [[db::column("first_name")]];
};
```

```
int x [[omp::shared]];

[[omp::parallel]] for(;;)
  { .... }
```

Standard attributes

- **[[maybe_unused]]**
  - suppressed warnings

```
void g() {
  [[maybe_unused]] int y;
  assert(y==0);
}
```

- **[[noreturn]]**
  - function never returns
  - _Exit, abort, exit, quick_exit, unexpected, terminate, rethrow_exception, throw_with_nested, nested_exception::rethrow_nested
  - optimalization, warnings – dead code

```
[[noreturn]] void the_end()
{ ....whatever
  exit();
}
obscure_type f()
{ if( smth_wrong)
    the_end();
  else return obscure_type();
}
```

- **[[carries_dependency]]**
  - atomic, memory_order – *later*
- **[[deprecated]]**
- **[[deprecated("reason")]]**
  - correct but deprecated use
- **[[likely]], [[unlikely]]**
  - optimization

```
if( cond) [[likely]] { ....
```

Standard attributes

- **[[fallthrough]]**
  - switch fallthrough
  - otherwise warning if `break` omitted

```
switch (n) {
case 1:
case 2: // no warning
  f();
  [[fallthrough]];
case 3: // warning
  g();
  break;
}
```

- **[[nodiscard]]**
  - warning when return value is not used
  - aplicable to type/function declaration

```
struct [[nodiscard]] mt {};
mt f();
[[nodiscard]] int g();

f(); // warning
g(); // warning
```

- **[[no_unique_address]]**
  - unique address not required

```
struct S {
  char c;
  [[no_unique_address]]
  Empty e1;
};
```

- syntax: [[– always attribute beginning
  - lambda!

syntax error

```
myArray[[]{ return 0; }()] = 1;
```