Modern C++

CODERS.SCHOOL

http://coders.school

- Kamil Szatkowski, kamil.szatkowski@nokia.com
- Łukasz Ziobroń, lukasz@coders.school

About authors

Kamil Szatkowski

- Work at Nokia:
  - C++ software engineer @ CCH
  - C++ software engineer @ LTE CPlane
  - RAIN Developer @ LTE Cplane
  - Code Reviewer
  - Code Mentor
- Trainer:
  - Practial Aspects Of Software Engineering
  - Nokia Academy
  - Internal Nokia trainings
- Occassional speaker:
  - Academic Championships in Team Programming
  - code::dive community
  - code::dive conference

Łukasz Ziobroń

- Work at Nokia:
  - C++ software engineer @ LTE Cplane
  - C++ software engineer @ LTE OAM
  - Python developer @ LTE LOM
  - Scrum Master
  - Code Reviewer
- Trainer:
  - Practial Aspects Of Software Engineering
  - Nokia Academy
  - Coders.school
  - Internal Nokia trainings
- Occassional speaker:
  - Academic Championships in Team Programming
  - code::dive community
  - code::dive conference

Agenda

1. **Language history**
2. Language core novelties
3. New modifiers
4. New constructions
5. Standard library
6. Other C++17 novelties

Introduction to C++ standard

C++ standarization history

- 1998 – first ISO C++ standard
- 2003 - TC1 (“Technical Corrigendum 1”) published as (“C++03”). Bug fixes for C++98
- 2005 - “Technical Report 1” published
- 2011 – ratified C++0x -> C++11
- 2013 – full version of C++14 draft
- 2014 - C++14 published (minor revision)
- 2017 – C++17
- 2020 – C++20?

Introduction to C++ standard

Compilers support

C++17 support

- Full support – gcc7, clang5
- Compiler flag:
  - -std=c++1z
  - -std=c++17

C++14 support

- Full support – gcc5, clang3.4
- Compiler flag:
  - -std=c++1y
  - -std=c++14

More details:

- https://gcc.gnu.org/projects/cxx-status.html
- http://clang.llvm.org/cxx_status.html

Agenda

1. Language history
2. **Language core novelties**
   - **Static assert**
   - Nullptr
   - Using alias
   - Scoped enum
   - Automatic type deduction
3. New modifiers
4. New constructions
5. Standard library
6. Other C++17 novelties

static_assert

Performs compile-time assertion checking. Message is optional from C++17.

```cpp
template <class T>
void swap(T& a, T& b)
{
    static_assert(std::is_copy_constructible<T>::value, "Swap requires copying");
    static_assert(std::is_nothrow_move_constructible<T>::value &&
                  std::is_nothrow_move_assignable<T>::value);
    auto c = b;
    b = a;
    a = c;
}
```

Agenda

1. Language history
2. **Language core novelties**
   - Static assert
   - **Nullptr**
   - Using alias
   - Scoped enum
   - Automatic type deduction
3. New modifiers
4. New constructions
5. Standard library
6. Other C++17 novelties

nullptr

New keyword - *nullptr:*

- value for pointers which point to nothing,
- more expressive and safer than NULL/0 constant,
- has defined type - std::nullptr_t,
- solves the problem with overloaded functions taking pointer or integer as an argument.

nullptr
Examples

```
int* p1 = nullptr;
int* p2 = NULL;
int* p3 = 0;

p2 == p1; // true
p3 == p1; // true

int* p {}; // p is set to nullptr
```

nullptr
Examples

```
void foo(int);

foo(0);       // calls foo(int)
foo(NULL);    // calls foo(int)
foo(nullptr); // compile-time error

void bar(int);
void bar(void*);
void bar(nullptr_t);

bar(0);       // calls bar(int)
bar(NULL);    // calls bar(int) if NULL is 0, ambigous if NULL is 0L
bar(nullptr); // calls bar(void*) or bar(nullptr_t) if provided
```

Agenda

1. Language history
2. **Language core novelties**
    - Static assert
    - Nullptr
    - **Using alias**
    - Scoped enum
    - Automatic type deduction
3. New modifiers
4. New constructions
5. Standard library
6. Other C++17 novelties

Using alias

Type alias is a name that refers to a previously defined type (similar to typedef)

```
using flags = std::ios_base::fmtflags; // equal to typedef std::ios_base::fmtflags flags;
flags fl = std::ios_base::dec;

using SocketContainer = std::vector<std::shared_ptr<Socket>>;
typedef std::vector<std::shared_ptr<Socket>> SocketContainer;
std::vector<std::shared_ptr<Socket>> typedef SocketContainer;
```

Template aliases

Type alias can be templatized:

```
template <typename T>
using StrKeyMap = std::map<std::string, T>;

StrKeyMap<int> my_map; // std::map<std::string, int>
```

Agenda

1. Language history
2. **Language core novelties**
   - Static assert
   - Nullptr
   - Using alias
   - **Scoped enum**
   - Automatic type deduction
3. New modifiers
4. New constructions
5. Standard library
6. Other C++17 novelties

Scoped enums

enum class, enum struct

C++11 enumeration type was extended by a definition of scoped enum type. This type restricts range of defined constants only to defined in enum type and does not allow implicit conversions to integers.

```
enum Colors
{
    RED = 10,
    BLUE,
    GREEN
};

Colors a = RED;
int c = BLUE;
```

```
enum class Languages
{
    ENGLISH,
    GERMAN,
    POLISH
};

Languages d = Languages::ENGLISH;
//int e = Languages::ENGLISH; // Not possible
int e = static_cast<int>(Languages::ENGLISH);
```

Scoped enums
enum-base

In C++11 it is allowed to provide a type specification of enum base type.

```
enum Colors
{
    RED = 10,
    BLUE,
    GREEN
};

std::cout << sizeof(Colors) << std::endl; // size(int) but may be different if GREEN is defined
                                          // as value higher than int can hold

enum Colors : unsigned char
{
    RED = 10,
    BLUE,
    GREEN
};

std::cout << sizeof(Colors) << std::endl; // size(unsigned char)
```

Scoped enums

forward declaration

It is possible to provide a forward declaration for enumeration, which needs to have a base type.

```
enum Colors : unsigned int;

enum struct Languages : unsigned char;
```

Agenda

1. Language history
2. **Language core novelties**
   - Static assert
   - Nullptr
   - Using alias
   - Scoped enum
   - **Automatic type deduction**
3. New modifiers
4. New constructions
5. Standard library
6. Other C++17 novelties

Auto keyword
Type declaration with *auto*

Variable declaration with keyword *auto* allows to automatically deduce a type by compiler.

In previous versions *auto* was used to create automatic variable (created on stack) – noone was using it.

Const and *volatile* modificators can be used when defining an automatic variable, as well as references and pointers.

Typical and convenient usage of auto is to allow a compiler to automatically deduce a type of iterator.

To get const_iterator you need to use methods cbegin() or cend() from the interface of standard containers.

Auto keyword
Examples

```
auto i = 42;                // i : int
const auto *ptr_i = &i; // ptr_i : const int*

double f();
auto r1 = f();              // r1 : double
const auto& r2 = f();   // r2: const double&

std::set<std::string> someStringSet;
auto it = someStringSet.begin();                 // it : std::set<std::string>::iterator
const auto& ref_someStringSet = someStringSet; // ref_someStringSet :
                                                //     const std::set<std::string>&
```

Auto keyword
Examples

```
void do_something(int& x);
void print(const int& x);

std::vector<int> vec = { 1, 2, 3, 4, 5 };

for(auto it = vec.begin(); it != vec.end(); ++it)
{
    do_something(*it);      // it : vector<int>::iterator
}

for(const auto& item : vec) // ok – range-based for
{
    print(item);            // item : const int &
}
```

Auto keyword
Examples

```
const vector<int> values;
auto v1 = values;  // v1 : vector<int>
auto& v2 = values; // v2 : const vector<int>&

volatile long clock = 0L;
auto c = clock;    // c : long

Gadget items[10];
auto g1 = items;   // g1 : Gadget*
auto& g2 = items;  // g2 : Gadget(&)[10] - reference to an array

int func(double) { return 10; }
auto f1 = func;    // f1 : int(*)(double)
auto& f2 = func;   // f2: int(&)(double)
```

Decltype keyword

Type declaration with *decltype*

decltype keyword allows a compiler to deduce a declared type of an object or an expression given as its argument.

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

decltype(coll) coll2;           // coll2 has type of coll
decltype(coll)::key_type val;   // val has type std::string
```

New syntax of function declaration

Function declaration with returned type ->

New, alternative syntax of function declaration allows to declare returned type after the arguments list.

It allows to specify returned type inside function of using function arguments. In combination with decltype, returned type can be provided as an expression using function arguments.

```
int sum(int a, int b);
auto sum(int a, int b) -> int;

template <typename T1, typename T2>
auto add(T1 a, T2 b) -> decltype(a + b)
{
    return a + b;
}
```

Automatic deduction of returned type (C++14)

Deduction with *auto*

In C++14 returned type can be automatically deduced from function imlementation. Deduction mechanism is the same as for automatic deducation of variable types.

If function has many *return* instructions, all of them must return values of the same type.

Recursion for functions with auto return types is possible, only if recursive function call occurs after at least one return statement returning non-recursive value.

Automatic deduction of returned type (C++14)

Examples

```
auto multiply(int x, int y)
{
    return x * y;
}

auto get_name(int id)
{
    if (id == 1)
        return string("Gadget");
    else if (id == 2)
        return string("SuperGadget");
    return string("Unknown");
}
```

```
auto factorial(int n)
{
    if (n == 1)
        return 1;
    return factorial(n-1) * n;
}
```

New rules for direct-list initializations (C++17)

Examples

```
auto x1 = {1, 2};     // std::initializer_list<int>
auto x2 = {1, 2.0};   // error: cannot deduce element type
auto x3{1, 2};        // error: not a single element
auto x4 = {3};        // std::initializer_list<int>
auto x5{3};           // int
```

Agenda

1. Language history
2. Language core novelties
3. **New modifiers**
   - **New function modifiers (default, delete, final, override)**
   - Attributes
   - Noexcept
   - Constexpr expressions
   - Data structure alignment (alignas, alignof)
4. New constructions
5. Standard library
6. Other C++17 novelties

Default, delete, override, final keywords

default

default declaration enforces a compiler to generate default implementation for marked functions (eg. default constructor when other constructors were defined).

You can mark as default only special member functions like: default constructor, copy constructor, copy assignment operator, move constructor (C++11), move assignment operator (C++11), destructor

```
class Gadget
{
public:
    Gadget(const Gadget&); // copy constructor will prevent
                           // generating implicitly declared
                           // default ctor and move operations
    Gadget() = default;
    Gadget(Gadget&&) noexcept = default;
};
```

Default, delete, override, final keywords

delete

delete declaration deletes marked function from the class interface. No code is generated for this function. Calling it, getting its address or usage in *sizeof* causes compilation error.

```
class NoCopyable
{
protected:
    NoCopyable() = default;

public:
    NoCopyable(const NoCopyable&) = delete;
    NoCopyable& operator=(const NoCopyable&) = delete;
};

class NoMoveable
{
    NoMoveable(NoMoveable&&) = delete;
    NoMoveable& operator=(NoMoveable&&) = delete;
};
```

Default, delete, override, final keywords

Prohibiting implicit conversions with *delete*

Marking as delete some of a function overloaded versions helps to avoid implicit convertions.

```
void integral_only(int a)
{
    cout << "integral_only: " << a << endl;
}

void integral_only(double d) = delete;

// ...

integral_only(10);  // OK

short s = 3;
integral_only(s);   // OK - implicit conversion to short

integral_only(3.0); // error - use of deleted function
```

Default, delete, override, final keywords

override

override declaration enforces a compiler to check, if given function overrides virtual function from a base class.

```
struct A
{
   virtual void foo() = 0;
   void dd() {}
};

struct B : A
{
   void foo() override {}   // OK, method overrides in base class
   void bar() override {}   // error, there is no virtual method in struct A
   void dd() override {}    // error, only virtual methods can be overridden
}
```

Default, delete, override, final keywords

Prohibiting inheritance with *final*

final declaration used after a class name does not allow to create a derived class, inheriting from a marked class.

```
struct A final
{
};

struct B : A        // error, cannot derive from class marked as final
{
};
```

Default, delete, override, final keywords

Prohibiting overriding with *final*

final used after virtual function declaration prohibits its override in a derived class.

```
struct A
{
   virtual void foo() const final
   {}

   void bar() const final              // error, only virtual functions can be marked as final
   {}
};

struct B : A
{
   void foo() const override           // error, cannot override function marked as final
   {}
};
```

Agenda

1. Language history
2. Language core novelties
3. **New modifiers**
   - New function modifiers (default, delete, final, override)
   - **Attributes**
   - Noexcept
   - Constexpr expressions
   - Data structure alignment (alignas, alignof)
4. New constructions
5. Standard library
6. Other C++17 novelties

Attributes

Attributes provide the unified standard syntax for implementation-defined language extensions, such as the GNU and IBM language extensions __attribute__((...)), Microsoft extension __declspec(), etc.

Standard attributes:

`[[noreturn]]` – function does never return, like std::terminate. If it does, we have UB

`[[deprecated]] (C++14)` – function is deprecated

`[[deprecated("reason")]] (C++14)` – as above, but compiler will emit the reason

`[[fallthrough]] (C++17)` – in switch statement, indicated that fall through is intentional

`[[nodiscard]] (C++17)` – you cannot ignore value returned from function

`[[maybe_unused]] (C++17)` – suppress compiler warning on unused class, typedef, variable, function, etc.

Attributes

```
[[ noreturn ]] void f() {
throw "error";
// OK
}

[[ noreturn ]] void q(int i) {
if (i > 0)
    throw "positive";
// behavior is undefined if called with an argument <= 0
}

[[deprecated("Please use f2 instead")]] int f1()
{ /* do something */ }
```

Attributes

```
void f(int n) {
  void g(), h(), i();
  switch (n) {
    case 1:
    case 2:
      g();
     [[fallthrough]];
    case 3: // no warning on fallthrough
      h();
    case 4: // compiler may warn on fallthrough
      i();
      [[fallthrough]]; // illformed, not before a case label
  }
}
```

Attributes

```
struct [[nodiscard]] error_info { };



[[maybe_unused]] void f([[maybe_unused]] bool thing1,
                        [[maybe_unused]] bool thing2)
{
   [[maybe_unused]] bool b = thing1 && thing2;
   assert(b); // in release mode, assert is compiled out, and b is unused
              // no warning because it is declared [[maybe_unused]]
} // parameters thing1 and thing2 are not used, no warning
```

Attributes

Attributes for namespaces and enumerators (C++17)

```
enum E {
  foobar = 0,
  foobat [[deprecated]] = foobar
};
E e = foobat; // Emits warning


namespace [[deprecated]] old_stuff{
    void legacy();
}
old_stuff::legacy(); // Emits warning
```

Agenda

1. Language history
2. Language core novelties
3. **New modifiers**
    - New function modifiers (default, delete, final, override)
    - Attributes
    - **Noexcept**
    - Constexpr expressions
    - Data structure alignment (alignas, alignof)
4. New constructions
5. Standard library
6. Other C++17 novelties

Noexcept keyword

1) Specifies whether a function will throw exceptions or not.
2) The *noexcept* operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions. Returns bool.
3) If exception occur in noexcept function, std::terminate is called

```
void bar() noexcept(true) {}
void baz() noexcept { throw 42; }
// noexcept is the same as noexcept(true)

int main()
{
   bar(); // fine
   baz(); // compiles, but calls std::terminate
}
```

```
void may_throw();
void no_throw() noexcept;

int main()
{
  std::cout << std::boolalpha
            << "Is may_throw() noexcept? "
            << noexcept(may_throw()) << '\n' \\false
            << "Is no_throw() noexcept? "
            << noexcept(no_throw()) << '\n'; \\true
}
```

Noexcept keyword

Since C++17 exception specification is a part of the type system. Below functions are functions of two distinct types:

- `void f() noexcept(true);`
- `void f() noexcept(false);`

This change strengthens the type system, e.g. by allowing APIs to require non-throwing callbacks.

Agenda

1. Language history
2. Language core novelties
3. **New modifiers**
   - New function modifiers (default, delete, final, override)
   - Attributes
   - Noexcept
   - **Constexpr expressions**
   - Data structure alignment (alignas, alignof)
4. New constructions
5. Standard library
6. Other C++17 novelties

Constexpr

C++11 introduces two meanings of constants:

- constexpr - constant evaluated during compile time
- const - constant, which value can not change

Constant expression (*constexpr*) is evaluated by compiler during compilation. It can not have values which are not known during compilation and can not have any side effects.

If constant expression can not be computed during compilatation, compiles will raise an error.

```
int x1 = 7;
constexpr int x2 = 7;

constexpr int x3 = x1; // error: initializer is not a contant expression
constexpr int x4 = x2; // OK
```

Constexpr

Constexpr variables

In C++11 constexpr variables must be initialized with constant expression.

Important: const does not need to be initalized with constant expression.

```
constexpr int x = 7;
constexpr auto prefix = "Data";
constexpr int n_x = factorial(x);
constexpr double pi = 3.1415;
constexpr double pi_2 = pi / 2;
```

Constexpr functions

Examples in C++11

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

template <typename T, size_t N>
constexpr size_t size_of_array(T (&)[N])
{
    return N;
}

// ...

const int SIZE = 2;
int arr1[factorial(1)];
int arr2[factorial(SIZE)];
int arr3[factorial(3)];
int arr4[factorial(size_of_array(arr3))];
```

Constexpr functions
constexpr in C++14

In C++14 constexpr restrictions were relaxed. Every function can be marked as constexpr, unless it:

- uses static or thread_local variables,
- uses variable declarations without initializations,
- is virtual,
- calls non-constexpr functions,
- uses non-literal types (values unknown during compilation),
- uses ASM code block,
- has try-catch blocks or throws exceptions

Constexpr functions

Examples

```
constexpr int foo(int bar)
{
    if(bar < 20)
    {
        return 4;
    }

    int k = 5;
    for(int i = 0; i < 54; ++i)
    {
        bar++;
    }

    if(bar > 51)
    {
        return bar + k;
    }

    return 1;
}
```

Constexpr functions

Examples

```
struct Point
{
constexpr Point(int x_, int y_)
   : x(foo(x_)), y(y_)
{}

int x, y;
};

constexpr Point a = { 1, 2 };
```

Constexpr if (C++17)
Examples

In C++17 if expressions can be evaluated at compile time. If condition is evaluated to true, only the first branch of code is generated and the other part is discarded. Otherwise the other part is generated and first part is discarded.

```
if constexpr (cond)
    statement1;
else if constexpr (cond)
    statement2;
else if constexpr (cond)
    statement3;
else
    statement4;
```

Agenda

1. Language history
2. Language core novelties
3. **New modifiers**
    - New function modifiers (default, delete, final, override)
    - Attributes
    - Noexcept
    - Constexpr expressions
    - **Data structure alignment (alignas, alignof)**
4. New constructions
5. Standard library
6. Other C++17 novelties

Alignas keyword

The *alignas* specifier may be applied to:

- the declaration of a variable or a class data member
- the declaration or definition of a class/struct/union or enumeration.

alignas(expression) – expression needs to be positive power of 2.

alignas(type-id) - equivalent to *alignas(alignof(type-id))*

alignas(0) has no effect

Exception: if *alignas* would weaken the alignment the type would have had without this alignas, it will not be applied.

```
// every object of type sse_t will be aligned to 16-byte boundary
struct alignas(16) sse_t
{
  float sse_data[4];
};

// error: requested alignment is not a positive power of 2 alignas(129) char cacheline[128];
alignas(129) char cacheline[128];
```

Alignof keyword

The *alignof* specifier returns a value of type std::size_t, which is alignment in bytes. If the type is reference type, the operator returns the alignment of referenced type; if the type is array type, alignment requirement of the element type is returned.

```cpp
#include <iostream>
using namespace std;

struct Foo {
    int   i;
    float f;
    char  c;
};

struct Empty {};

struct alignas(64) Empty64 {};

struct alignas(1) Double {
    double d;
};
```

```cpp
int main()
{
    cout << "Alignment of" << endl
    << "char: "      << alignof(char)    << endl  // 1
    << "pointer: "   << alignof(int*)    << endl  // 8
    << "class Foo: " << alignof(Foo)     << endl  // 4
    << "Empty: "     << alignof(Empty)   << endl  // 1
    << "Empty64: "   << alignof(Empty64)<< endl  //64
    << "Double: "    << alignof(Double) << endl; // 8
}
```

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. **New constructions**
    - **Unified variable initialization**
    - Move semantics
    - Smart pointers
    - Delegating constructors
    - Lambda expressions
    - Variadic templates
5. Standard library
6. Other C++17 novelties

Uniform variable initialization
Use of {} braces to initialize variables

C++11 introduced possibility to initialize variable with {} braces.

It allows to avoid many problems known from C++98 such as:

- most vexing parse,
- no possibility to initialize containers with list of values,
- different methods for initializing variables of simple types, complex types, structures and arrays.

All methods for initialization of variables from C++98 are correct excluding type narrowing implicit conversion in initialization list.

Uniform variable initialization

Examples

```
int i;          // undefined value

int va(5);      // c++98: "direct initialization", v = 5
int vb = 10;    // c++98: "copy initialization", v = 10
int vc();        // c++98: "function declaration", common error named
                // "most-vexing-parse", compiles normally, but generally
                // this behaviour is not expected

int vd{};       // c++11: brace initialzation – default value
int ve{5};      // c++11: brace initialzation

int values[] = { 1, 2, 3, 4 };   // c++98: brace initialization

struct P { int a, b; };
P p = { 20, 40 };             // c++98: brace initialization
```

Uniform variable initialization

Examples

```
std::complex<float> ca(12.0f, 54.0f);  // c++98: initialization of classes
                                       // using constructor
std::complex<float> cb{12.0f, 54.0f};  // c++11: brace initialization, using
                                       // the same constructor as above

std::vector<std::string> colors;       // c++98: no brace initialization like with
colors.push_back("yellow");            // simple arrays/structs
colors.push_back("blue");

std::vector<std::string> names = {     // c++11: brace initialization with
  "John",                              // std::initializer_list
  "Mary"
};
std::vector<std::string> names{        // c++11: brace initialization with
  "John",                              // std::initializer_list
  "Mary"
};

int array[] = { 1, 2, 5.5 };           // C++98: OK,
                                       // C++11: error - implicit type narrowing
```

Intializing non-static variables in class

brace-or-equal initializer

In C++98 class variables could be initialized only on initializer list of constructor or in its body. The exception existed only for static, integer constants.

Since C++14 it is possible to initialize all variables and constants in class body. Such initialization defined default values for class fields but they can be overwritten in initializer list of constructor or in its body.

Intializing non-static variables in class

Example

```
class Foo
{
public:
   Foo()
   {}

   Foo(std::string a) :
      m_a(a)
   {}

   void print()
   {
      std::cout << m_a << std::endl;
   }

private:
   std::string m_a = "Fooooo";        // C++98: error, C++11: OK
   static const unsigned VALUE = 20u; // C++98: OK, C++11: OK
};

Foo().print();         // Fooooo
Foo("Baar").print();   // Baar
```

Initialization with use of initialization list

std::initializer_list

In C++98 initialization with use of initialization list was possible only for arrays and POD structures (Pure Old Data).

In C++11 this syntax was extended also for class object with use of special class template - std::initializer_list.

std::initializer_list utilizes copy semantics so once value is put on such list it cannot be moved frome there somewhere else (e.g. std::unique_ptr cannot moved from such list).

std::initializer_list has some auxiliary functions: size(), begin()/end().

Constructors that has std::initialize_list as parameter has higher priority over others.

Initialization with use of initialization list

Example

```
template<class Type>
class Bar
{
public:
   Bar(std::initializer_list<Type> values)
   {
      for(auto a : values)                // only example, can be much better
      {
         m_values.push_back(value);
      }
   }

   Bar(Type a, Type b) :
      m_values{a, b}
   {}

private:
   std::vector<Type> m_values;
};


Bar<int> b = { 1, 2 };                                        // OK, first constructor is used
Bar<int> b = { 1, 2, 5, 51 };                                 // OK, first constructor is used
Bar<std::unique_ptr<int>> c = { new int{1}, new int{2} };     // error - std::unique_ptr is non-copyable
```

Aggregate initialization of classes with base classes (C++17)

Example

An aggregate is an array or a class with:

* no user-provided constructors (including those inherited from a base class),

* no private or protected non-static data members,

~~* no base classes and *// removed now!*~~

* no virtual functions and

* no virtual, private or protected base classes

```
struct base { int a1, a2; };
struct derived : base { int b1; };

derived d1{{1, 2}, 3};      // full explicit initialization
derived d1{{}, 1};          // the base is value initialized
```

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. **New constructions**
   - Unified variable initialization
   - **Move semantics**
   - Smart pointers
   - Delegating constructors
   - Lambda expressions
   - Variadic templates
5. Standard library
6. Other C++17 novelties

Move semantics
Advantages and novelties

Better performance from recognition of temporary objects and ability to move variables from them instead making copies (mostly deep copies).

New syntax by introducing *r-value* references (**auto && value**).

New class methods:

- move constructor **Class(Class && src),**
- move assignment operator **Class& operator=(Class && src).**

New auxiliary functions:

- std::move() – forces the use of move constructor or move assignment operator,
- std::forward() – transfer of value forward as is.

Move semantics

l-value vs *r-value*

l-value objects has name and it is possible to get their valid address. State transfer operation of such object can be dangerous because it can be used somewhere else.

r-value objects usually do not have name, it is not possible to get their valid address but transfering state of such object to another one is possible with keeping source in valid state for destruction.

l-value references – they can be bound with l-value object, r-value references and exceptionally const l-value references can be bound with r-value objects. It is not possible to bind temporary objects to l-value references.

r-value references – they can be bound to r-value objects, but it is not possible to bind them with l-value objects and references.

Move semantics
Examples

```
struct A
{
   int a, b;
};

A foo()
{
   return {1, 2};
}

A a;                     // l-value
A & ra = a;              // l-value reference to l-value, OK
A & rb = foo();          // l-value reference to r-value, ERROR
A const& rc = foo();     // const l-value reference to r-value, OK (exception in rules)

A && rra = a;            // r-value reference to l-value, ERROR
A && rrb = foo();        // r-value reference to r-value, OK

A const ca{20, 40};
A const&& rrc = ca;      // const r-value reference to const l-value, ERROR
```

Move semantics

Move constructor and move assignment operator

Both move constructor and move assignment operator are generated automatically by the compiler, just like copy constructor and copy assignment operator.

Default move constructor moves every component of the class.

Default move assignement operator delegates move of every component of the class to such operator defined for this component.

Move semantics

Example of move constructor and move assignment operator

```
struct A
{
   A(A && src) :
       m_value(src.m_value)// only example, can be much better
   {
      src.m_value.reset();
   }

   A & operator=(A && src)
   {
       m_value = src.m_value;    // only example, can be much better
       src.m_value.reset();
       return *this;
   }

   std::shared_ptr<int> m_value;
};
```

Move semantics
New auxiliary functions

std::move() – template class that accepts universal reference. It utilizes reference collapsing and casts this reference to r-value reference. In case of l-value this template will generate an function instance that takes l-value reference and casts it to r-value reference which is the returned from the function.

```
template <typename T>
typename std::remove_reference<T>::type&& move(T&& obj) noexcept
{
    using ReturnType = std::remove_reference<T>::type&&;
    return static_cast<ReturnType>(obj);
}

A a;
A b = std::move(a);  // generates following template instance: A && move(A & obj) noexcept;
```

Move semantics

Example of std::move usage

```
struct A
{
   A(A && src) :
       m_value(std::move(src.m_value))
   {
   }

   A & operator=(A && src)
   {
       m_value = std::move(src.m_value);
       return *this;
   }

   std::shared_ptr<int> m_value;
};
```

Move semantics

New auxiliary functions

std::forward() forwards reference to given variable. It is a function template that just like std::move() use reference collapsing on universal references. This function in contrary to std::move() when given l-value reference will return l-value reference and for r-value reference will return r-value reference.

In other words it performs so called *perfect forwarding* which means it forward given parameter keeping its r-value/l-value nature.

Move semantics

Example of std::forward usage

```
template<class Type>
class Bar
{
public:
   template<class... Args>
   Bar(Args &&... args) :
       m_values(std::forward<Args>(args)...)       // much better
   {}

private:
   std::vector<Type> m_values;
};

Bar<int> b = { 1, 2, 5, 51 };
```

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. **New constructions**
   - Unified variable initialization
   - Move semantics
   - **Smart pointers**
   - Delegating constructors
   - Lambda expressions
   - Variadic templates
5. Standard library
6. Other C++17 novelties

Smart pointers
Mechanism of exceptions vs resources

Using raw pointers for managing resources can cause resource leaks when exception is thrown. In order to secure code from such problem we can use try-catch construction and release them by hand.

Unfortunately in result the code is much less readable and it consists of many code duplication for releasing the resources.

```
void use_resource()
{
    Resource* rsc = nullptr;
    try
    {
        rsc = new Resource();
        rsc->use(); // Code that use rsc can throw an exception
        may_throw();
    }
    catch(...) // Catching all exceptions
    {
        delete rsc;
        throw;
    }
    delete rsc;
}
```

Smart pointers
std::unique_ptr<T>

Template class *std::unique_ptr* is used to ensure the appriopriate release of dynamically given object.

It implements RAII – destructor of smart pointer removes kept object. Object of std::unique_ptr cannot be copied, only move operation is allowed.

Move of the resource is done by utilizing move semantics from C++11 – for l-value references it requires explicit transfer by use of std::move() function template.

Smart pointers

std::unique_ptr<T> - Examples

```
void f()
{
    std::unique_ptr<Gadget> my_gadget {new Gadget()};

    my_gadget->use();  // this code may throw exception

    std::unique_ptr<Gadget> your_gadget = std::move(my_gadget);  // explicit move
}  // Destructor of std::unique_ptr will execute the delete for inside pointer.


// pointers to derived classes
std::unique_ptr<Gadget> pb = std::make_unique<SuperGadget>();  // SuperGadget derives from
                                                               // Gadget
auto pb = std::unique_ptr<Gadget>{ std::make_unique<SuperGadget>() };
```

Smart pointers

std::unique_ptr<T> - Examples

```cpp
auto ptr = std::make_unique<Gadget>(arg); // C++14 ptr: std::unique_ptr<Gadget>

void sink(std::unique_ptr<Gadget> gdgt)
{
    gdgt->call_method();
    // sink takes ownership - deletes the object pointed by gdgt
}

sink(std::move(ptr)); // explicitly moving into sink
```

Smart pointers

std::shared_ptr<T>

Smart pointers with reference counting eliminate the need to explicitely write the code that manages shared resources.

std::shared_ptr is a class template that keeps the pointer to object and counts all references to pointed object.

How it works:

- constructor creates the reference counter and initializes it with 1,
- copy constructor and copy assignment operator increment reference counter,
- destructor decrements reference counter, if value after this operation has value 0, pointed object is released.

Smart pointers

std::shared_ptr<T> - Examples

```cpp
#include <memory>

class Gadget { /* implementation */ };

std::map<std::string, std::shared_ptr<Gadget>> gadgets;  // it wouldn't compile with C++03. Why?

void foo()
{
    std::shared_ptr<Gadget> p1 {new Gadget(1)};  // reference counter = 1
    {
        auto p2 = p1;                            // copying of shared_ptr (reference counter == 2)

        gadgets.insert(make_pair("mp3", p2));    // copying shared_ptr to a std container
                                                 // (reference counter == 3)
        p2->use();
    }                                            // destruction of p2 decrements reference counter = 2
}                                                // destruction of p1 decrements reference counter = 1

gadgets.clear();                                 // reference counter = 0 - gadget is removed
```

Smart pointers

std::make_shared<T> and std::make_unique<T>

Using *std::shared_ptr* eliminates the need to explicitely invoke *delete*, but it doesn't eliminate the use of *new*. It is possible to replace the use of *new* by using special auxiliary function template – *std::make_shared()* which is a factory method for *std::shared_ptr*. This factory method utilizes perfect forwarding to pass all parameters to created object constructor.

Using *std::make_shared()* is also more efficient when using constructor of *std:shared_ptr* and new because it allocated only one memory segment for both the object and control block with reference counters.

There is also *std::make_unique()* function template which was introduced in C++14.

```
auto x = std::make_shared<std::string>("hello, world!"); // std::shared_ptr<std::string>
std::cout << *x << std::endl;


auto ptr = make_unique<Gadget>(arg); // C++14
```

Smart pointers
Application

std::unique_ptr class should be used when:

- exception may be thrown while managing pointers,
- function has many paths of execution and many return points,
- there is only one object that controles life-time of allocated object,
- resitance to exceptions is important.

std::shared_ptr can be used when:

- there are many users of an object but no explicit owner,
- there is no way to implicitely transfer an ownership from and to external library.

std::weak_ptr can be used to:

- break cycles in shared_ptrs
- observe resources

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. **New constructions**
   - Unified variable initialization
   - Move semantics
   - Smart pointers
   - **Delegating constructors**
   - Lambda expressions
   - Variadic templates
5. Standard library
6. Other C++17 novelties

Delegating constructors

Since C++11 you can provide another constructor on constructor's initialization list. This allows to remove code duplications.

```
class Foo {
public:
    Foo() {
        // code to do A
    }
    Foo(int nValue): Foo() { // use Foo() default constructor to do A
        // code to B
    }
};
```

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. **New constructions**
   - Unified variable initialization
   - Move semantics
   - Smart pointers
   - Delegating constructors
   - **Lambda expressions**
   - Variadic templates
5. Standard library
6. Other C++17 novelties

Lambda expressions
Basic lambda expressions

Lambda expression is defined directly in-place of its usage. Usually it is used as a parameter of another function that expects pointer to function or functor – in general a callable object.

Every lambda expression causes the compiler to create unique closure class that implements function operator with code from the expression.

Closure is an object of a closure class. According to way of capture type this object keeps references or copies to local variables.

```
[](){}; // empty lambda

[] { std::cout << "hello world" << std::endl; } // unnamed lambda

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

auto result = l(2, 3); // result = 5
```

Lambda expressions
Basic lambda expressions

If implementation of lambda doesn't contain return statement, the returned type is void.

If implementation of lambda has only return statement, the returned type is a type of used expression.

In every other case returned type must be declared.

It is much better to use lambda expressions to create predicates and functors required by algorithms in standard library (e.g. for std::sort).

```
[](bool condition) -> int
{
    if (condition)
        return 1;
    else
        return 2;
}
```

```
std::array<double, 6> values = { 5.0, 4.0, -1.4, 7.9, -8.22, 0.4 };

std::sort(values.begin(), values.end(), [](double a, double b)
{
    return std::abs(a) < std::abs(b);  // sorting values using
                                       // absolute values
});
```

Lambda expressions
Scope of variables

Inside brackets [] we can include elements that the lambda should capture from the scope in which it is create. Also the way how they are captured can be specified.

- [ ] empty brackets means that inside the lambda no variable from outer scope can be used.
- [&] means that every variable from outer scope is captured by reference, including *this* pointer. Functor created by lambda expression can read and write to any captured variable and all of them are kept inside lambda by reference.
- [=] means that every variable from outer scope is captured by value, including *this* pointer. All variables from outer scope are copied to lambda expression and can be read and written to but with no effect on those captured variable, except for *this* pointer. *this* pointer when copied allows lambda to modify all variables it points to.
- [capture-list] allows to explicitely capture variable from outer scope by mentioning their names on the list. By default all elements are captured by value. If variable should be captured by reference it should be preceded by & which means capturing by reference.
- [*this] (C++17) captures this pointer by value. Anyway, this is implicitly captured by [&] and [=].

Lambda expressions
Scope of variables

```
#include <memory>
int a {5};
auto add5 = [=](int x) { return x + a; };

int counter {};
auto inc = [&counter] { counter++; }


int even_count = 0;
for_each(v.begin(), v.end(), [&even_count] (int n)
{
    cout << n;
    if (n % 2 == 0)
        ++even_count;
});

cout << "There are " << even_count
     << " even numbers in the vector." << endl;
```

Lambda expressions

Generic lambdas (C++14)

In C++11 parameters of lambda expression must be declared with use of specific type.

C++14 allows to declare paramater as *auto* (*generic lambda*).

This allows compiler to deduce the type of lambda parameter in the same way parameters of templates are deduced. In result compiler generates code equivalent to closure class given below.

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


struct UnnamedClosureClass
{
    template <typename T1, typename T2>
    auto operator()(T1 x, T2 y) const
    {
        return x + y;
    }
};

auto lambda = UnnamedClosureClass();
```

Lambda expressions

Lambda capture expressions (C++14)

C++11 lambda functions capture variables declared in their outer scope by value-copy or by reference. This means that value members of a lambda cannot be move-only types.

C++14 allows captured members to be initialized with arbitrary expressions. This allows both capture by value-move and declaring arbitrary members of the lambda, without having a correspondingly named variable in an outer scope.

```
auto lambda = [value = 1]{ return value; };


std::unique_ptr<int> ptr(new int(10));
auto anotherLambda = [value = std::move(ptr)] {return *value;};
```

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. **New constructions**
    - Unified variable initialization
    - Move semantics
    - Smart pointers
    - Delegating constructors
    - Lambda expressions
    - **Variadic templates**
5. Standard library
6. Other C++17 novelties

Variadic templates
Syntax

Templates with variable number of arguments (*variadic template*) use new syntax of parameter pack, that represents many or zero parameters of template.

```
template<class... Types>
class variadic_class
{
/*...*/
};

template<class... Types>
void variadic_foo(Types&&... args)
{
/*...*/
}

variadic_class<float, int, std::string> v;

variadic_foo(1, "", 2u);
```

Variadic templates
Unpacking function parameters

Unpacking group parameters uses new syntax of elipsis operator (...).

In case of function arguments it unpacks them in order given in template function call.

It is possible to call a function on a parameter pack. In such case given function will be called on every argument from a function call.

It is also possible to use recursion to unpack every single argument. It requires the variadic template Head/Tail and non-template function to be defined.

Variadic templates
Example

```
template<class... Types>
void variadic_foo(Types&&... args)
{
   callable(args...);
}


template<class... Types>
void variadic_perfect_forwarding(Types&&... args)
{
   callable(std::forward<Types>(args)...);
}

void variadic_foo() {}

template<class Head, class... Tail>
void variadic_foo(Head const& head, Tail const&... tail)
{
   /*action on head*/
   variadic_foo(tail...);
}
```

Variadic templates
Unpacking template class parameters

Unpacking template class parameters looks the same as unpacking template function arguments but with use of template classes.

It is possible to unpack all types at once (e.g. in case of base class that is variadic template class) or using partial and full specializations.

Variadic templates
Example

```
template<class... Types>
struct Base
{
};

template<class... Types>
struct Derived : Base<Types...>
{
};

template<int... Number>
struct Sum;

template<int Head, int... Tail>
struct Sum<Head, Tail...>
{
   const static int RESULT = Head + Sum<Tail...>::RESULT;
};

template<>
struct Sum<>
{
   const static int RESULT = 0;
}

Sum<1, 2, 3, 4, 5>::RESULT; // = 15
```

Variadic templates
sizeof... operator

sizeof... returns the number of parameters in parameter pack.

```
template<class... Types>
struct NumOfArguments
{
   const static unsigned NUMBER_OF_PARAMETERS = sizeof...(Types);
};
```

Variadic templates

Fold expressions (C++17)

Allows to write compact code with variadic templates without using explicit recursion.

```
template<typename... Args>
auto SumWithOne(Args... args){
    return (1 + ... + args);
}


template<typename... Args>
bool f(Args... args) {
    return (true + ... + args); // OK
}

template<typename... Args>
bool f(Args... args) {
    return (args && ... && args); // error: both operands
                                  // contain unexpanded
                                  // parameter packs
}
```

Operator	Value when param pack is empty
`*`	`1`
`+`	`int()`
`&`	`-1`
`|`	`int()`
`&&`	`true`
`||`	`false`
`,`	`void()`

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. New constructions
5. **Standard library**
   - **Multithreading**
   - Chrono – data & time library
   - Other
6. Other C++17 novelties

Multithreading

C++11 introduces support for multithreading, like:

- Standardized memory model,
- New syntax elements for thread variables,
- Extension of standard library for elements associated with multithreading.

Multithreading
Memory model

C++98/03 does not provide support for multithreading, it means that a try to write/read global variables by two threads simultaneously is not defined (even as *Undefined* Behaviour).

C++11 has defined memory model, which states, that try to write/read global variables by two threads simultaneously is *Undefined Behaviour.*

C++11 has a special type of variables - std::atomic, which specify behaviour when threads try to write into that variables.

Multithreading
New syntax elements

C++11 introduces new keyword - *thread_local*, which allow to define global variables inside one thread. thread_local variable has a tread storage duration.

It means that every thread will have it’s own variable of this types.

Multithreading
Standard library elements

Standard library was extended by number of elements like:

- std::thread,
- std::mutex (and others),
- std::lock_guard, std::unique_lock (and others),
- std::atomic
- std::condition_variable
- std::async, std::future, std::promise, std::packaged_task

Multithreading

Parallelism – expectations vs reality

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. New constructions
5. **Standard library**
   - Multithreading
   - **Chrono – data & time library**
   - Other
6. Other C++17 novelties

Date & time library
chrono

Standard library was extended by number of elements like:

- std::chrono::system_clock::now
- std::chrono::system_clock::time_point
- std::chrono::duration
- std::chrono::nanoseconds
- std::chrono::microseconds
- std::chrono::milliseconds
- std::chrono::seconds
- std::chrono::minutes
- std::chrono::hour

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. New constructions
5. **Standard library**
    - Multithreading
    - Chrono – data & time library
    - **Other**
6. Other C++17 novelties

New elements in standard library

With addition to already mentioned improvements in language, following new elements were introduced into C++ standard library (including elements from std::tr1) :

- <array>, <unordered_map>, <unordered_set>,
- <chrono>,
- <tuple>,
- <regex>,
- <thread>, <mutex>, <condition_variable>, <future>,
- <functional> (major changes),
- <random>
- <type_traits>

Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. New constructions
5. Standard library
6. **Other C++17 novelties**

New elements proposed in C++17

- File system library
- Paralelism library
- std::optional, std::any, std::variant
- Structured bindings

Nested namespace definitions (C++17)

```
namespace A::B::C
{
    ...
}
```

You can use above form rather than below.

```
namespace A
{
namespace B
{
namespace C
{
    ...
}
}
}
```

Class template argument dedution (C++17)

From C++17 class template arguments can be deduced automatically. Automatic template arguments deductions was available earlier only for template functions.

```
std::pair p(1, 'x');                     // C++17: OK, C++14: error: missing
                                         // template arguments before ‘p’

std::pair p<int, std::string>(1, 'x,);   // C++14: OK

auto p = std::make_pair(1, 'x,);         // C++17: OK, C++14: OK
```

Selection statements with initializer (C++17)

New versions of the if and switch statements for C++:

- if (init; condition)
- switch (init; condition)

```
status_code foo() {  // C++14
  {  // variable c scope
    status_code c = bar();
    if (c != SUCCESS) {
      return c;
    }
  }
  // ...
}
```

```
status_code foo() {  // C++17
  if (status_code c = bar(); c != SUCCESS) {
    return c;
  }
  // ...
}
```

Selection statements with initializer (C++17)

```
{
  Foo gadget(args);
  switch (auto s = gadget.status()) {  // C++14
    case OK: gadget.zip(); break;
    case Bad: throw BadFoo(s.message());
  }
}
```

```
switch (Foo gadget(args); auto s = gadget.status()) {  // C++17
  case OK: gadget.zip(); break;
  case Bad: throw BadFoo(s.message());
}
```

Removed elements (C++17)

- trigraphs ??!
- register keyword
- operator++(bool)
- auto_ptr<T> class
- random_shuffle()
- throw() – exception specifier

```
!ErrorHasOccured() ??!??! HandleError();

// Will the next line be executed????????????????/
a++;
```

Trigraph	Equivalent
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

CODERS.SCHOOL

http://coders.school

Things to remember

- Lambda – you need to add mutable in case you have [=] on capture list and you want to modify captured elements
- Lambda – unique_ptr on capture list [a=std::move(a)]
- Delegating constructor – there cannot be anything else on initilization list besides the delegation to another constructor
- Shared_ptr are heavy when copied (atomic counters incrementation). Prefer passing tchem as const shared_ptr<> &
- Prefer using make_shared/make_unique functions to initialize smart_pointers
- Moving is just casting to r-value underneath
- Try marking as many functions as constexpr as possible
- Write override instead of virtual in functions of derived classes