# Modern C++

Piotr Uść, piotr.usc@nokia.com
Paweł Krysiak, pawel.krysiak@nokia.com
2020-03-30



NOKIA

# Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. New constructions
5. Standard library

# Introduction to new C++ standards

## 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 (Committee Draft completed during Prague ISO C++ meeting)
2023 – C++23 (Work In Progress…)

 
NOKIA

# Introduction to new C++ standards
## Compilers support

### C++11 support
Full support - gcc4.8.1, clang3.3
Compiler flag:
- -std=c++0x
- -std=c++11

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

### C++17 support
Full support - gcc7, clang6
Compiler flag:
- -std=c++1z
- -std=c++17

### C++20 support
Partial support ongoing on master branches
Compiler flag:
- -std=c++2a
- -std=c++20

More details:
https://en.cppreference.com/w/cpp/compiler_support

 

# Agenda

1. Language history
2. Language core novelties
   - *nullptr*
   - *using* aliases
   - nested namespaces
   - scoped enums
   - structured bindings
   - automatic type deduction
3. New modifiers
4. New constructions
5. Standard library

# *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
```

 

# *nullptr*

**Task 1.**

Change all NULL to nullptrs

# Agenda

1. Language history
2. Language core novelties
   - *nullptr*
   - *using* aliases
   - nested namespaces
   - scoped enums
   - structured bindings
   - automatic type deduction
3. New modifiers
4. New constructions
5. Standard library

# 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;

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<typename T>
using V = std::vector<T>;

V<int> v;
```

 

NOKIA

# Using alias

**Task 2.**

Change typedef to using alias

 

# Agenda

1. Language history
2. Language core novelties
   - *nullptr*
   - *using* aliases
   - nested namespaces
   - scoped enums
   - structured bindings
   - automatic type deduction
3. New modifiers
4. New constructions
5. Standard library

# Nested namespaces

Simple syntax sugar, useful with deep hierarchy of namespaces.

```
//before
namespace outer {
    namespace middle {
        namespace inner {
            // code...
        }
    }
}

//after
namespace outer::middle::inner {
  // code...
}
```

 

# Agenda

1. Language history
2. Language core novelties
   - *nullptr*
   - *using* aliases
   - nested namespaces
   - scoped enums
   - structured bindings
   - automatic type deduction
3. New modifiers
4. New constructions
5. Standard library

# 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.

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

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

```cpp
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;
```

 

# Scoped enums

**Task 3.**

Write a new scoped enum named Color and define in it 3 colors of your choice. Inherit from unsigned char.

Add a new field: `Color color` in the `Shape` class, so that every shape has it's own defined color.

 

# Agenda

1. Language history
2. **Language core novelties**
   - *nullptr*
   - *using* aliases
   - nested namespaces
   - scoped enums
   - **structured bindings**
   - automatic type deduction
3. New modifiers
4. New constructions
5. Standard library

# Structured bindings

Allows 'unpacking'/binding of subobjects directly into named initializers.
Previously std::tie() provided similar, but inferior behaviour.

```cpp
std::set<int> s = {1, 3, 5, 7, 11};

//before
auto result = s.insert(9);
//result is std::pair<iterator, bool>
if (result.second)
    *result.first;


//after
auto [iter, success] = s.insert(9);
if (success)
    *iter;
```

```cpp
struct Quaternion {
    int x;
    int y;
    int z;
    int w;
};

Quaternion fun();

auto [a, b, c, d] = fun();
```

 

NOKIA

# Structured bindings

Allows 'unpacking'/binding of subobjects directly into named initializers.
Previously std::tie() provided similar, but inferior behaviour.

```
std::set<int> s = {1, 3, 5, 7, 11};

bool success{false};
std::set<int>::iterator iter;

std::tie(iter, success) = s.insert(9);
if (success)
    *iter;
```

 

# Agenda

1. Language history
2. **Language core novelties**
   - *nullptr*
   - *using* aliases
   - nested namespaces
   - scoped enums
   - structured bindings
   - **automatic type deduction**
3. New modifiers
4. New constructions
5. Standard library

# *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>&
```

 



NOKIA

# *Auto* keyword

## Examples

```cpp
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 implementation.

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;
}
```

 
NOKIA

# Auto

**Task 4.**

Use `auto`, wherever you should.

**Task 5.**

Use range-based for loops, wherever possible.

# It's quiz time!

- https://b.socrative.com/login/student/
- Room name: NOKIAPARO
- (multiple choice test)

# Agenda

1. Language history
2. Language core novelties
3. New modifiers
   - new function modifiers (*default*, *delete*, *final*, *override*)
   - attributes
   - *noexcept*
   - *constexpr* expressions
4. New constructions
5. Standard library

# *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;
};
```

 


NOKIA

# *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 from 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
};
```



NOKIA

# *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
{};
```

 
NOKIA

# *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
   {}
};
```

 
NOKIA

# *Default*, *delete, override, final* keywords

**Task 6.**

- Mark copy constructors as `default`.
- Delete `getY()` method in `Square` and all default constructors of shapes

**Task 7.**

- Mark `Circle` class as `final`
- Mark `getX()` in `Rectangle` as `final`. What is the problem?
- Mark all overridden virtual methods. Can you spot the problem?

 

NOKIA

# Agenda

1. Language history
2. Language core novelties
3. **New modifiers**
   - new function modifiers (*default*, *delete*, *final*, *override*)
   - **attributes**
   - *noexcept*
   - *constexpr* expressions
4. New constructions
5. Standard library

# 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]] (C++11)` - function does not 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, indicates that fall through between cases is intentional, silences compiler warning on missing 'break;'

`[[nodiscard]] (C++17)` - issues warning if result of function is unused

`[[nodiscard("reason")]] (C++20)` - as above, but with the reason

`[[likely]], [[unlikely]] (C++20)` - hints for optimization of branching

 

# Attributes

[[noreturn]], [[deprecated]]

```
[[ 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 */ }
```

 

NOKIA

# Attributes

[[fallthrough]]

```
//before
switch (int i) {
    case 0: //intentional fallthrough
    case 2: process(i); break;
    default: print_warning(); break;
}

//after
switch (int i) {
    case 0: [[fallthrough]];
    case 2: process(i); break;
    default: print_warning(); break;
}
```

 

NOKIA

# Attributes

## [[nodiscard]]

```
struct [[nodiscard]] VeryImportant { ... };
// every place which uses VeryImportant as return type will issue warning if discarded

[[nodiscard]] bool launch() { ... }
void f() {
    launch(); // warning on discarded return value
}

void g() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    vec.empty(); // it's not making vector empty! warning will be issued
    bool isEmpty = vec.empty(); // OK
}
```

 

NOKIA

# Attributes

**Task 8.**

Add a new method `double getPi()` in `Circle` class, which returns a PI number. Mark it as deprecated.

 

# Agenda

1. Language history
2. Language core novelties
3. New modifiers
   - new function modifiers (*default*, *delete*, *final*, *override*)
   - attributes
   - ***noexcept***
   - *constexpr* expressions
4. New constructions
5. Standard library

# *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.

```
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
}
```

 


NOKIA

# *Noexcept* keyword

**Task 9.**

Mark some `getArea()` and `getPerimeter()` methods as `noexcept`

 

# Agenda

1. Language history
2. Language core novelties
3. New modifiers
   - new function modifiers (*default*, *delete*, *final*, *override*)
   - attributes
   - *noexcept*
   - *constexpr* expressions
4. New constructions
5. Standard library

# 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 compilation, compiler will raise an error.
In C++11 constexpr variables must be initialized with constant expression.
Important: const does not need to be initalized with constant expression.

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

constexpr int x3 = x1; // error: initializer is not a constant expression
constexpr int x4 = x2; // OK
constexpr int n_x = factorial(x);
```

 

NOKIA

# 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))];
```

 

NOKIA

# 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

 

NOKIA

# 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;
}
```

 
NOKIA

# Constexpr functions

## Examples

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

int x, y;
};

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

 

NOKIA

# Constexpr functions

## constexpr in C++17

Two new features added:

- compile-time if constexpr
- constexpr lambda

 

# Constexpr in C++17
## Examples

```
template<class T> struct dependent_false : std::false_type
{};

template <typename T>
constexpr bool is_integral()
{
    if constexpr (std::is_integral<T>::value)
        return true;
    else
        static_assert(dependent_false<T>::value);
}

static_assert(is_integral<int>());
```

 

NOKIA

# Constexpr

**Task 10.**

Write a function that calculates n-th Fibonacci's number. Do not mark it `constexpr`.

In the first line of `main()` add computing 45-th Fibonacci's number. Measure the time of program execution (time ./modern_cpp)

Mark fibonacci function as `constexpr`, compile the program and measure the time of execution once again.

If you can't see a big difference assign the result to the constexpr variable.

 

# It's quiz time!

- https://b.socrative.com/login/student/
- Room name: NOKIAPARO
- (multiple choice test)

# 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
    - fold expressions
5. Standard library

# Variable initialization in modern C++


 

NOKIA

# 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
```

 

NOKIA

# 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 (Plain 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
```

 

# Uniform variable initialization

**Task 11.**

- Use `initializer_list` to initialize the collection.
- Add a new constructor to Shape - `Shape(Color c)`. What happens?
- Use constructor inheritance to allow initialization of all shapes providing only a `Color` as a parameter. Create some shapes providing `Color` only param.
- Add in-class field initialization for all shapes to safely use inherited constructor.

 

# 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
   - fold expressions
5. Standard library

# 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.

 

NOKIA

# 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

## 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

## 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 };
```

 

NOKIA

# Move semantics

**Task 12.**

- Add move constructors and move assignment operators to all shapes.
- Mark them as `noexcept`.
- What about Rule of 5?
- Move some shapes into the collection.

 

# 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
   - fold expressions
5. Standard library

# 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
   - fold expressions
5. Standard library

# 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
    }
};
```

 

# Delegating constructors

**Task 13.**

- Add a new constructor, which takes also the previously defined Color of a shape. You can use a default parameter for Color.
- Delegate a call in the old constructor to the new one.

 

# 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
   - fold expressions
5. Standard library

# 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
```

 

NOKIA

# Lambda expressions

## Basic lambda expressions

If implementation of lambda doesn't contain return statement, the returned type is void.
If implementation of lambda has return statement(s), the returned type is a type of used expression(s).
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
});
```

 
NOKIA

# 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.

# Lambda expressions

## Scope of variables

```
int a {5};
auto add5 = [=](int x) { return x + a; };
cout << add5(1); // 6

int counter {};
auto inc = [&counter] { counter++; };
inc();  // counter = 1

int even_count = 0;
vector<int> v = {1, 2, 3, 4, 5};
for_each(v.begin(), v.end(), [&even_count] (int n)
{
    if (n % 2 == 0)
        ++even_count;
});
cout << "There are " << even_count // = 2
     << " even numbers in the vector." << endl;
```

 
NOKIA

# 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;};
```

 
NOKIA

# Lambda expressions
## (C++17)

C++17 introduce:

- Constexpr lambda
- Lambda capture of *this

```
constexpr int inc(int n)
{
    return [n] { return n + 1; }();
}

auto inc2 = [](int n) constexpr
            { return n + 1; };


static_assert(inc(1) == 2);
static_assert(inc2(1) == 2);
```

```
struct S2 { void f(int i); };
void S2::f(int i)
{
    [=]{};          // OK: by-copy capture default
    [=, &i]{};      // OK: by-copy, except i by reference
    [*this]{};      // until C++17: Error: invalid syntax
                    // since c++17: OK: S2 by copy
    [=, this] {};   // until C++20: Error: this when = is default
                    // since C++20: OK, same as [=]
}
```

 

NOKIA

# Lambda expressions

**Task 14.**

- Change functions from `main.cpp` into lambdas (`sortByArea`, `perimeterBiggerThan20`, `areaLessThan10`)
- Change lambda `areaLessThan10` into lambda `areaLessThanX`, which takes `x = 10` on a capture list. What is the problem?
- Use `std::function` to solve the problem.

 

# 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**
   - fold expressions
5. Standard library

# 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

**Task 15.**

- Write a factory method which should work like `std::make_shared`.
- It should have below signature:

```
template<class DerivedType, class... Arguments>
std::shared_ptr<Shape> make_shape(Arguments&&... args);
```

- Inside, it should create a `shared_ptr` to DerivedType and pass all arguments into constructor of DerivedType via perfect forwarding.

 
NOKIA

# 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
    - **fold expressions**
5. Standard library

# Fold expressions

Provides very concise way to perform repetitive action over all elements of a parameter pack in order to reduce them into single result.
It's also more performant than previous usage of variadic templates, since it does not use recursion, therefore avoiding costly template instantiations.

```
template<typename... Args>
bool all(Args... args) {
    return (... && args);
}

bool b = all(true, true, true, false);
// within all(), the unary left fold expands as
// return ((true && true) && true) && false;
// b is false
```

 

# It's quiz time!

- https://b.socrative.com/login/student/
- Room name: NOKIAPARO
- (multiple choice test)

# Agenda

1. Language history
2. Language core novelties
3. New modifiers
4. New constructions
5. Standard library
    - new things in standard library in short

# New elements in standard library

C++11

With addition to already mentioned improvements in language, following new elements were introduced into C++ standard library:

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



NOKIA

# New elements in standard library

C++17

With addition to already mentioned improvements in language, following new elements were introduced into C++ standard library:

- <optional>, <variant>, <any>
- <filesystem>
- <string_view>
- <execution>
- <charconv>

# New elements in standard library

C++20

With addition to already mentioned improvements in language, following new elements were introduced into C++ standard library:

- <chrono> (dates, calendars, time zones)
- <concepts>
- <compare> (operator <=>)
- <format>
- <ranges>
- <bit>




NOKIA

# References

- Version with features from cppreference.com (c++11, c++14, c++17, c++20)
- Modern cpp features with examples (nice collection here)
- "*Effective Modern C++*" by Scott Meyers (c++11 & 14)
- godbolt.org - great online compiler and more





NOKIA