MODERN C++ #1

BASIC FEATURES

ŁUKASZ ZIOBROŃ

KAMIL SZATKOWSKI

AGENDA

1. intro (30′)
2. `static_assert` (15′)
3. `nullptr` (10′)
4. scoped enums (30′)
5. ☕ break (15′)
6. `auto` keyword and range-based for loop (1h)
7. ☕ break (10′)
8. `using` alias (15′)
9. uniform initialization (40′)
10. 🍝 lunch break (50′)
11. new keywords: `default`, `delete`, `final`, `override` (1h)
12. recap (15′)

LET'S GET TO KNOW EACH OTHER

- Your name and programming experience
- What you do not like in C++?
- Your hobbies

ŁUKASZ ZIOBROŃ

NOT ONLY A PROGRAMMING XP

- Frontend dev & DevOps @ Coders School
- C++ and Python developer @ Nokia & Credit Suisse
- Team leader & Trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Webdeveloper (HTML, PHP, CSS) @ StarCraft Area

TRAINING EXPERIENCE

- C++ trainings @ Coders School
- Practical Aspects Of Software Engineering @ PWr & UWr
- Nokia Academy @ Nokia
- Internal corporate trainings

PUBLIC SPEAKING EXPERIENCE

- code::dive conference
- code::dive community
- Academic Championships in Team Programming

HOBBIES

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Andragogy

CONTRACT

- 🎰 Vegas rule
- 🗣️ Discussion, not a lecture
- ☕ Additional breaks on demand
- ⌚ Be on time after breaks

LINK TO PRESENTATION ON GITHUB

PRE-TEST

1. WHAT IS THE TYPE OF VARIABLE `v`?

```
int i = 42;
const auto v = &i;
```

1. `const int`
2. `const int&`
3. `const int*`
4. other

2. WHICH OF THE FOLLOWING INITIALIZATIONS ARE VALID IN C++14?

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

1. `int values[] = { 1, 2, 3, 4, 5 };`
2. `P v = { 1, 4 };`
3. `P v{1, 4};`
4. `P v(1, 4);`
5. `std::vector<int> v = { 1, 2, 3, 4 };`
6. `std::vector<int> v(1, 2, 3, 4);`
7. `int v[] = { 1, 3, 5, 6.6 };`

3. WHICH OF THE FOLLOWING ELEMENTS CAN BE DEFINED AS DELETED (`= delete;`)?

1. default constructor
2. copy constructor
3. move constructor
4. copy assignment operator
5. move assignment operator
6. destructor
7. free function
8. class method
9. class member object

C++ STANDARDS

THE HISTORY OF C++ STANDARDIZATION

WHEN C++ WAS CREATED?

1979

- 1998 - first ISO C++ standard - C++98
- 2003 - TC1 (Technical Corrigendum 1) published as C++03. Bug fixes for C++98
- 2005 - Technical Report 1 published (`std::tr1` namespace)
- 2011 - ratified C++0x as C++11
- 2013 - full version of C++1y draft
- 2014 - C++1y published as C++14
- 2017 - C++1z published as C++17
- 2020 - C++2a should be published as C++20

COMPILERS SUPPORT

GCC - CLANG

C++20

- Full support: not implemented yet
- Compiler flags: -std=c++2a

C++17

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

C++14

- Full support: gcc5, clang3.4
- Compiler flags: -std=c++14, -std=c++1y
- Enabled by default since gcc6.1

C++11

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

static_assert

static_assert

```
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_v<T> &&
                  std::is_nothrow_move_assignable_v<T>);
    auto c = b;
    b = a;
    a = c;
}
```

Rationale: Preventing compilation on user defined conditions (usually specific types).

Performs compile-time assertion checking. Usually used with `<type_traits>` library.

The message is optional from C++17.

EXERCISE

Assert that `M_PI` used in `Circle.cpp` file is not equal to `3.14`

```
static_assert(condition, "optional message");
```

nullptr

POINTER COMPARISON

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

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

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

OVERLOAD RESOLUTION

```
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(std::nullptr_t);

bar(0);         // calls bar(int)
bar(NULL);      // calls bar(int) if NULL is 0, may be ambiguous if NULL is 0L
bar(nullptr);   // calls bar(std::nullptr_t) if provided, bar(void*) otherwise
```

`nullptr`

- value for a pointer that points to nothing
- more expressive and safer than `NULL`/`0` constant
- has defined type `std::nullptr_t`
- solves the problem with overloaded functions taking a pointer or an integer as an argument

SCOPED `enum`

STANDARD `enum`

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

Colors a = RED;      // OK
int b = GREEN;       // OK

enum Fruits {
    ORANGE,
    BANANA
};

Colors c = ORANGE;   // 11 or 0?
// Hopefully: error: ‘ORANGE’ conflicts with a previous declaration
```

enum class

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

Languages a = Languages::ENGLISH;
// Languages b = GERMAN;
// int c = Languages::ENGLISH;
int d = static_cast<int>(Languages::ENGLISH);   // only explicit cast allowed
```

Rationale: Stronger and less error-prone enumeration types.

- Introduced in C++11
- Restricts range of defined constants only to those defined in an enum class
- Enum values must be accessed with the enum name scope
- Does not allow implicit conversions, `static_cast` must be used
- `enum class == enum struct`

enum BASE

```cpp
#include <iostream>
#include <limits>

enum Colors    { YELLOW = 10, ORANGE };
enum BigValue { VALUE = std::numeric_limits<long>::max() };
enum RgbColors : unsigned char {
    RED = 0x01,
    GREEN = 0x02,
    BLUE = 0x04,
    // BLACK = 0xFF + 1  // error: enumerator value 256 is outside
};                       // the range of underlying type 'unsigned char'

int main() {
    std::cout << sizeof(Colors) << std::endl;     // 4 - sizeof(int)
    std::cout << sizeof(BigValue) << std::endl;  // 8 - sizeof(long)
    std::cout << sizeof(RgbColors) << std::endl; // 1 - sizeof(unsigned char)
    return 0;
}
```

Change the code in ideone.com

`enum` SIZE

- Default enum size is `sizeof(int)`
- `enum` underlying type is extended automatically if values greater than `int` are provided
- To save some memory we can define the underlying type using inheritance
- A compiler will not allow defining value greater than the defined base can hold
- Inheritance work on both `enum` and `enum class`

`enum` FORWARD DECLARATION

For enums with the defined underlying type, it is possible to provide only a forward declaration, if values do not need to be known.

There will be no need to recompile source file if new enum values are added.

```
enum Colors : unsigned int;
enum struct Languages : unsigned char;
```

EXERCISE

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 can have its own defined color.

Add a default color value in a Shape class.

BREAK

RECAP

WHAT WERE WE TALKING ABOUT BEFORE THE BREAK?

AUTOMATIC TYPE DEDUCTION

`auto` KEYWORD

```
auto a;          // error: declaration of 'auto a' has no initializer
auto i = 42;     // i is int
auto u = 42u;    // u is unsigned
auto d = 42.0;   // d is double
auto f = 42.0f;  // f is float

double f();
auto r1 = f();   // r1 is double

std::set<std::string> collection;
auto it = collection.begin(); // it is std::set<std::string>::iterator
```

Rationale: Not important (but strongly defined) types, less typing, less refactoring.

- A compiler can automatically deduce the type of variable during initialization
- Deduction is made from a literal, other variable or a function return type
- The same rules as for templates deduction are applied

VARIABLE MODIFIERS

```
int func() { return 10; }

int main() {
    const auto& v1 = func(); // v1 is const int&
    const auto v2 = func();  // v2 is const int
    // auto& v3 = func();    // error: cannot bind non-const lvalue reference
                             // of type ‘int&’ to an rvalue of type ‘int’
    auto v4 = func();        // v4 is int
    return 0;
}
```

DEDUCTION RULES FOR REFERENCES

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

volatile long clock = 0L;
auto c1 = clock;    // c1 : long
auto& c2 = clock;   // c2 : volatile long&

Gadget items[10];
auto g1 = items;    // g1 : Gadget*
auto& g2 = items;   // g2 : Gadget(&)[10] - a reference to
                    // the 10-elementh array of Gadgets

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

- Reference means the same object with the same properties
- Reference preserves cv-qualifiers (`const`, `volatile`)
- Copy drops cv-qualifiers
- Copy of array decays to a pointer

FUNCTION DECLARATION WITH ARROW

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


auto isEven = [](int a) -> bool {
    return a % 2;
}
```

Introduced to allow definition of the type returned from lambda functions.

DEDUCTION OF A FUNCTION RETURNED TYPE

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

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

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

- Introduced in C++14
- Deduction mechanism is the same as for deduction of variable types
- All `return` instructions must return the same type
- Recursion allowed only if recursive function call is not a first `return` statement

RANGE-BASED FOR LOOP

```cpp
std::vector<int> v = {0, 1, 2, 3, 4, 5};

for (const auto & element : v) {
    std::cout << element << ' ';
}
std::cout << '\n';
```

GENERATED CODE FOR-RANGE BASED FOR LOOP

```
{
    auto && __range = range_expression ;
    auto __begin = begin_expr ;
    auto __end = end_expr ;
    for ( ; __begin != __end; ++__begin) {
        range_declaration = *__begin;
        loop_statement
    }
}
```

EXERCISE

Put auto wherever you think is good.

Use range-based for loops wherever possible.

LET'S HAVE SOME FUN :)

Connect to my VSC and edit my program simultaneously.

BREAK

RECAP

WHAT WERE WE TALKING ABOUT BEFORE THE BREAK?

`using` ALIAS

TYPE ALIASING

```
typedef std::ios_base::fmtflags Flags;
using Flags = std::ios_base::fmtflags;  // the same as above
Flags fl = std::ios_base::dec;
```

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

Rationale: More intuitive alias creation.

A type alias is a name that refers to a previously defined type. It could be created with typedef. From C++11 type aliases should be created with `using` keyword.

TEMPLATE ALIASES

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

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

Type alias can be parametrized with templates. It was impossible with typedef.

Template aliases cannot be specialized.

CONSTRUCTORS INHERITANCE

```
struct A {
    explicit A(int);
    int a;
};

struct B : A {
    using A::A;  // implicit declaration of B::B(int)
    B(int, int); // overloaded inherited Base ctor
};
```

- Derived class constructors are generated implicitly, only if they are used
- Derived class constructors take the same arguments as base class constructors
- Derived class constructor calls according base class constructor
- Constructor inheritance in a class that adds a new field might be risky - new fields can be uninitialized

EXERCISE

Change a `typedef` to `using` alias.

UNIFORM INITIALIZATION

C++98/03 INITIALIZATION

```
int a;              // undefined value
int b(5);           // direct initialization, b = 5
int c = 10;         // copy initialization, c = 10
int d = int();      // default initialization, d = 0
int e();            // function declaration - "most vexing parse"

int values[] = { 1, 2, 3, 4 };  // brace initialization of aggregate
int array[] = { 1, 2, 3.5 };    // C++98 - ok, implicit type narrowing

struct P { int a, b; };
P p = { 20, 40 };                  // brace initialization of POD

std::complex<double> c(4.0, 2.0);  // initialization of classes

std::vector<std::string> names;    // no initialization for list of values
names.push_back("John");
names.push_back("Jane");
```

C++11 INITIALIZATION WITH {}

```
int a;              // still undefined value
int b{5};           // brace initialization, b = 5
int c{};            // brace initialization, c = 0

int values[] = { 1, 2, 3, 4 };  // brace initialization of aggregate
int array[] = { 1, 2, 3.5 };    // C++11: error - implicit type narrowing

struct P { int a, b; };
P p = { 20, 40 };               // brace initialization of POD

std::complex<double> c{4.0, 2.0};   // brace initialization calls adequate

std::vector<std::string> names = { "John", "Jane" };  // brace initializati
```

Rationale: eliminate problematic initialization cases from C++98, initialization of STL containers, have one universal way of initialization.

IN-CLASS INITIALIZATION OF NON-STATIC VARIABLES

```cpp
struct Foo
{
    Foo() {}
    Foo(std::string a) : a_(a) {}
    void print() { std::cout << a_ << std::endl; }

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

Foo().print();            // Foo
Foo("Bar").print();       // Bar
```

std::initializer_list<T>

```cpp
auto values = {1, 2, 3, 4, 5};   // values is std::initializer_list<int>
std::vector<int> v = {1, 2, -3}; // creates a vector from
                                 // std::initializer_list<int>
```

- Defined in `initializer_list` header
- Elements are kept in an array
- Elements are immutable
- Elements must be copyable
- Have limited interface and access via iterators - `begin()`, `end()`, `size()`
- Should be passed to functions by value

CONSTRUCTOR PRIORITY

```
template<class Type>
class Bar {
    std::vector<Type> values_;
public:
    Bar(std::initializer_list<Type> values) : values_(values) {}
    Bar(Type a, Type b) : values_{a, b} {}
};

Bar<int> c = {1, 2, 5, 51};   // calls std::initializer_list c-tor
Bar<int> d{1, 2, 5, 51};      // calls std::initializer_list c-tor
Bar<int> e = {1, 2};          // calls std::initializer_list c-tor
Bar<int> f{1, 2};             // calls std::initializer_list c-tor
Bar<int> g(1, 2);             // calls Bar(Type a, Type b) c-tor
Bar<int> h = {};              // calls std::initializer_list c-tor
                              // or default c-tor if exists
Bar<std::unique_ptr> c = {new int{1}, new int{2}};
// error - std::unique_ptr is non-copyable
```

C-tor with `std::initializer_list` has greater priority, even if other c-tors match.

EXERCISE

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 a Color only param.

Add in-class field initialization for all shapes to safely use inherited constructor.

LUNCH BREAK

RECAP

WHAT WERE WE TALKING ABOUT BEFORE THE BREAK?

NEW KEYWORDS

`default`, `delete`, `final`, `override`

`default` KEYWORD

```cpp
class AwesomeClass {
public:
    AwesomeClass(const AwesomeClass&);
    AwesomeClass& operator=(const AwesomeClass&);
    // user defined copy operations prevents implicit generation
    // of default c-tor and move operations

    AwesomeClass() = default;
    AwesomeClass(AwesomeClass&&) = default;
    AwesomeClass& operator=(AwesomeClass&&) = default;
};
```

`default` KEYWORD

`delete` KEYWORD

```cpp
class NoCopyable {  // NoCopyable idiom
public:
    NoCopyable() = default;
    NoCopyable(const NoCopyable&) = delete;
    NoCopyable& operator=(const NoCopyable&) = delete;
};

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

`delete` KEYWORD

- `delete` declaration removes marked function
- Calling a deleted function or taking its address causes a compilation error
- No code is generated for deleted function
- Deleted function are treated as user-declared
- `delete` declaration can be used on any function, not only special class member functions
- `delete` can be used to avoid unwanted implicit conversion of function arguments

delete KEYWORD

```cpp
void integral_only(int a) {
    // ...
}
void integral_only(double d) = delete;

integral_only(10);  // OK
short s = 3;
integral_only(s);   // OK - implicit conversion to int
integral_only(3.0); // error - use of deleted function
```

EXERCISE

Mark copy constructors as default.

Delete `getY()` method in `Square` and all default (non-parametric) constructors of shapes.

`final` KEYWORD

```cpp
struct A final {};

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

`final` keyword used after a class/struct declaration blocks inheritance from this class.

`final` KEYWORD

```
struct A {
    virtual void foo() const final {}
    void bar() const final {}    // compilation error, only virtual
                                 // functions can be marked as final
};

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

`final` used after a virtual function declaration blocks overriding the implementation in derived classes.

`override` KEYWORD

```cpp
struct Base {
    virtual void a();
    virtual void b() const;
    virtual void c();
    void d();
};
```

```cpp
struct WithoutOverride : Base {
    void a(); // overrides Base::a()
    void b(); // doesn't override B::b() const
    virtual void c(); // overrides B::c()
    void d(); // doesn't override B::d()
};
```

```cpp
struct WithOverride : Base {
    void a() override; // OK - overrides Base::a()
    void b() override; // error - doesn't override B::b() const
    virtual void c() override; // OK - overrides B::c(char)
    void d() override; // error - B::d() is not virtual
};
```

`override` declaration enforces a compiler to check, if given virtual function is declared in the same way in a base class.

EXERCISE

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?

RECAP

WHAT DO YOU REMEMBER FROM TODAY'S SESSION?

1. intro (25′)
2. `static_assert` (15′)
3. `nullptr` (10′)
4. scoped enums (30′)
5. ☕ break (15′)
6. `auto` keyword and range-based for loop (1h)
7. ☕ break (10′)
8. `using` alias (15′)
9. uniform initialization (40′)
10. 🍝 lunch break (50′)
11. new keywords: `default`, `delete`, `final`, `override` (1h)
12. recap (15′)

PRE-TEST

ANSWERS

1. WHAT IS THE TYPE OF VARIABLE v?

```
int i = 42;
const auto v = &i;
```

1. `const int`
2. `const int&`
3. `const int*`
4. other

2. WHICH OF THE FOLLOWING INITIALIZATIONS ARE VALID IN C++14?

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

1. `int values[] = { 1, 2, 3, 4, 5 };`
2. `P v = { 1, 4 };`
3. `P v{1, 4};`
4. `P v(1, 4);`
5. `std::vector<int> v = { 1, 2, 3, 4 };`
6. `std::vector<int> v(1, 2, 3, 4);`
7. `int v[] = { 1, 3, 5, 6.6 };`

3. WHICH OF THE FOLLOWING ELEMENTS CAN BE DEFINED AS DELETED (`= delete;`)?

1. default constructor
2. copy constructor
3. move constructor
4. copy assignment operator
5. move assignment operator
6. destructor
7. free function
8. class method
9. class member object

POST-TEST

The link to post-test will be sent to you in a next week.

It's better to forget some of the content and refresh your knowledge later.

It enhances knowledge retention :)

HOMEWORK

Take a look into `README.md` file from modern_cpp repository. You can complete all tasks and raise a Pull Request if you wish me to check your homework.

FEEDBACK

- What could be improved in this training?
- What was the most valuable for you?
- Training evaluation

THANK YOU 🙂