MODERN C++

C++11, C++14, C++17 FEATURES

Created by Łukasz Ziobroń and Kamil Szatkowski

`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

```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

- `default` declaration enforces a compiler to generate default implicit implementation for marked functions
- 6 special functions can be marked as `default`: default c-tor, copy c-tor, copy assignment operator, move c-tor, move assignment operator, destructor
- Operations declared as `default` are treated as user-declared (not implicitly-declared)
- The default implementation of default c-tor is calling default c-tor for every member
- The default implementation of d-tor is calling d-tor for every member
- The default implementation of copy operations is calling copy operation for every member
- The default implementation of move operations is calling move operation for every member

`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

```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 an 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

```
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
```

`final` KEYWORD

```
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

```cpp
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.

`constexpr` KEYWORD

Rationale: faster runtime binary by moving some computations at compile-time.

`constexpr` is an expression that can be evaluated at compile time and can appear in constant expressions. We can have:

- constexpr variable
- constexpr function
- constexpr constructor
- constexpr lambda (default from C++17)
- constexpr if (until C++17)

`constexpr` VARIABLES

```
int a = 10;                 // variable
const int b = 20;           // constant
const double c = 20;        // constant
constexpr int d = 30;       // constant at compile-time

constexpr auto e = a;       // error: initializer is not a contant expression
constexpr auto f = b;       // OK for integral, C++03 compatibility exception
constexpr auto g = c;       // error: initializer is not a contant expression
constexpr auto h = d;       // OK
```

- `constexpr` variable must be initialized immediately with constant expression. `const` does not need to be initialized with constant expression.
- `constexpr` variable must be a LiteralType

`constexpr` FUNCTIONS

```
constexpr int factorial11(int n) {  // C++11 compatible
{
    return (n == 0) ? 1 : n * factorial11(n-1);
}

constexpr int factorial14(int n) {  // C++14
    if (n == 0) {
        return 1;
    } else {
        return n * factorial14(n-1);
    }
}
```

`constexpr` function can be evaluated in both compile time and runtime. Evaluation at compile time can occur when the result is assigned to `constexpr` variable and arguments can be evaluated at compile time.

`constexpr` FUNCTIONS RESTRICTIONS

In C++11 `constexpr` functions were very restricted - only 1 return instruction (not returning void). From C++14 the only restrictions are, that function must not:

- contain `static` or `thread_local` variables
- contain uninitialized variables
- call non `constexpr` function
- use non-literal types
- be virtual (until C++20)
- use asm code blocks (until C++20)
- have try-catch block or throw exceptions (until C++20)

`constexpr` CONSTRUCTOR

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

    int x_;
    int y_;
};

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

class `Point` can be used in constexpr computations, eg in constexpr functions. It is a literal type. Constexpr constructor has the same restrictions as a constexpr function and a class cannot have a virtual base class.

`constexpr` LAMBDA

From C++17 all lambda functions are by default implicitly marked as constexpr, if possible.

`constexpr` keyword can also be used explicitly.

```cpp
auto squared = [](auto x) {                   // implicitly constexpr
    return x * x;
};

std::array<int, squared(8)> a;                // OK - array<int, 64>

auto squared = [](auto x) constexpr {         // OK
    return x * x;
};
```

`constexpr if`

```
if constexpr (a < 0)
    doThis();
else if constexpr (a > 0)
    doThat();
else
    doSomethingElse();
```

`constexpr if` selects only one block of instructions, depening on which condition is met. The condition and other blocks are not compiled in the binary. The condition must be a constant expression.

`constexpr if` IN SFINAE

`constexpr if` allows a simplification of template code used by SFINAE idiom.

```cpp
template<class T>   // C++17
auto compute(T x) {
    if constexpr(std::is_scalar_v<T>) {
        return singleComputation(x);
    } else {
        return multipleComputation(x);
    }
}
```

```cpp
template<class T>   // C++11
auto compute(T x) -> enable_if<std::is_scalar<T>::value, int>::type {
    return singleComputation(x);
}
template<class T>
auto compute(T x) -> enable_if<!std::is_scalar<T>::value, int>::type {
    return multipleComputation(x);
}
```

UNIFORM INITIALIZATION

C++98 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");
```

UNIFORM INITIALIZATION

C++11 INITIALIZATION WITH {}

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

```
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 narrow

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

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

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

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 : std::initializer_list<int>
std::vector<int> v = {1, 2, -3}; // creates vector from
                                 // std::initalizer_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

std::initializer_list<T>

```cpp
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<int>> c = {new int{1}, new int{2}};
// error - std::unique_ptr is non-copyable
```

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

MODERN C++

- `default` keyword
- `delete` keyword
- `final` keyword
- `override` keyword
- `constexpr` keyword
- Uniform initialization
- `std::initializer_list<T>`