MODERN C++

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

Created by Łukasz Ziobroń and Kamil Szatkowski

CODERS SCHOOL

C++ STANDARDS

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

C++ STANDARDS

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

C++11

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.

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

`using` ALIAS

C++11

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.

```
1 typedef std::ios_base::fmtflags Flags;
2 using Flags = std::ios_base::fmtflags;
3 Flags fl = std::ios_base::dec;
4
5 typedef std::vector<std::shared_ptr<Socket>> SocketContainer;
6 std::vector<std::shared_ptr<Socket>> typedef SocketContainer;  // equal t
7 using SocketContainer = std::vector<std::shared_ptr<Socket>>;
```

TEMPLATE ALISES

```
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

```
1 struct A {
2     explicit A(int);
3     int a;
4 };
5
6 struct B : A {
7     using A::A;  // implicit declaration of B::B(int)
8     B(int, int); // overloaded inherited Base ctor
9 };
```

- 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

SCOPED `enum`

C++11

Rationale: Stronger and less error-prone enumeration types.

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

enum Fruits {
    ORANGE,
    BANANA
};

Colors c = ORANGE;  // 11 or 0?
```

```
enum class Languages {
    ENGLISH,
    GERMAN,
    POLISH
};
Languages a = Languages::ENGLISH;
// Languages b = GERMAN;
// int c = Languages::ENGLISH;
int d = static_cast<int>(Languages::ENGLISH);
```

- 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 cha
    return 0;
}
```

- 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`

Change the code in ideone.com

`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;
```

AUTOMATIC TYPE DEDUCTION

C++11

`auto` KEYWORD

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

```
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
```

- 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 referer
                             // 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

```
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

`decltype`

Rationale: Deduction provided in contexts where auto is not allowed.

`decltype` allows a compiler to deduce the type of the variable or expression, eg. the returned type can be deduced from function parameters.

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

decltype(collection) otherCollection;    // otherCollection has type of col
decltype(collection)::mapped_type value; // value is float

template <typename T1, typename T2>
auto add(T1 a, T2 b) -> decltype(a + b)  // from C++14 decltype not neccess
{
    return a + b;
}
```

`decltype((expression))` will be presented after move semantics :)

`decltype(auto)`

`decltype(auto)` deduction mechanism preserves type modifiers (references, const, volatile).

`auto` deduction mechanism does not preserve type modifiers.

```
template<class FunctionType, class... Args>
decltype(auto) Example(FunctionType fun, Args&&... args)
{
    return fun(std::forward<Args>(args)...);
}
```

MODERN C++

- C++ language history and standards
- `static_assert`
- `using` alias
- Template aliases
- Constructor inheritance
- Scoped `enum`
- `enum` base
- `enum` forward declaration
- `auto` keyword
- Deduction rules
- Function declaration with arrow
- `decltype` keyword
- `decltype(auto)`