

C++ 14

complete feature description

Dawid Pilarski

dawid.pilarski@panicsoftware.com

blog.panicsoftware.com

Introduction

Language changes

- type deduction
- better lambdas
- variable templates
- relaxed constexpr
- binary literals
- non const constexpr functions
- deprecated attribute
- member initializers and aggregates
- sized deallocations
- digit separation
- contextual conversions

Library changes

- ud literals extended
- constexpr for STL
- std::quoted
- null forward iterators
- std::result_of SFINAE-friendly
- std::shared_timed_mutex
- std::get<T>()
- integral_constant as a functor
- std::index_sequence
- std::exchange
- std::equal,
std::is_permutation,
std::mismatch

Language changes

Type deduction

Type deduction

we could deduce the returned type in C++ 11:

```
template <typename T>
auto sum(T a, T b) -> decltype(a+b){
    return a + b;
}
```

Type deduction

we could deduce the returned type in C++ 11:

```
template <typename T>
auto sum(T a, T b) -> decltype(a+b){
    return a + b;
}
```

The issue in C++ 11

$a + b$ is now a code repetition!

C++14

How to avoid the code repetition?

```
template <typename T>  
auto sum(T a, T b){  
    return a + b;  
}
```


C++14

How to avoid the code repetition?

```
template <typename T>  
auto sum(T a, T b){  
    return a + b;  
}
```

```
template <typename T>  
decltype(auto) sum(T a, T b){  
    return a + b;  
}
```

Where can we use deduction?

- for regular functions

```
struct A {  
    auto f(); // forward declaration  
};  
auto A::f() { return 42; }
```

Where can we use deduction?

- for regular functions

```
auto f(); // return type is unknown
auto f() { return 42; } // return type is int
auto f(); // redeclaration
int f(); // error, declares a different function
```

Where can we use deduction?

- for regular functions
- for function templates

```
template <class T> auto g(T t); // forward declaration  
template <class T> auto g(T t) { return t; }  
template <class T> auto g(T t); // redeclaration
```

Where can we use deduction?

- for regular functions
- for function templates

```
template <class T> auto f(T t) { return t; } // #1
template auto f(int); // OK
template char f(char); // error, no matching template
template<> auto f(double); // OK
```

```
template <class T> T f(T t) { return t; } // #2 OK
template char f(char); // OK
template auto f(float); // OK, matches #1
```

Where can we use deduction?

- for regular functions
- for function templates
- in trailing-return-types

```
int global;  
auto foo() -> auto&{  
    return global;  
}
```

Where can we use deduction?

- for regular functions
- for function templates
- in trailing-return-types

```
[] () -> auto& {}
```

```
[] () -> decltype(auto) {}
```

Where can we use deduction?

- for regular functions
- for function templates
- in trailing-return-types
- variable definition

```
int&& f();
```

```
auto          a1 = f(); // deduced int
```

```
decltype(auto) a2 = f(); // deduced int&&
```


decltype(auto)

`decltype(auto)` is another way to perform type deduction.

Algorithm for the deduction is:

- for id-expressions and class member accesses not in `()` deduced type is type of denoted variable

decltype(auto)

`decltype(auto)` is another way to perform type deduction.

Algorithm for the deduction is:

- for id-expressions and class member accesses not in `()` deduced type is type of denoted variable
- if expression is lvalue deduced type is `T&`

decltype(auto)

`decltype(auto)` is another way to perform type deduction.

Algorithm for the deduction is:

- for id-expressions and class member accesses not in `()` deduced type is type of denoted variable
- if expression is lvalue deduced type is `T&`
- if expression is xvalue deduced type is `T&&`

decltype(auto)

`decltype(auto)` is another way to perform type deduction.

Algorithm for the deduction is:

- for id-expressions and class member accesses not in `()` deduced type is type of denoted variable
- if expression is lvalue deduced type is `T&`
- if expression is xvalue deduced type is `T&&`
- if expression is prvalue deduced type is `T`

For `auto` deduction algorithm is following:

- for “auto” deduced type is never a reference

For `auto` deduction algorithm is following:

- for “`auto`” deduced type is never a reference
- for “`auto&`” deduced type is always lvalue reference

For `auto` deduction algorithm is following:

- for “`auto`” deduced type is never a reference
- for “`auto&`” deduced type is always lvalue reference
- for “`auto&&`” deduced type is lvalue or rvalue reference depending on expression

deduction algorithms comparison

```
int i;  
int&& f();
```

```
auto x3a = i;      //int  
auto x4a = (i);    //int  
auto x5a = f();    //int  
auto x6a = {1, 2}; //init_list<int>  
auto *x7a = &i;    //int*
```

```
decltype(auto) x3d = i;      //int  
decltype(auto) x4d = (i);    //int&  
decltype(auto) x5d = f();    //int&&  
decltype(auto) x6d = {1, 2}; //error  
decltype(auto)*x7d = &i;     //error
```


deduction algorithms comparison

```
int i;  
int&& f();
```

```
auto x3a = i;      //int  
auto x4a = (i);    //int  
auto x5a = f();    //int  
auto x6a = {1, 2}; //init_list<int>  
auto *x7a = &i;    //int*
```

```
decltype(auto) x3d = i;      //int  
decltype(auto) x4d = (i);    //int  
decltype(auto) x5d = f();    //int  
decltype(auto) x6d = {1, 2}; //error  
decltype(auto)*x7d = &i;     //error
```

deduction algorithms comparison

```
int i;  
int&& f();
```

```
auto x3a = i;      //int  
auto x4a = (i);    //int  
auto x5a = f();    //int  
auto x6a = {1, 2}; //init_list<int>  
auto *x7a = &i;    //int*
```

```
decltype(auto) x3d = i;      //int  
decltype(auto) x4d = (i);    //int  
decltype(auto) x5d = f();    //int  
decltype(auto) x6d = {1, 2}; //error  
decltype(auto)*x7d = &i;     //error
```

deduction algorithms comparison

```
int i;  
int&& f();
```

```
auto x3a = i;      //int  
auto x4a = (i);    //int  
auto x5a = f();    //int  
auto x6a = {1, 2}; //init_list<int>  
auto *x7a = &i;    //int*
```

```
decltype(auto) x3d = i;      //int  
decltype(auto) x4d = (i);    //int&  
decltype(auto) x5d = f();    //int&&  
decltype(auto) x6d = {1, 2}; //error  
decltype(auto)*x7d = &i;     //error
```

deduction algorithms comparison

```
int i;  
int&& f();
```

```
auto x3a = i;      //int  
auto x4a = (i);    //int  
auto x5a = f();    //int  
auto x6a = {1, 2}; //init_list<int>  
auto *x7a = &i;    //int*
```

```
decltype(auto) x3d = i;      //int  
decltype(auto) x4d = (i);    //int&  
decltype(auto) x5d = f();    //int&&  
decltype(auto) x6d = {1, 2}; //error  
decltype(auto)*x7d = &i;    //error
```

How can this improve the code

Production code example

```
template <typename Functor, typename... Args>
auto call_wrapper(Functor fun, Args&&...) ->
typename std::result_of<decltype(fun)(Args&&...)>
                                ::type {
    auto future = fun(std::forward<Args>(args)...);
    //...
    return future;
}
```

How can this improve the code

Production code example

```
template <typename Functor, typename... Args>
auto call_wrapper(Functor fun, Args&&...) ->
typename std::result_of<decltype(fun)(Args&&...)>
                                ::type {
    auto future = fun(std::forward<Args>(args)...);
    //...
    return future;
}
```

```
template <typename Fun, typename... Args>
auto call_wrapper(Fun func, Args&&... args){
    auto future = func(std::forward<Args>(args)...);
    //...
    return future;
}
```

Functionality improvements

perfect return type forwarding (wrappers)

```
template <typename F, typename... Args>
decltype(auto) log_wrapper(F&& f, Args&&... args){
    std::cout << "function called." << std::endl;
    return f(std::forward<Args>(args)...);
}
```

Better lambdas

Better lambdas

there were couple of lambdas improvements made:

- generic lambdas were introduced

Better lambdas

there were couple of lambdas improvements made:

- generic lambdas were introduced
- lambdas requirements on parameters were relaxed

Better lambdas

there were couple of lambdas improvements made:

- generic lambdas were introduced
- lambdas requirements on parameters were relaxed
- generalized lambda capture was introduced

Lambdas in C++ 11

C++ 11 lambdas syntax

```
auto l = [<lambda-capture>]<(parameter-list)>  
    <mutable> <noexcept/throws()> <-> trailing-return-type >  
    {}
```

Lambdas in C++ 11

C++ 11 lambdas syntax

```
auto l = [<lambda-capture>]<(parameter-list)>  
    <mutable> <noexcept/throws()> <-> trailing-return-type >  
    {}
```

generates

```
class lambda_unique{  
    using function_type = deduced_ret_type(*) (parameter-list);  
public:  
    lambda_unique() = delete;  
    lambda_unique& operator =(lambda_unique&) = delete;  
  
    inline deduced_ret_type operator()(parameter-list) <const> <noexcept>{};  
  
    // if empty lambda capture  
    operator function_type() const noexcept;  
    //endif  
}
```

Lambdas in C++ 11

C++ 11 lambdas syntax

```
auto l = [<lambda-capture>]<(parameter-list)>  
    <mutable> <noexcept/throws()> <-> trailing-return-type >  
    {}
```

generates

```
class lambda_unique{  
    using function_type = deduced_ret_type(*) (parameter-list);  
public:  
    lambda_unique() = delete;  
    lambda_unique& operator =(lambda_unique&) = delete;  
  
    inline deduced_ret_type operator()(parameter-list) <const> <noexcept>{};  
  
    // if empty lambda capture  
    operator function_type() const noexcept;  
    //endif  
}
```

Lambdas in C++ 11

C++ 11 lambdas syntax

```
auto l = [<lambda-capture>](<parameter-list>
    <mutable> <noexcept/throws()> <-> trailing-return-type >
    {}
```

generates

```
class lambda_unique{
    using function_type = deduced_ret_type(*)(<parameter-list>);
public:
    lambda_unique() = delete;
    lambda_unique& operator=(lambda_unique&) = delete;

    inline deduced_ret_type operator()(<parameter-list> <const> <noexcept>{});

    // if empty lambda capture
    operator function_type() const noexcept;
    //endif
}
```

Lambdas in C++ 11

C++ 11 lambdas syntax

```
auto l = [<lambda-capture>]<(parameter-list)>  
    <mutable> <noexcept/throws()> <-> trailing-return-type >  
    {}
```

generates

```
class lambda_unique{  
    using function_type = deduced_ret_type(*) (parameter-list);  
public:  
    lambda_unique() = delete;  
    lambda_unique& operator =(lambda_unique&) = delete;  
  
    inline deduced_ret_type operator()(parameter-list) <const> <noexcept>{};  
  
    // if empty lambda capture  
    operator function_type() const noexcept;  
    //endif  
}
```


Limitations of the C++11 lambdas

Issues identified in the C++11 lambdas:

- default arguments could not appear in the parameter-list

```
//invalid according to standard only  
auto foo = [](int i=0){/*...*/};
```

Limitations of the C++11 lambdas

Issues identified in the C++11 lambdas:

- default arguments could not appear in the parameter-list
 - there is no way to reuse the compound statement from the lambda to different types
-

```
auto comparator = template<typename T>
    [](T lhs, T rhs){return lhs < rhs;};
```

Limitations of the C++11 lambdas

Issues identified in the C++11 lambdas:

- default arguments could not appear in the parameter-list
 - there is no way to reuse the compound statement from the lambda to different types
 - no deduction on parameters
-

production code example

```
[this] (std::vector<std::unique_ptr<Element>>&  
        container){/*...*/}
```

`auto` type deduction can appear in parameter list

`auto` type deduction can appear in parameter list

production code example

```
[this](auto& container){/*...*/}
```

Generated lambdas type

If the `auto` type-specifier in the parameter-declaration the lambda's type looks following:

```
class lambda_unique{
    template <typename types_to_deduce>
    using function_type = deduced_ret_type(*)
                          (types_to_deduce parameter-list);
public:
    lambda_unique() = delete;
    lambda_unique& operator=(lambda_unique&) = delete;

    template <types_to_deduce>
    inline deduced_ret_type
    operator()(types_to_deduce parameter-list) <const> <noexcept>{};

    // if empty lambda capture
    template <types_to_deduce>
    operator function_type<types_to_deduce>() const noexcept;
    //endif
}
```

Generated lambdas type

If the `auto` type-specifier in the parameter-declaration the lambda's type looks following:

```
class lambda_unique{
    template <typename types_to_deduce>
    using function_type = deduced_ret_type(*)
                        (types_to_deduce parameter-list);
public:
    lambda_unique() = delete;
    lambda_unique& operator =(lambda_unique&) = delete;

    template <types_to_deduce>
    inline deduced_ret_type
    operator()(types_to_deduce parameter-list) <const> <noexcept>{};

    // if empty lambda capture
    template <types_to_deduce>
    operator function_type<types_to_deduce>() const noexcept;
    //endif
}
```

Generated lambdas type

If the `auto` type-specifier in the parameter-declaration the lambda's type looks following:

```
class lambda_unique{
    template <typename types_to_deduce>
    using function_type = deduced_ret_type(*)
                          (types_to_deduce parameter-list);
public:
    lambda_unique() = delete;
    lambda_unique& operator =(lambda_unique&) = delete;

    template <types_to_deduce>
    inline deduced_ret_type
    operator()(types_to_deduce parameter-list) <const> <noexcept>{};

    // if empty lambda capture
    template <types_to_deduce>
    operator function_type<types_to_deduce>() const noexcept;
    //endif
}
```


Generated lambdas type

If the `auto` type-specifier in the parameter-declaration the lambda's type looks following:

```
class lambda_unique{
    template <typename types_to_deduce>
    using function_type = deduced_ret_type(*)
                        (types_to_deduce parameter-list);

public:
    lambda_unique() = delete;
    lambda_unique& operator =(lambda_unique&) = delete;

    template <types_to_deduce>
    inline deduced_ret_type
    operator()(types_to_deduce parameter-list) <const> <noexcept>{};

    // if empty lambda capture
    template <types_to_deduce>
    operator function_type<types_to_deduce>() const noexcept;
    //endif
}
```

Default parameters in lambda

Since C++ 14 we can have default arguments in lambdas

```
auto lambda = [](int i=42){cout << i << endl;};  
lambda();
```

Output

42

Default parameters in lambda

Since C++ 14 we can have default arguments in lambdas

```
auto lambda = [](int i=42){cout << i << endl;};  
lambda(0);
```

Output

0

Generic lambdas + default parameter = fail

ACHTUNG! ACHTUNG!

Consider following code:

```
template <typename T>  
void foo(T arg=5){} // this is correct  
//...
```

```
foo(); // ill-formed
```

Generic lambdas + default parameter = fail

ACHTUNG! ACHTUNG!

Consider following code:

```
template <typename T>  
void foo(T arg=5){} // this is correct  
//...
```

```
foo(); // ill-formed  
foo<int>(); // it's ok
```

Generic lambdas + default parameter = fail

ACHTUNG! ACHTUNG!

```
template <typename T>
void foo(T arg=5){}
//...
```

```
foo(); // ill-formed
```

```
template <typename T=int>
void foo(T arg=5){}
//...
```

```
foo(); // now it's fine too
```

Generic lambdas + default parameter = fail

Since we know what code is generated for the lambda, we might expect the same behavior with lambdas:

```
[] (auto i=5){return i;}(); // ill-formed
```

Fixing the failure

```
[] (auto i=5){return i;}(); // ill-formed
```

```
[] <typename T=int>(T i=5){return i;}(); // it's ok
```


Fixing the failure

```
[] (auto i=5){return i;}(); // ill-formed
```

```
[] <typename T=int>(T i=5){return i;}(); // it's ok
```

But it is available only since C++ 20.

Summarizing

- Generic lambdas are cool.

Summarizing

- Generic lambdas are cool.
- Default parameters are cool.

Summarizing

- Generic lambdas are cool.
- Default parameters are cool.
- But do not mix them together.

Generalized lambda capture

Consider following code

```
struct Foo{  
    int value=0;  
    auto foo(){return [=]{return value;};}  
};
```

Generalized lambda capture

Consider following code

```
struct Foo{  
    int value=0;  
    auto foo(){return [=]{return value;};}  
};  
  
int main(){  
    auto lambda = Foo{}.foo();  
    std::cout << lambda() << std::endl;  
}
```

Generalized lambda capture

Consider following code

```
struct Foo{  
    int value=0;  
    auto foo(){return [=]{return value;};}  
};  
  
int main(){  
    auto lambda = Foo{}.foo();  
    std::cout << lambda() << std::endl;  
}
```

Where is the bug?

How to solve this issue?

In C++ 11

```
struct Foo{  
    int value=0;  
    auto foo(){  
        int value = value;  
        return [=]{  
            return value;  
        };  
    }  
};
```


How to solve this issue?

In C++ 11

```
struct Foo{  
    int value=0;  
    auto foo(){  
        int value = value;  
        return [=]{  
            return value;  
        };  
    }  
};
```

Where is the issue now?

How to solve this issue?

In C++ 11

Probably Bug free implementation

```
struct Foo{  
    int value=0;  
    auto foo(){  
        int value_ = value;  
        return [=]{  
            return value_;  
        };  
    }  
};
```

How to solve the issue?

In C++ 14

```
struct Foo{  
    int value=0;  
    auto foo(){  
        return [value=value]{  
            return value;  
        };  
    }  
};
```

Lambda capture

- capture default

Lambda capture

- capture default
 - `&`

```
int a, b, c, d;  
//...  
[&]{d = a + c;  
    return d;  
} // test, a, c  
    // captured by ref
```

Lambda capture

- capture default
 - &
 - =

```
int a, b, c, d;
```

```
//...
```

```
[=]{d = a + c;
```

```
    return d;
```

```
} // test, a, c
```

```
    // captured by copy
```

Lambda capture

- capture default
 - &
 - =
- simple capture

Lambda capture

- capture default
 - &
 - =
- simple capture
 - identifier

```
int a, b, c, d;
```

```
//...
```

```
[&, d]{d = a + c;
```

```
    return d;
```

```
} // a, c - reference
```

```
    // d - copy
```


Lambda capture

- capture default
 - &
 - =
- simple capture
 - identifier
 - & identifier

```
int a, b, c, d;
```

```
//...
```

```
[=, &d]{d = a + c;
```

```
return d;
```

```
} // a, c - copy
```

```
    // d - reference
```

Lambda capture

- capture default
 - &
 - =
- simple capture
 - identifier
 - & identifier
 - this

```
class Operands{  
    int a, int b;  
    // ...  
    template <typename F>  
    auto create_expression(F& f){  
        return [this, &f]  
            {return f(a, b);};  
    }  
};
```

Lambda capture

- capture default
 - &
 - =
- simple capture
 - identifier
 - & identifier
 - this
- init-capture (C++ 14)

Lambda capture

- capture default
 - &
 - =
- simple capture
 - identifier
 - & identifier
 - this
- init-capture (C++ 14)
 - identifier initializer

```
int a, b, c, d;  
//...  
[lhs=a, rhs=c, result=d]  
{  
    result = lhs + rhs;  
    return result;  
} // all by value
```

Lambda capture

- capture default
 - &
 - =
- simple capture
 - identifier
 - & identifier
 - this
- init-capture (C++ 14)
 - identifier initializer
 - & identifier initializer

```
int a, b, c, d;  
//...  
[&lhs=a, rhs=c, result=d]  
{  
    result = lhs + rhs;  
    return result;  
} // all by reference
```

Other ways to use capture list

- move object into closure

```
std::unique_ptr<int> a = /**/;  
[a=std::move(a)]{};
```

Other ways to use capture list

- move object into closure
- constness removal

```
const int i=42;  
[i=i]()mutable{i++;}
```

Warning!

- capturing by & captures this through the pointer

Warning!

- capturing by `&` captures `this` through the pointer
- capturing by `=` captures `this` through the pointer

Warning!

- capturing by `&` captures `this` through the pointer
- capturing by `=` captures `this` through the pointer
- As of C++ 20 capturing this through pointer with `=` is deprecated

Warning!

- capturing by `&` captures `this` through the pointer
- capturing by `=` captures `this` through the pointer
- As of C++ 20 capturing this through pointer with `=` is deprecated
- Use explicit capture and you are safe.

Variable templates

Variable templates

let's calculate area of the circle:

```
template <typename T>  
T calculate_circle_area(T r){  
    return r*r*pi; // what is pi?  
}
```

Variable templates

let's calculate area of the circle:

```
template <typename T>  
T calculate_circle_area(T r){  
    return r*r*pi; // what is pi?  
}
```

```
#define pi 3.1415926535897932385
```

What is the issue?

Variable templates

let's calculate area of the circle:

```
template <typename T>
T calculate_circle_area(T r){
    return r*r*pi; // what is pi?
}
```

#define pi 3.1415926535897932385

calculation performed on doubles.

Common workarounds

- create a member of the class

```
template <typename T>
struct Pi{
    static const T value =
        3.1415926535897932385;
};
```


Common workarounds

- create a member of the class

```
template <typename T>
struct Pi{
    static const T value =
        3.1415926535897932385;
};

template <typename T>
struct Pi<T>::value =
    3.1415926535897932385;
```

Common workarounds

- create a member of the class
- create function template

```
template <typename T>  
T pi(){return 3.1415926535897932385;}
```

How to do it with variable template

```
template <typename T>  
T pi = 3.1415926535897932385;
```

```
template <typename T>  
T calculate_circle_area(T r){  
    return r*r*pi<T>;  
}
```

How to do it with variable template

```
template <typename T>  
T pi = 3.1415926535897932385;
```

```
template <typename T>  
T calculate_circle_area(T r){  
    return r*r*pi<T>;  
}
```

What might go wrong?

```
template <typename T>  
T value = 2;
```

What might go wrong?

```
template <typename T>
T value = 2;

value<char>=3;
assert(value<signed char> == 3); // failure
```

How to use variable templates?

Use them to define constants.

Relaxed constexpr constraints

relaxed constexpr functions

since C++ 14 you can:

- declare variables

```
constexpr int foo(){  
    int a; //allowed  
    static int i = 5; // ill-formed  
    thread_local int k = 10; // not allowed  
  
    return a;  
}
```

relaxed constexpr functions

since C++ 14 you can:

- declare variables
- use if and switch statements

```
constexpr const char* to_string(bool b){  
    if (b)  
        return "true";  
    return "false";  
}
```

relaxed constexpr functions

since C++ 14 you can:

- declare variables
- use if and switch statements
- use loops

```
constexpr int fibonacci(int idx){
    int pre_prev = 0;
    int prev = 1;
    for(int i=2; i < idx+2; i++){
        int current = pre_prev + prev;
        pre_prev = prev;
        prev=current;
    }
    return prev;
}
```

relaxed constexpr functions

since C++ 14 you can:

- declare variables
 - use if and switch statements
 - use loops
 - mutate objects (with limitations)
-

```
int my_global;  
constexpr int foo(int& test){  
    my_global++; //ill-formed  
    test++; //fine  
    return test;  
}
```

C++ 11

```
constexpr int operator[](int idx){  
    return *(iBegin + idx);  
}
```

C++ 11

```
constexpr int operator[](int idx){  
    assert(idx < iSize); //will not work  
    return *(iBegin + idx);  
}
```

C++ 11

```
constexpr int operator[](int idx){  
    return assert(idx < iSize),  
           *(iBegin + idx); // might not work  
}
```

C++ 11

```
constexpr int operator[](int idx){  
    return ((idx < iSize) ?  
            static_cast<void>(nullptr) :  
            []{assert(false)}()),  
    *(iBegin + idx); // will work  
}
```



```
constexpr int operator[](int idx){  
    assert(idx < iSize); //works fine  
    return *(iBegin + idx);  
}
```

Binary literals

Binary literals

Since C++ 14 we can use binary literals to denote value of integers:

Test for the `std::bitset`

C++ 11:

```
EXPECT_EQ(std::bitset<8>{0xe5},  
          ~std::bitset<8>{0x1a});
```

Binary literals

Since C++ 14 we can use binary literals to denote value of integers:

Test for the `std::bitset`

C++ 11:

```
EXPECT_EQ(std::bitset<8>{0xe5},  
          ~std::bitset<8>{0x1a});
```

C++ 14:

```
EXPECT_EQ(std::bitset<8>{0b11100101},  
          ~std::bitset<8>{0b00011010});
```

Digit separator

How to make it even more readable?

```
EXPECT_EQ(std::bitset<8>{0b11100101},  
          ~std::bitset<8>{0b00011010});
```

How to make it even more readable?

```
EXPECT_EQ(std::bitset<8>{0b11100101},  
          ~std::bitset<8>{0b00011010});
```

```
EXPECT_EQ(std::bitset<8>{0b1110'0101},  
          ~std::bitset<8>{0b0001'1010});
```

Further improvements

```
// Note: be sure W is always positive so add 7000  
W += 7000;
```


Further improvements

```
// Note: be sure W is always positive so add 7000  
W += 7000;
```

```
// Note: be sure W is always positive so add 7000  
W += 7'000;
```

Much more readable :)

Non const constexpr

But ... why?

Consider following C++ 11 code:

```
struct Wrapper{  
    constexpr Wrapper(int& v);  
    int& get() {return v_;}  
    constexpr const int& get() /*const*/ {return v_;}  
private:  
    int& v_;  
};
```

But ... why?

Consider following C++ 11 code:

```
struct Wrapper{  
    constexpr Wrapper(int& v);  
    int& get() {return v_;}  
    constexpr const int& get() /*const*/ {return v_;}  
private:  
    int& v_;  
};
```

But ... why?

Consider following C++ 11 code:

```
struct Wrapper{  
    constexpr Wrapper(int& v);  
    int& get() {return v_;}  
    constexpr const int& get() /*const*/ {return v_;}  
private:  
    int& v_;  
};
```

But ... why?

consider the usage of the code:

```
int a = 42;  
constexpr int b = Wrapper{a}.get(); //??
```

But ... why?

consider the usage of the code:

```
int a = 42;  
constexpr int b = Wrapper{a}.get(); //??
```

The code will not compile!

How to fix this?

```
struct Wrapper{  
    constexpr Wrapper(int& v);  
    constexpr int& get() {return v_;}  
    constexpr const int& get() {return v_;}  
private:  
    int& v_;  
};
```


How to fix this?

```
struct Wrapper{  
    constexpr Wrapper(int& v);  
    constexpr int& get() {return v_;}  
    constexpr const int& get() {return v_;}  
private:  
    int& v_;  
};
```

```
struct Wrapper{  
    constexpr Wrapper(int& v);  
    constexpr int& get() {return v_;}  
    constexpr const int& get() const {return v_;}  
private:  
    int& v_;  
};
```

C++14 is not fully backward compatible with C++11.
Some compilation issues might occur.

Deprecation in C++

[[deprecated]]

Since C++14 we can mark stuff as deprecated:

```
[[deprecated("because of reasons")]] void foo();  
// ...  
foo();
```

GCC

```
warning: 'void foo()' is deprecated: Because of reasons  
[-Wdeprecated-declarations]
```

[[deprecated]]

Since C++14 we can mark stuff as deprecated:

```
[[deprecated("because of reasons")]] void foo();  
// ...  
foo();
```

Clang

```
warning: 'foo' is deprecated: Because of reasons  
[-Wdeprecated-declarations]
```

Initializers and Aggregates

Initializing Aggregates

In the C++ 11 following code was ill-formed:

```
struct Test{  
    int a;  
    int b{1};  
};
```

```
//...
```

```
Test test{1,2}; // ill-formed
```

C++14 Aggregates

Since C++14 aggregates can have default initializers for non-static data members.

```
struct Test{  
    int a;  
    int b{1};  
};
```

```
Test test{1,2};  
// test.a == 1  
// test.b == 2
```


C++14 Aggregates

Since C++14 aggregates can have default initializers for non-static data members.

```
struct Test{  
    int a;  
    int b{1};  
};
```

```
Test test{1,2};  
// test.a == 1  
// test.b == 2
```

```
Test test1{1};  
// test1.a == 1  
// test1.b == 1
```

C++14 Aggregates

Since C++14 aggregates can have default initializers for non-static data members.

```
struct Test{  
    int a;  
    int b{1};  
};
```

```
Test test{1,2};  
// test.a == 1  
// test.b == 2
```

```
Test test1{1};  
// test1.a == 1  
// test1.b == 1
```

```
Test test2{};  
// test2.a == 0  
// test2.b == 1
```

Memory allocations improvement

What was changed

- memory allocations can be optimized

What was changed

- memory allocations can be optimized
- sized deallocations functions added

Speed up!

```
class Foo{/**/};  
int main(){  
    //might be faster due to sized deallocations  
    volatile auto ptr = std::make_unique<Foo>();  
}
```

Speed up!

```
class Foo{/**/};  
int main(){  
    //might result in single allocation / deallocation  
    volatile auto ptr  = std::make_unique<Foo>();  
    volatile auto ptr2 = std::make_unique<Foo>();  
}
```

More intuitive C++

Contextual conversions

Some wording in the standard caused following code to not compile:

```
class zero_init {  
public:  
    zero_init( )          : val( static_cast<T>(0) ) { }  
    zero_init( T val ) : val( val ) { }  
    operator T& ( )      { return val; }  
    operator T ( ) const { return val; }  
private:  
    T val;  
};
```

Contextual conversions

Some wording in the standard caused following code to not compile:

```
class zero_init {  
public:  
    zero_init( )      : val( static_cast<T>(0) ) { }  
    zero_init( T val ) : val( val )              { }  
    operator T& ( )    { return val; }  
    operator T  ( ) const { return val; }  
private:  
    T val;  
};
```

Contextual conversions

Some wording in the standard caused following code to not compile:

```
class zero_init {  
public:  
    zero_init( )          : val( static_cast<T>(0) ) { }  
    zero_init( T val ) : val( val ) { }  
    operator T& ( )      { return val; }  
    operator T ( ) const { return val; }  
private:  
    T val;  
};
```

Contextual conversions

Some wording in the standard caused following code to not compile:

```
class zero_init {
public:
    zero_init( )          : val( static_cast<T>(0) )  { }
    zero_init( T val ) : val( val )                  { }
    operator T& ( )       { return val; }
    operator T  ( ) const { return val; }
private:
    T val;
};
```

```
zero_init<int*> p;  assert( p == 0 );
p = new int(7);    assert(*p == 7 );
delete p;          // error!
delete (p+0);      // okay
delete +p;         // also okay
```

Contextual conversions

Some wording in the standard caused following code to not compile:

```
class zero_init {  
public:  
    zero_init( )          : val( static_cast<T>(0) ) { }  
    zero_init( T val ) : val( val ) { }  
    operator T& ( )       { return val; }  
    operator T ( ) const { return val; }  
private:  
    T val;  
};
```

```
zero_init<int*> p;  assert( p == 0 );  
p = new int(7);   assert(*p == 7 );  
delete p;         // error!  
delete (p+0);     // okay  
delete +p;        // also okay
```

Contextual conversions

Some wording in the standard caused following code to not compile:

```
class zero_init {
public:
    zero_init( )          : val( static_cast<T>(0) ) { }
    zero_init( T val ) : val( val )                { }
    operator T& ( )       { return val; }
    operator T ( ) const { return val; }
private:
    T val;
};
```

```
zero_init<int*> p;  assert( p == 0 );
p = new int(7);    assert(*p == 7 );
delete p;          // error!
delete (p+0);      // okay
delete +p;         // also okay
```

Library extensions

User defined literals

User defined literals

`operator""h`

`std::chrono::hours`

User defined literals

operator""h

std::chrono::hours

operator""min

std::chrono::minutes

User defined literals

operator""h

std::chrono::hours

operator""min

std::chrono::minutes

operator""s

std::chrono::seconds

User defined literals

operator""h

std::chrono::hours

operator""min

std::chrono::minutes

operator""s

std::chrono::seconds

operator""ms

std::chrono::milliseconds

User defined literals

operator""h

std::chrono::hours

operator""min

std::chrono::minutes

operator""s

std::chrono::seconds

operator""ms

std::chrono::milliseconds

operator""us

std::chrono::microseconds

User defined literals

operator""h

std::chrono::hours

operator""min

std::chrono::minutes

operator""s

std::chrono::seconds

operator""ms

std::chrono::milliseconds

operator""us

std::chrono::microseconds

operator""ns

std::chrono::nanoseconds

User defined literals

operator""h

std::chrono::hours

operator""min

std::chrono::minutes

operator""s

std::chrono::seconds

operator""ms

std::chrono::milliseconds

operator""us

std::chrono::microseconds

operator""ns

std::chrono::nanoseconds

operator""s

std::string

User defined literals

operator""h

std::chrono::hours

[std::literals::chrono_literals]

operator""min

std::chrono::minutes

[std::literals::chrono_literals]

operator""s

std::chrono::seconds

[std::literals::chrono_literals]

operator""ms

std::chrono::milliseconds

[std::literals::chrono_literals]

operator""us

std::chrono::microseconds

[std::literals::chrono_literals]

operator""ns

std::chrono::nanoseconds

[std::literals::chrono_literals]

operator""s

std::string

[std::literals::string_literals]

Usage example

```
std::string test("some text here");
```

Usage example

```
std::string test("some text here");  
  
using namespace std::literals::string_literals;  
auto test = "some text here"s;
```

Other uses of user defined literals

```
#include <thread>
#include <chrono>

using namespace std::literals::chrono_literals;
//...

//sleeps for 20 micro seconds
std::this_thread::sleep_for(20us);
```

Other uses of user defined literals

```
#include <thread>
```

```
#include <chrono>
```

```
using namespace std::literals::chrono_literals;
```

```
//...
```

```
//sleeps for 20 micro seconds
```

```
std::this_thread::sleep_for(20us);
```

```
std::this_thread::sleep_for(std::chrono::milliseconds{100});
```

```
//
```

```
std::this_thread::sleep_for(100ms);
```

`integer_sequence`

Integer sequence for metaprogramming

```
template<typename F, typename... T, std::size_t... Seq>
void apply_impl(F f, std::tuple<T...> u,
                std::index_sequence<Seq...>) {
    f(std::get<Seq>(u)...);
}
```

```
template<typename F, typename... T,
        typename Seq =
            std::make_index_sequence<sizeof...(T)> >
void apply(F f, std::tuple<T...> u) {
    apply_impl(f, u, Seq{});
}
```

```
int main() {
    apply([](int a, int b, int c) {},
          std::tuple{1, 2, 3});
}
```

Integer sequence for metaprogramming

```
template<typename F, typename... T, std::size_t... Seq>
void apply_impl(F f, std::tuple<T...> u,
               std::index_sequence<Seq...>) {
    f(std::get<Seq>(u)...);
}
```

```
template<typename F, typename... T,
        typename Seq =
            std::make_index_sequence<sizeof...(T)> >
void apply(F f, std::tuple<T...> u) {
    apply_impl(f, u, Seq{});
}
```

```
int main() {
    apply([](int a, int b, int c) {},
          std::tuple{1, 2, 3});
}
```

Integer sequence for metaprogramming

```
template<typename F, typename... T, std::size_t... Seq>
void apply_impl(F f, std::tuple<T...> u,
               std::index_sequence<Seq...>) {
    f(std::get<Seq>(u)...);
}
```

```
template<typename F, typename... T,
        typename Seq =
            std::make_index_sequence<sizeof...(T)> >
void apply(F f, std::tuple<T...> u) {
    apply_impl(f, u, Seq{});
}
```

```
int main() {
    apply([](int a, int b, int c) {},
          std::tuple{1, 2, 3});
}
```


Integer sequence for metaprogramming

```
template<typename F, typename... T, std::size_t... Seq>
void apply_impl(F f, std::tuple<T...> u,
               std::index_sequence<Seq...>) {
    f(std::get<Seq>(u)...);
}
```

```
template<typename F, typename... T,
        typename Seq =
            std::make_index_sequence<sizeof...(T)> >
void apply(F f, std::tuple<T...> u) {
    apply_impl(f, u, Seq{});
}
```

```
int main() {
    apply([](int a, int b, int c) {},
          std::tuple{1, 2, 3});
}
```

Integer sequence for metaprogramming

```
template<typename F, typename... T, std::size_t... Seq>
void apply_impl(F f, std::tuple<T...> u,
               std::index_sequence<Seq...>) {
    f(std::get<Seq>(u)...);
}
```

```
template<typename F, typename... T,
        typename Seq =
            std::make_index_sequence<sizeof...(T)> >
void apply(F f, std::tuple<T...> u) {
    apply_impl(f, u, Seq{});
}
```

```
int main() {
    apply([](int a, int b, int c) {},
          std::tuple{1, 2, 3});
}
```

Integer sequence for metaprogramming

```
template<typename F, typename... T, std::size_t... Seq>
void apply_impl(F f, std::tuple<T...> u,
               std::index_sequence<Seq...>) {
    f(std::get<Seq>(u)...);
}
```

```
template<typename F, typename... T,
        typename Seq =
            std::make_index_sequence<sizeof...(T)> >
void apply(F f, std::tuple<T...> u) {
    apply_impl(f, u, Seq{});
}
```

```
int main() {
    apply([](int a, int b, int c) {},
          std::tuple{1, 2, 3});
}
```

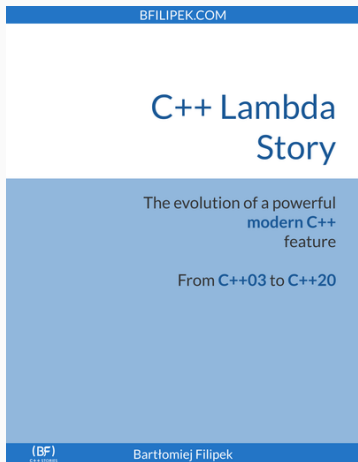
Thank you for your attention!

Bibliography:

- N3323
- N3472
- N3638
- N3648
- N3649
- N3651
- N3652
- N3653
- N3664
- N3760
- N3778
- N3302
- N3462
- N3469
- N3470
- N3471
- N3642
- N3644
- N3654
- N3659
- N3669
- N3670
- N3781
- N4140
- AKrzemi blog
- C++ Lambda Story

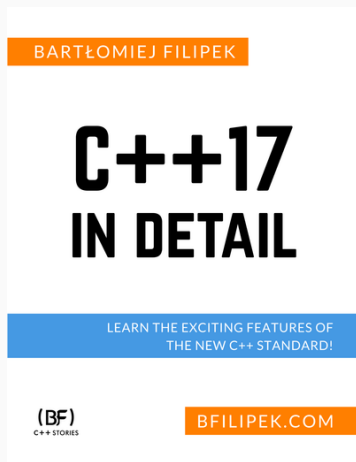
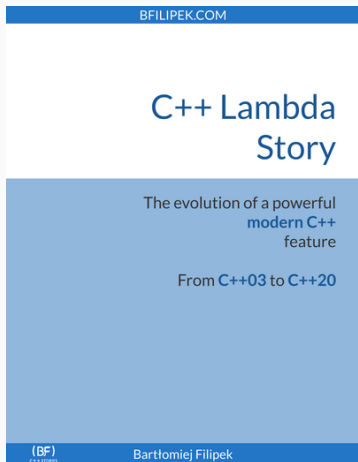
Recommendations:

Bartłomiej Filipek



Recommendations:

Bartłomiej Filipek



Invitations:

My blog blog.panicsoftware.com

Invitations:

My blog blog.panicsoftware.com

Cpp Polska cpp-polska.pl

Invitations:

My blog blog.panicsoftware.com

Cpp Polska cpp-polska.pl

Cpp Polska Slack cpppolska.slack.com

Invitations:

My blog blog.panicsoftware.com

Cpp Polska cpp-polska.pl

Cpp Polska Slack cpppolska.slack.com

Bartłomiej Filipek's blog www.bfilipek.com

Questions?
