From C++98 to C++11/14/17

Biggest Changes in Modern C++

Jan Hönig

Friedrich-Alexander-Universität Erlangen-Nürnberg

10.05.2022

Overview

Preamble

Definitions

- Old C++ : $\leq$ C++03
- Modern C++ : $\geq$ C++11 && $\leq$ C++17
- Next C++ : $\geq$ C++20

Compiler Support

Almost all compilers support C++14, most of compilers support C++17. Only “MSVC” supports all of C++20 (although gcc is close)

Feedback

jan.hoenig@fau.de

Not Covered

We can't possibly cover all new features in 45 minutes. Although C++ is backwards compatible, Modern C++ is a new language!

- Multi-threading
- `constexpr` and co
- New template features
- Huge parts of the standard library
- User defined literals
- `static_assert`
- attributes, new keywords
- And many more

References

“Diff”

- Effective Modern C++ : Scott Meyers
- C++17: Peter Pohmann
- C++11: Rainer Grimm

“All”

- cppreference.com

Further Resources & Exercises

Lectures @ FAU

- Advanced Programming Techniques:
- Advanced C++ Programming: online class at vhb.org

YouTube

- C++ Weekly With Jason Turner
- CopperSpice
- CppCon's "Back to Basics" track

Exercises

- `exercism.org`
- `codingame.com`
- `adventofcode.com`

Old C++'s Enum

```
enum Color { red, green, blue };
switch(r)
{
    case red  : std::cout << "red\n";   break;
    case green: std::cout << "green\n"; break;
    case blue : std::cout << "blue\n";  break;
}
```

- Pollutes namespace
- Implicitly converted to `int`

Modern C++'s Enum

```
enum class Color : uint8_t { red, green = 20, blue };
Color r = Color::blue;

switch(r)
{
    case Color::red  : std::cout << "red\n";   break;
    case Color::green: std::cout << "green\n"; break;
    case Color::blue : std::cout << "blue\n";  break;

    // int n = r; // error
    int n = static_cast<int>(r); // OK, n = 21
}
```

Not mentioned

Modern C++

- Opaque `enum`
- Unscoped `enum` can have a type
- Using-enum-declaration

Initializer Lists

Old C++

Cumbersome initialization of container classes

```cpp
std::vector<int> v;
v.push_back(0);
v.push_back(1);
v.push_back(2);
//...
```

Initializer Lists

```
std::vector<int> v = {0, 1, 2, 3};
std::map<int, int> m = {{0,1}, {1,2}, {3,10}};
```

1. `{0, 1, 2, 3}` creates a `std::initializer_list<int>`
2. Calling of the `vector( std::initializer_list<T> init)`

Uniform Initialization Syntax

Everything (scalars, sequences, aggregates) can be initialized with {}.

```
int i{5};
std::string{"foo"};
struct C{
    C(int a, int b): //...
}
C c{1, 2};
```

So many ways to initialize an integer!

```
int a(0);
int b = 0;
int c{0};
int d = {0};
```

Curly brackets everywhere?

```
struct Foo {
    Foo(int i, bool b);
    Foo(std::initializer_list<int> il);
}
Foo f{0, true}; // calls initializer list constructor
```

auto

Old C++

```
int a = 1;
std::vector<int>.iterator begin = v.begin();

template<typename Iterator>
void foo(It begin, It end){
    ??? value = *begin;
```

Modern C++

```
auto a = 1;
auto begin = v.begin()

template<typename Iterator>
void foo(It begin, It end){
    auto value = *begin;
```

When not to use auto

```cpp
auto foo(int a, int b, int c) {
    return 1;
}

template<typename T>
auto bar(T a, T b) { // ...
```

Explicitly Typed Initializer Idiom

Old C++

```
double foo(); //...
float f = foo() // implicit conversion
```

Modern C++

```
double foo(); //...
auto f = static_cast<float>(foo());
```

decltype

Gives you a type of a “thing”

Old C++

```
std::vector<int>.size_type v_size = v.size();
```

Modern C++

```
decltype(v.size()) v_size = v.size();
```

How to iterate over a vector

Old C++

```
for(std::vector<int>.size_type i; i < v.size(); ++i)
    // use v[i]
for(std::vector<int>::iterator it = v.begin();
    it != v.end(); ++it)
    // use *it
```

Modern C++

```
for(auto i; i < v.size(); ++i)
for(auto it = v.begin(); it != v.end(); ++it)
```

Range based loop

```
for (auto i : v) // changing i

for (auto & i: v) // changing i and values in v

for (const int& i : v) // nothing changes and no copy

for (auto& [first,second] : mymap) // iterate over a map
```

lambda

Functional Programming

- Functions are first-class citizens
- Function can be bound to names, passed as arguments and returned from other functions

Function object

- i.e. functors, callable, . . .
- Examples: function, templated function, class with `operator()`, lambda, member function

lambda

Usage

- Predicates for sorting
- Callback functions
- Function objects for function in stdlib
- Functions for multi-threading

lambda

```
[](int i) -> int {return i;};
[](auto i){return i;};
[](){}
```

Why on earth do we need nameless functions?

With loops

```
for(auto& e: v) {
    ++e;
}
```

Functional

```
std::for_each(v.begin(), v.end(), [](auto i){++i;});
```

More Examples

```
std::all_of(v.begin(), v.end(),
            [](auto i){return i > 0;});

std::sort(v.begin(), v.end(),
    [](MyClass a, MyClass b){return a.first < b.first;});

std::accumulate(v.begin(), v.end(), 1,
                std::multiplies<int>());
```

captures

What about the []?

[] captures surrounding variables by value or by References

- `[&foo]` capture by reference
- `[foo]` capture by value

```
int n = 0;
std::for_each(v.begin(), v.end(),
              [&n](auto i){n += i;});
```

lambda: details

```
int main(){
  int n = 0;
  auto f = [&n](auto i){n += 1;};
}
```

cppinsights.io

```
class __lambda_4_14
{
    public:
    __lambda_4_14(int & _n) : n{_n} {}

    template<class type_parameter_0_0>
    inline auto operator()(type_parameter_0_0 i) const
    {
        n = static_cast<int>(
                static_cast<<dependent type>>(n) + i);
    }
    private:
        int & n;
};

__lambda_4_14 f = __lambda_4_14{n};
```

Value Category

Each and every C++ expression is characterized by two properties

- type
- value category

Value Categories

- prvalue
- xvalue
- lvalue

lvalue

- has an address, or a name
- e.g.: variable, function
- string literal
- indirection
- assignments
- subscript expression

```
x
a = b
++a
*p
a[n]
```

xvalue

- function call or overloaded operator expression, whose return type is rvalue reference
- temporary objects
- address cannot be taken by address-of operator
- about to eXpire

```
std::move(x)
```

prvalue

- address cannot be taken by address-of operator
- can't be used as the left-hand operand of =
- literals
- product of an (overloaded) operator

```
42
i++
&a
a.m
[](auto i){return i;}
```

References

Lvalue reference

```cpp
int i = 0;
int& j = i;
```

Rvalue reference

```cpp
std::string&& s = std::string("foo");
// error: std::string& s = std::string("foo");
```

Example: Copy vs move constructors

```
struct Vector {
  int* data_;
  size_t size_;
  Vector::Vector(size_t size) :
        size_(size), data(new int[size_]) {}
  //...
```

Copy constructor

```
  Vector::Vector(const Vector& other) :
        Vector(other.size_) {
    std::copy(other.begin(), other.end(), data_);
}
```

Move constructor

```cpp
Vector::Vector(Vector&& other) noexcept :
        data_(other.data_), size_(other.size_) {
    other.data_ = nullptr;
    other.size_ = 0;
}
```

All this, so I can sometimes move instead of copying?

Yes, but there is more!

- move assignments
-

Rule of 5

If you have a copy/move constructor, copy/move assignment or a destructor, you need to implement this whole set!

(Formerly known as rule of 3)

Raw pointer

Any good old C pointer:

```
int* ip = new int(10);
//...
delete ip;
return;
```

Unique Pointer

Unique pointer **owns** the resource. It destructs it when it leaves the scope.

```
std::unique_ptr<int> ip(new int(15));
return;
```

It cannot be copied

```
std::unique_ptr<int> ip(new int(15));
//error std::unqiue_ptr<int> ip2 = ip;
```

It has to be moved

```
std::unique_ptr<int> ip(new int(15));
std::unique_ptr<int> ip2 = std::move(ip);
//ip is nullptr now
```

```
struct Vector {
    std::unique_ptr<int[]> data_;
    size_t size_;
    Vector(size_t)
        : data_(std::make_unique<int[]>(size)),
          size_(size) {}

    Vector(const Vector& other) : Vector(other.size_)
    {
        std::copy(other.data_.get(), other.data_.get()
                                    + other.size_, data_.get());
    }

    Vector(Vector && other) noexcept = default;
    ~Vector() = default;
    //... TODO copy assignment and move assignment
```

Shared Pointer

Reference counter. Increases reference on copies, decreases reference on destructs. Destroys the resource if amount of references $==0$

```
std::shared_ptr<int> pi(new int(15)); //init ref == 1
auto pi2 = pi; //copy, ref == 2
return pi; //pi2 gets deleted, ref == 1
//pi gets "copied out", ref == 1
```

Weak Pointer

Non owning pointer, which can be converted to `shared_ptr`

```
auto sp = std::make_shared<int>(15);
std::weak_ptr<int> wp = sp;
//...
if(std::shared_ptr<int> spt = wp.lock())
    //...
else
    // wp expired, i.e. sp was freed somewhere
```

Questions?