Cool, new features of C++11
(Pt. 1: Language Version)

An incomplete(!) introduction to changes introduced in C++11

Who am I?

- Alexander Hirsch
  - E-Mail: 1zeeky@gmail.com
  - Jabber/XMPP: z33ky@jabber.ccc.de
- Computer Science student at TU Darmstadt
- hobby-programmer
- looked into many languages and programming paradigms
- most experience with C++ (started getting proficient around 2007)

Disclaimer

What I state as facts might actually be untrue (though they are to my knowledge).

What I present as something "you should do", might not actually me a common idiom and just my personal opinion.

Examples are not necessarily in good coding style. They are just intended to show specific features and hint at use cases.

They should be valid C++, apart from missing `#include`s and some code should be inside a function, albeit it not being shown this way. Except of course where noted that the example would produce an error.

A shout out

A shout out to these websites I find very helpful for getting information about C++:

- `https://en.wikipedia.org/wiki/C%2B%2B11`
- `http://cppreference.com`
- `https://cppandbeyond.com`
- `https://stackoverflow.com`
- `https://isocpp.org`
- `http://meetingcpp.com`

The Standard document is obviously also a very good source. The (free) draft is mostly fine, too.

Cool, new features of C++11

- (rvalue-references)
- (initializer lists)
- `auto`
- `decltype`
- `constexpr`
- defaulting and deleting functions
- delegating constructors
- `static_assert`
- enum classes
- explicit type conversion operators
- `final` and `override`
- lambda expressions
- member initialization
- raw string literals
- range-based `for`

`auto`

type deduction for declarations

When defining variables you can let C++ **deduce the type** from the **initializing expression**.

`auto` resolves only to **non-cv[1]-qualified value- and pointer-types**.

[1] `const` and `volatile`

`auto` examples

```cpp
auto a = 5;
const auto b = 23;
const auto &c = b;

auto &d = b;         //error: mutable reference from const
auto e = 0, f = 0u; //error: different types inferred
auto g;              //error: nothing to deduce type from
```

`auto` examples

```
template<typename T>
struct Foo
{
    typedef std::map<int, T> TypeAssocMap;

    static const TypeAssocMap& GetAssociations();
};
```

```
const auto &assocs = Foo<std::size_t>::GetAssociations();
//assocs is of type const std::map<int, std::size_t>&
//a.k.a. const Foo<std::size_t>::TypeAssocMap&
```

`auto` examples

```
auto iter = assocs.rbegin();
//iter is of type std::reverse_iterator<implementation-defined>
//  implementation-defined a.k.a.
//  std::map<int, std::size_t>::const_iterator
//a.k.a. std::map<int, std::size_t>::const_reverse_iterator
//a.k.a. Foo<std::size_t>::TypeAssocMap::const_reverse_iterator
```

```
const auto &elem = *iter;
//elem is of type const std::pair<int, std::size_t>&
//a.k.a. const std::map<int, std::size_t>::value_type&
//a.k.a. const Foo<std::size_t>::TypeAssocMap::value_type&
```

`decltype`

type of expressions and declared variables

With `decltype` you can **deduce the type** of an expression or the **declared type** of a variable.

This includes cv-qualified value-, pointer- and reference-types.

`decltype` examples

```
volatile int a = 5, &b = a;
decltype(b) c = b; //c is of type volatile int&
auto        d = b; //d is of type          int
```

```
//weird to express before
template<typename T, typename U>
decltype(T() - U()) foo(T a, U b) { return a - b; }
```

`decltype` examples

```
decltype(Foo<std::size_t>::GetAssociations()) assocs =
    Foo<std::size_t>::GetAssociations();
//assocs will be a const reference
//more to type, but also more agnostic to changes
//with our implementation
```

Not too useful in a familiar environment, but when you want to write **generic, type agnostic algorithms** or such, `decltype` can help immensely.
Also when just **declaring variables**, as `auto` won't work there (nothing to infer the type from).

`constexpr`

compile-time evaluation of functions

Compile-time constants then:

- integral constants via `enum` or `const` (except `extern`'d)
- literals `#define`
- (most) expressions containing these

Now also single-statement functions (cannot return `void`) and **constructors** (must have (more or less) empty body).

Variables can also be declared `constexpr` $\rightarrow$ ensures "compile-timeness".

constexpr examples

```
struct Foo
{
    constexpr Foo(std::size_t i):I(i) {}

    constexpr Foo operator+(std::size_t i) const
        { return Foo(GetI() + i); }

    constexpr std::size_t GetI() const { return I; }

private:
    std::size_t I;
};

constexpr std::size_t one = 1;
char buf[(Foo(255) + one).GetI()];
```

defaulting and deleting functions

implement trivial functions and remove functions

defaulting and deleting functions - `default`

C++ can implicitly define the default-, copy- and move-constructors, destructors, and copy- and move-assignment operators.

The compiler may not do that (like writing a constructor prevents implicit definition of default-, copy- and move-constructors), but you can **explicitly default** them, in which case they will be defined as if they had been implicitly created.

defaulting and deleting functions - `default` examples

```
struct Foo
{
    Foo();
    Foo(const Foo&) = default;
    virtual ~Foo();
};

Foo() { /*...*/  }
Foo::~Foo() = default;
```

defaulting and deleting functions - `default` examples

```
struct Singleton
{
    static Singleton& Instance()
        { static Singleton s; return s; }

private:
    Singleton() = default;
};
```

defaulting and deleting functions - `delete`

You can also `delete` functions with the same syntax.

`delete` can:

- prevent implicit functions
- suppress inherited functions
- `delete` overloads (to prevent implicit conversion)

defaulting and deleting functions - `delete` examples

```
struct Foo
{
    Foo(const Foo&) = delete;
    int Bar();
    int Bar(long);
    int Bar(int) = delete;
};
struct Baz : Foo
{
    int Bar(long) = delete;
};

Foo f;
Foo o(f);  //error: Foo(const Foo&) deleted
f.Bar(1);  //error: Bar(int) deleted
f.Bar(1l); //ok

Baz b;
b.Bar(1l); //error: Bar(long) deleted
b.Bar();   //ok
```

delegating constructors

invoking a constructor from another constructor

delegating constructors

Sometimes you do very **similar initialization** from constructors with different parameters.

In C++11 you can write one constructor to do most or all of that work and let **other constructors delegate** to it.

The syntax is like calling a parent constructor.

You cannot do any further initialization.
You can put code in the constructor body though, just nothing else in the initialization list.

delegating constructors examples

```
struct Foo
{
    Foo():Foo(1, 1, 1) {}

    Foo(int a, int b, int c):A(a), B(b), C(c), Sum(A + B + C) {}

    template<typename Container>
    Foo(const Container &c):Foo(c[0], c[1], c[2]) {}

    int A, B, C;
    int Sum;
};
```

`static_assert`

compile-time assertions

`static_assert` - runtime assert

We already have the `assert`-macro in assert.h for run-time assertions.

One possible approach for a compile-time assertion was to conditionally allocate an array of size 0 or -1.

```
#define STATIC_ASSERT(cond) \
    { typedef int _assert[(cond) ? 0 : -1]; }
```

This does not make for a precise error message.

There are better, but less smaller ways to do this in C++98.

`static_assert`

C++11 defines `static_assert`, which takes a constant expression evaluating to `bool`-expression and a string literal as arguments.

When the expression is true, nothing happens.
When it is false, the compiler will error and display the string.

`static_assert` examples

```
template<typename Float, typename Integer = int>
Integer round2int(const Float f)
{
    static_assert(std::is_floating_point(f),
        "Trying to round non-floating point type");
    return static_cast<Integer>(f - static_cast<Float>(0.5));
}
```

`static_assert` examples

```
struct RGB
{
    RGB(const RGB&) = default;
    RGB(const uint8_t val[]):RGB(*reinterpret_cast<RGB*>(val)) {}

    operator uint8_t*()
    {
        return reinterpret_cast<uint8_t*>(this);
    }

    uint8_t R, G, B;
};
static_assert(
    std::is_standard_layout<RGB>::value,
    "struct RGB must be of standard layout.");
```

enum classes

scoped and stronger typed enums

enum classes - traditional enums

In C++, enumeration-types are one-way implicitly convertible:
Enum-member to integers.
You can still `static_cast` integers to enum-types.

enum classes examples - traditional enums

```
enum Day
{
    Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday
};

Day d = Sunday;
int i = d + 2;  //okay
i  = d / d;     //okay
i *= d;         //okay
++d;            //error: no operator++(Day)
d += 2;         //error: no operator+=(Day, int)
d  = d + d;     //error: cannot convert int to Day
d  = -d;        //error: cannot convert int to Day
```

enum classes examples - traditional enums overload operators

```
//do increment with wrap-around
Day& operator++(Day &day)
{
    static_assert(sizeof(day) >= sizeof(int));
    if(static_cast<int&>(day)++ == Sunday)
    {
        day = Monday;
    }
    return day;
}

//disallow arbitrary addition
int operator+(Day&, int) = delete;
//.. other integer-types, swapped parameters
```

enum classes

Enum classes are **not** implicitly convertible to integer.
You can still `static_cast` between these types.

Enum classes are scoped.
C++11 also allows scoped access to non-class enums.

Enum classes support specifying an underlying type.
This is done via a colon and the type after the enum class declaration.
C++11 also allows specifying one for non-class enums.

enum classes examples

```
enum class Day
{
    Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday
};

Day d = Sunday; //error: undeclared identifier 'Sunday'
int i = d + 2;  //error: no operator+(Day, int)
i  = d / d;     //error: no operator/(Day, Day)
i *= d;         //error: no operator*=(int, Day)
++d;            //error: no operator++(Day)
d += 2;         //error: no operator+=(Day, int)
d  = d + d;     //error: no operator+(Day, Day)
d  = -d;        //error: no operator-(Day)
```

enum classes examples “fixed”

```
enum class Day
{
    Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday
};

Day d = Day::Sunday;
int i = static_cast<int>(d) + 2;
d = static_cast<Day>(2);
```

enum classes examples - overload operators

```
//do increment with wrap-around
Day& operator++(Day &day)
{
    //no static_assert -> enum class Day : int { ... }
    if(static_cast<int&>(day)++ == Sunday)
    {
        day = Monday;
    }
    return day;
}
Day operator++(Day &day, int)
{
    Day prev = day;
    ++day;
    return prev;
}

//Day + int already not possible
//due to no implicit enum class -> int cast
```

explicit type conversion operators

requiring `static_cast` for type conversion

explicit type conversion operators

C++ type conversion operators convert types implicitly,
like single-argument constructors can.

We were already able to specify that constructors have to be invoked explicitly.
Now we can do the same thing for type conversion operators.

explicit type conversion operators examples - ctor

```
struct Type
{
    Type() = default;
    explicit Type(ExType);
    Type(ImType);
};

void doType(Type);
```

```
ExType e; ImType i;
Type f0 = i;                    //okay
Type f1 = e;                    //error
Type f2(e);                     //okay
doType(i);                      //okay
doType(e);                      //error
doType(Type(e));                //okay
```

explicit type conversion operators examples

```
struct Type
{
    Type() = default;
    explicit operator ExType();
    operator ImType();
};

void doExType(ExType);
void doImType(ImType);
```

```
Type t;
ImType  i = t;                    //okay
ExType e0 = t;                    //error
ExType e1 = static_cast<ExType>(t); //okay
doImType(t);                      //okay
doExType(t);                      //error
doExType(static_cast<ExType>(t));   //okay
```

`final` and `override` specifiers

convey (and assert) information about the inheritance hierarchy

`final` and `override` specifiers - `final`

The `final` specifier can be applied to classes and virtual functions.

Applied to a

virtual function	$\rightarrow$	function cannot be overridden
class	$\rightarrow$	cannot be inherited

`final` and `override` specifiers - `final` examples

```
struct Foo
{
    virtual void DoFoo() final;
};

struct Bar final : Foo
{
    void DoFoo(); //error: Foo::DoFoo is declared final
};

struct Baz : Bar //error: Bar is declared final
{
};
```

`final` and `override` specifiers - `override`

The `override` specifier can only be applied to virtual functions.

If a function is specified as `override`, it must override a method.

final and override specifiers - override examples

```
struct Base
{
    virtual void Foo();
    virtual void Bar(int);
            void Baz();
};

struct D0 : Base
{
    int  Foo()      override; //error: return type differs
    void Bar(long) override; //error: parameter types differ
    void Baz()      override; //error: not virtual
};

struct D1 : Base
{
    void Foo()     const override; //error: cv-qualifiers differ
    void Bar(int) override;        //okay
    void Baz();                    //okay
};
```

lambda expressions

anonymous functions

lambda expressions

Expressions that create **anonymous**[1] **functions**.
Actually more than a function: a **closure**.
Closure: You can enclose variables from the current scope.

```
[captures](parameters) -> return-type { body }
[captures](parameters) { body }
```

(return type deduced)

Lambdas without capture can be implicitly converted to a **function pointer**.

[1] nameless

lambda expressions - capture

The “capture” of a lambda expression specifies the **variables to enclose**.

- `[]` no captures
- `[x]` capture x by copy
- `[&x]` capture x by reference
- `[=]` capture all (by copy)
- `[&]` capture all (by reference)

‘Capture all’ refers to all variables not already mentioned.
Multiple captures are separated with a comma.

The “capture all” captures only the variables actually used in the lambdas body.

lambda expressions examples

```
std::vector<int> list = MaekAwsumLst();
int inc = 2, total = 0;
std::for_each(list.begin(), list.end(),
              [inc,&](int& x){ total += x; x += inc; });
```

```
//std::qsort(void *ptr, std::size_t count, std::size_t size,
//           int (*comp)(const void *, const void *));
int32_t a[] = { 7, 5, 8, 7, 4, 6, 9 };
std::qsort(a, 7, 4,
           [](const void *pa, const void *pb) -> int
           {
               auto a = *static_cast<const int32_t*>(pa);
               auto b = *static_cast<const int32_t*>(pb);
               return a < b ? -1 :
                      a > b ?  1 :
                      0;
           });
```

member initialization

initialize members inline

member initialization

Via member initialization you can provide a **default value** for member variables that will be used if the constructor does not initialize them.

Although you can then default the default constructor, the constructor will not be trivial.
Ordering is important.
If you use other member-variables in the initialization they must be declared above it.
It will compile otherwise, but upon a construction using it you are entering the domain of undefined behavior.

member initialization examples

```cpp
struct Foo
{
    Foo();
    Foo(int a);

    std::string A = "forty";
    std::string B = A + "two";
    std::string C = B + " point zero";
};

//note: not trivial!
Foo::Foo() = default;
Foo::Foo(int a)
    : A(std::to_string(a))
    //B will be std::to_string(a) + "two"
    , C(B)
    //C will be std::to_string(a) + "two"
{}
```

raw string literals

string literals without escape sequences

raw string literals

Raw string literals are string-literals in the form

```
R"PREFIX(string)PREFIX"
```

where *PREFIX* is any character but ‘ ’ ‘(’ ‘)’ ‘\’ and the control characters for horizontal and vertical tab, form feed and newline .

The *PREFIX*-string may be at most 16 characters long.

raw string literals examples

```
std::string lion = R"L(\rawr\)L";
//lion == "\\rawr\\"

std::string newline = R"(
)";
//newline == "\n"
//(independent of OS-specific file newline, e.g. \r\n)

std::string dosPrompt = R"(C:\>)";
//dosPrompt == "C:\\>"

std::string optQuotedStringRegex = R"X((\w+)|"([\w\s]+)")X";
//optQuotedStringRegex == "(\\w+)|\"([\\w\\s]+)\")"

std::string mixed = R"(raw)" "\nand"
    R"(mix\n
    xed)";
//mixed == "raw\nandmix\\n\n\txed"
```

range-based `for`

syntactic sugar for container (range) iteration

range-based `for`

The range-based `for` provides a shorter syntax for iteration. Its main usage is for **containers**, but anything that returns `InputIterator`s for `begin()` and `end()` can be utilized.

range-based `for` examples

traditional `for`

```
for(auto i = begin(foo); i != end(foo); ++i)
{
    i->bar(*i);
}
```

range-based `for`

```
for(auto &e: foo)
{
    e.bar(e);
}
```

range-based `for` - custom iterator examples

`begin()` and `end()` as members

```
struct Foo
{
    iterator begin();
    iterator end();
};

for(auto x : Foo()){} //okay
```

`begin()` and `end()` as free functions

```
extern iterator begin(Foo&);
extern iterator end(Foo&);

for(auto x : Foo()){} //okay
```

Remember to provide `const` versions.

range-based `for` - Boost[1].Range

Works amazingly well with Boost.Range!

```
using namespace boost::adaptors;
std::map<std::string, whatevs> someMap = ...;
for(auto &x : someMap
              //take elements [2..10)
            | sliced(2, 10)
              //only take keys from the map
            | map_keys
              //reverse the order (so elements (10..2])
            | reversed
              //split key into individual words
            | tokenized(boost::regex(R"(\w+)"))
            )
{
}
```

[1] reputable, peer-reviewed collection of C++ libraries for general and various specific applications