What's new in C++11/14?

Cyril Soldani

Object-Oriented Programming Projects

April 5, 2020

A Short History of C++

1979 Bjarne Stroustrup develops **C with classes**.

1983 First version of C++.

1998 First ISO **standard** (C++98, *regular* or *old* C++).

2003 Small fixes.

2011 **C++11** brings significant changes and new features.

2014 Small fixes.

2017 New features and library cleanup.

2020 Next major version, lots of proposed additions.

C++11 most important changes are

- New syntax to make code more legible.
- New semantics to make code more efficient/flexible.
- Extensions of the standard library, including:
  - new/improved containers;
  - new algorithms;
  - built-in threading support;
  - smart pointers to ease memory management.
- More powerful templates.

How to use it

clang++/g++

C++11 might already be the default.
If in doubt, or for reliability, specify:

-std=c++98 to ensure C++98 compatibility.

-std=c++11 for C++11.

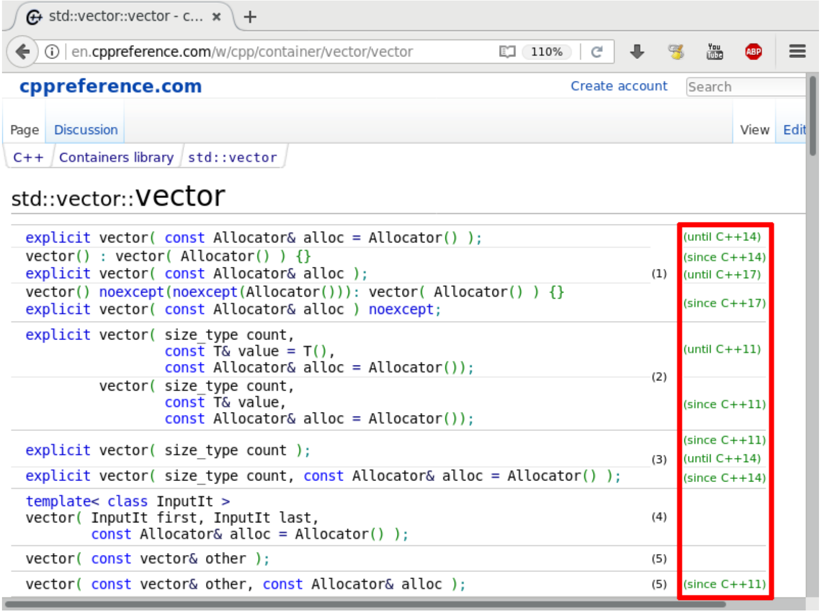
-std=c++14 for C++14.

Older g++ versions might require `c++0x` or `c++1y` instead.

Look the documentation for other compilers, to check:

- which arguments to use;
- if C++11/14 is supported at all, and with which features!

RTFM: Read The *Fine* Manual


std::vector::vector - c... ×

en.cppreference.com/w/cpp/container/vector/vector 110%

cppreference.com Create account Search

Page Discussion View Edit

C++ / Containers library / std::vector

std::vector::vector

Declaration		Version
`explicit vector( const Allocator& alloc = Allocator() );`	(1)	(until C++14)
`vector() : vector( Allocator() ) {}` `explicit vector( const Allocator& alloc );`	(1)	(since C++14) (until C++17)
`vector() noexcept(noexcept(Allocator())): vector( Allocator() ) {}` `explicit vector( const Allocator& alloc ) noexcept;`	(1)	(since C++17)
`explicit vector( size_type count, const T& value = T(), const Allocator& alloc = Allocator());`	(2)	(until C++11)
`vector( size_type count, const T& value, const Allocator& alloc = Allocator());`	(2)	(since C++11)
`explicit vector( size_type count );`	(3)	(since C++11) (until C++14)
`explicit vector( size_type count, const Allocator& alloc = Allocator() );`	(3)	(since C++14)
`template< class InputIt > vector( InputIt first, InputIt last, const Allocator& alloc = Allocator() );`	(4)	
`vector( const vector& other );`	(5)	
`vector( const vector& other, const Allocator& alloc );`	(5)	(since C++11)

auto keyword for compiler-inferred types

With auto, the compiler deduces variable's type from the right-hand side:

```
auto i = 42; // i has int type
auto l = 42L; // l has long int type
```

Pointers can be deduced, or specified explicitly:

```
auto p1 = new MyClass(); // p1 has type MyClass*
auto *p2 = new MyClass(); // p2 also has type MyClass*
```

auto keyword for compiler-inferred types

With auto, the compiler deduces variable's type from the right-hand side:

```
auto i = 42; // i has int type
auto l = 42L; // l has long int type
```

Pointers can be deduced, or specified explicitly:

```
auto p1 = new MyClass(); // p1 has type MyClass*
auto *p2 = new MyClass(); // p2 also has type MyClass*
```

... but references are **not** picked up!

```
int& f();
auto i = f(); // i has type int, not int&
auto &j = f(); // j has type int&
```

`auto` preserves constness only for references

`const` is not picked up:

```
const int foo();
auto i = foo(); // i has type int, not const int
i = 42; // Legal
const auto j = foo(); // j has type const int
j = 1984; // Compiler error
```

... except for references:

```
int& foo();
const int& bar();
auto &i = foo(); // i has type int&
i = 42; // Legal
auto &j = bar(); // j has type const int&
j = 1984; // Compile error
```

auto can make code more compact and more legible

C++98

```cpp
for (std::list<MyClass>::const_iterator it = xs.begin();
        it != xs.end(); ++it) {
    sum += it->value();
}
```

C++11

```cpp
for (auto it = xs.cbegin(); it != xs.cend(); ++it) {
    sum += it->value();
}
```

Note the addition of `cbegin`/`cend` to disambiguate between `iterator` and `const_iterator`.

C++14 allows even more type deduction using `auto`.

auto simplifies generic programming

C++98

```
template<typename Builder, typename Built>
void process(Builder& builder) {
    Built val = builder.make();
    // Do some more things with val
}
```

Built type can be deduced from Builder type, but must be specified explicitly.

C++11

```
template<typename Builder>
void process(Builder& builder) {
    auto val = builder.make();
    // Do some more things with val
}
```

Suffix return type syntax

```
T someFunc(int i, const MyObject *myObject);
```

can now also be written

```
auto someFunc(int i, const MyObject *myObject) -> T;
```

decltype extracts the type from an expression

```cpp
for (decltype(v.size()) i = 0; i < v.size(); ++i) {
  // Process v[i]
}
```

This is especially useful with templates, in conjunction with suffix return type syntax:

```cpp
template<typename Builder>
auto process(Builder& builder) -> decltype(builder.make())
{
    auto val = builder.make();
    // Do some more things with val
    return val;
}
```

Why is the suffix return type syntax needed here?

`decltype` extracts the type from an expression

```
for (decltype(v.size()) i = 0; i < v.size(); ++i) {
  // Process v[i]
}
```

This is especially useful with templates, in conjunction with suffix return type syntax:

```
template<typename Builder>
auto process(Builder& builder) -> decltype(builder.make())
{
    auto val = builder.make();
    // Do some more things with val
    return val;
}
```

Why is the suffix return type syntax needed here?

`builder` would not be in scope in usual return type position!

Ranged `for` loops

C++98

```
for (std::list<int>::const_iterator it = xs.begin();
        it != xs.end(); ++it) {
    doSomethingWithInt(*it);
}
```

C++11

```
for (auto i : xs) {
    doSomethingWithInt(i);
}
```

Beware

What are the problems here?

```
std::list<MyBigHeavyObject> xs;
for (auto x : xs)
    x.modifyElement();
```

Beware of implicit copies when using **auto**

One slow copy per iteration

```
std::list<MyBigHeavyObject> xs;
for (auto x : xs)
    x.modifyElement();
```

- `x` is a **copy** of corresponding element of `xs`.
- Copying can be slow.
- The original element is **not modified**!

Beware of implicit copies when using `auto`

One slow copy per iteration

```
std::list<MyBigHeavyObject> xs;
for (auto x : xs)
    x.modifyElement();
```

- `x` is a **copy** of corresponding element of `xs`.
- Copying can be slow.
- The original element is **not modified**!

Use a reference to modify element

```
for (auto &x : xs)
    x.modifyElement();
```

Use a `const` reference to avoid copying

```
for (const auto &x : xs)
    x.someNonModifyingOperation();
```

override indicates that a function overrides another one

```
struct A {
    virtual void foo();
    void bar();
};

struct B : A {
    void foo() const override; // Error: A::foo is not const
                               // (signature mismatch)
    void foo() override; // OK: B::foo overrides A::foo
    void bar() override; // Error: A::bar is not virtual!
};
```

- Makes developer intent clear.
- Allows compiler to detect errors.

`final` forbids overrides in derived classes

```cpp
struct Base {
    virtual void foo();
};

struct A : Base {
    // Base::foo is overridden and it is the final override
    void foo() final;
    // Error: non-virtual function cannot be final
    void bar() final;
};

struct B final : A // struct B is final
{
    void foo() override;
    // Error: cannot be overridden as it's final in A
};

struct C : B // Error: B is final
{
};
```

Type-safe enums

```
enum class Gender { Female, Male, Undetermined };
Gender gender = Gender::Male;
switch (gender) {
    case Gender::Female:
        break;
    case Gender::Male:
        break;
    // Warning: unhandled case Undetermined
}
```

Scoped enums are

- type-safe:

```
int i = Gender::Undetermined; // Type error
Gender g = 1; // Type error
```

- scoped: the `enum` introduces a new namespace for its variants.

You can specify underlying representation if needed:

```
enum class MyEnum : uint8_t { FortyTwo = 42, Other };
```

List initialization

C++98

```
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
```

C++11

```
vector<int> v = { 1, 2, 3 };
```

Available for your own objects too, just implement a constructor with `initializer_list`:

```
template<typename T> struct MyVector {
    vector<T> v;
    MyVector(initializer_list<T> xs) {
        v.insert(back_inserter(v), xs.begin(), xs.end());
    }
};
```

Uniform initialization

You can now use {} instead of ().

- Beware that if an `initializer_list` constructor is present, it will be called!
- {} forbids narrowing conversions.
- Can solve the *most-vexing-parse* problem.
- Don't mix with `auto`, would infer `initializer_list` type.

```
struct Point { Point(double x, double y, double z); /* ... */ };
Point p { 4.2, 19.84, 3.14 };

vector<int> v(5); // v contains 0 five times, i.e. { 0, 0, 0, 0, 0 }
vector<int> w{5}; // Calls initializer_list constructor, w is just { 5 }

int i(3.14); // Compiles fine, number is truncated
int j{3.14}; // Error: narrowing conversion
int xs[] = { 1, 2, 3.4 }; // Error: narrowing conversion, BREAKING CHANGE!

A a(B()); // Most-vexing parse: this is a function declaration!
a.f(); // Error!
A a{B()}; // Calls B constructor, and pass B object to A constructor
```

Lambdas

Lambda functions

- can be inlined (contrarily to function pointers);
- can be defined on-the-fly, and anonymous;
- can **capture** variables in the enclosing block;
- can be stored;
- are useful for manipulation of data structures.

`[captures](arguments) -> ReturnType { body }`

```
vector<int> xs = { 1, 2, 3, 4 };
int offset = 42;
for_each(begin(xs), end(xs), [offset](int &x) { x += offset; });
for_each(begin(xs), end(xs), [](int x) { cout << x << endl; });

vector<int> ys;
auto isEven = [](int n) -> bool { return (n % 2 == 0); };
copy_if(begin(xs), end(xs), back_inserter(ys), isEven);
```

Lambda capture modes

Captured variables can be captured **by value** or by (possibly `const`) **reference**.

```
double sum = 0.0;
auto addToSum = [&sum](double x) { sum += x; };
for_each(begin(xs), end(xs), addToSum);
```

You can also specify a default capture mode, which is used for all variables that are not explicitly specified:

= captures by value.

& captures by reference.

```
double sum = 0.0;
for_each(begin(xs), end(xs), [&](double x) { sum += x; });
```

Generally avoid default capture by reference, which is dangerous.

constexpr allows compile-time constant expressions

A **constexpr** can only refer to literal constants, and other **constexpr**s.

```
constexpr unsigned imageSize(
        unsigned width, unsigned height, unsigned nChannels,
        unsigned bitsPerPixel) {
    unsigned bytesPerPixel = (bitsPerPixel % 8 ==  0) ?
        (bitsPerPixel / 8) : (bitsPerPixel / 8 + 1);
    return width * height * nChannels * bytesPerPixel;
}
// ...
uint8_t imgBuf[imageSize(1024, 768, 3, 8)];
```

Move semantics

A common problem: functions creating output.

C++98 ways to multiply matrices

```
Matrix operator*(const Matrix& lhs, const Matrix& rhs);
// Ouch! Full matrix copy on return => slow!

void matMul(const Matrix& lhs, const Matrix& rhs, Matrix& output);
// Cumbersome syntax, mixes inputs and outputs.
// User needs to pre-allocate output matrix, with the right size!

Matrix* operator*(const Matrix& lhs, const Matrix& rhs);
// User needs to remember deleting the output matrix.
// Well, unless it is not a temporary static buffer he should copy!

boost::shared_ptr<Matrix>
    operator*(const Matrix& lhs, const Matrix& rhs);
// Clear intent, no manual memory management, but added overhead.
```

Move semantics

A common problem: functions creating output.

C++98 ways to multiply matrices

```
Matrix operator*(const Matrix& lhs, const Matrix& rhs);
// Ouch! Full matrix copy on return => slow!

void matMul(const Matrix& lhs, const Matrix& rhs, Matrix& output);
// Cumbersome syntax, mixes inputs and outputs.
// User needs to pre-allocate output matrix, with the right size!

Matrix* operator*(const Matrix& lhs, const Matrix& rhs);
// User needs to remember deleting the output matrix.
// Well, unless it is not a temporary static buffer he should copy!

boost::shared_ptr<Matrix>
    operator*(const Matrix& lhs, const Matrix& rhs);
// Clear intent, no manual memory management, but added overhead.
```

C++11

```
Matrix operator*(const Matrix& lhs, const Matrix &rhs);
// Returned matrix is no more copied, it is MOVED!
```

Move constructors

&& denotes a reference to a **r-value**.

```
class Matrix {
  // ...
  virtual ~Matrix() { delete[] data; }
  Matrix(Matrix&& origin) : data(origin.data),
          nRows(origin.nRows), nColumns(origin.nColumns)
  {
      origin.data = nullptr;
      origin.nRows = origin.nColumns = 0;
  }

private:
    double *data;
    unsigned nRows;
    unsigned nColumns;
};
```

Similarly, there is now also a **move assignment** operator.

r-values from l-values: `std::move`

- Move semantics are not just for return values.
- You can write functions expecting moved arguments.
- But how to pass them regular (*i.e.* l-value) objects?
- Using `std::move`.

```
void takeResponsibilityFor(MyBigHeavyObject&& moved) {
    // ...
}
// ...
MyBigHeavyObject heavy;
takeResponsibilityFor(heavy); // Error: heavy is a l-value!
takeResponsibilityFor(std::move(heavy));
// Move heavy into argument
```

- A value should not be used anymore in original scope after being moved.
- Good object design will enforce that (see `unique_ptr` for example).

unique_ptr replaces unsafe and deprecated auto_ptr

- It represents **exclusive ownership**.
- The ownership model is enforced though move semantics.
- Apart from that, it is used like a regular pointer.
- Very light-weight wrapper, mostly no performance cost.

```
unique_ptr<MyObject> p1(new MyObject());
unique_ptr<MyObject> p2 = p1; // Error: cannot copy unique pointers!
unique_ptr<MyObject> p3 = move(p1);
// p1 is now nullptr, and should not be used anymore

// Custom destructor (built-in RAII)
unique_ptr<FILE, decltype(&fclose)> f(fopen("file.txt", "r"), &fclose);
// fclose will be called automatically before f is destroyed

// Safer and cleaner alternative with C++14
auto p = make_unique<MyObject>();
```

share_ptr allows shared ownership

- It uses **reference counting** to know when to delete the pointed-to object.
- Always use `make_shared` to create shared pointers (also in C++11).
- You can use `weak_ptr` to break cycles. A `weak_ptr` keeps a reference to the object, but won't prevent deletion.
- When using a `weak_ptr`, call `lock()` to transform it into a `share_ptr` (avoid premature deletion).

Modern C++ avoids `new`/`delete`

The smart pointers can replace most, if not all use cases for explicit `new` and `delete`.

You can now call other constructors from a constructor

C++98

```
class C {
public:
    C() { init(42); }
    C(int i) { init(i); }
private:
    void init(int i) { /* Actual initialization code */ }
    // ...
};
```

C++11

```
class C {
public:
    C() : C(42) { }
    C(int i) { /* Actual initialization code */ }
private:
    // ...
};
```

You can now use initializers for non-static fields

C++98

```
struct C {
  C() : c('a'), i(42), d(3.14159265) { }
  C(bool flag) : c('a'), i(42), d(3.141593) { /* ... */ }
  char c;
  int i;
  double d;
};
```

Violates the DRY principle: tedious and error-prone.

C++11

```
struct C {
  C() { }
  C(bool flag) { /* ... */ }
  c = 'a';
  i = 42;
  d = 3.14159265;
};
```

default, delete and delegated constructors

- = delete will delete a constructor.
- = default will synthesize default for a constructor/destructor.
- You can inherit parent class constructor with using Parent::Parent.

```
struct Parent {
    Parent() = default;
    virtual ~Parent() = default;
    Parent(int i) { /* ... */ }
};
struct Child : Parent {
    using Parent::Parent; // Inherits parent constructors
    Child& operator=(const Child&) = delete;
        // Disable assignment operator
    Child(const Child&) = delete; // Disable copy constructor
};
```

Nested templates gain a nicer syntax

C++98

```
vector<vector<int> > matrix;
```

C++11

```
vector<vector<int>> matrix;
```

No more space needed between the right angle brackets!

nullptr is like 0 and NULL, but has only pointer type

```
void f(int); // #1
void f(const char *); // #2

f(0); // Which is called, #1 or #2?
f(nullptr); // Unambiguously call #2

int i = nullptr; // Compile error
```

nullptr can only be converted to a pointer type, or to bool.

`explicit` conversion constructors

We already saw `explicit` in the lecture about objects-as-values, but it is only available since C++11. What does it do?

explicit conversion constructors

We already saw explicit in the lecture about objects-as-values, but it is only available since C++11. What does it do?

explicit disables implicit conversions using conversion constructors or operators.

C++98

```
struct MyType {
  MyType(int i) { /* ... */ }
};
void f(MyType);
f(42); // Silently pass MyType(42) to f()
```

C++11

```
struct MyType {
  explicit MyType(int i) { /* ... */ }
};
void f(MyType);
f(42); // Error: f() expects MyType, not int!
f(MyType(42)); // Still fine
```

There is a lot more going on

- Perfect forwarding, move semantics on steroids.
- Variadic templates.
- Threading interface built into the language.
- New containers.
- New algorithms.
- New string literals.
- User-defined literals.
- Regular expressions.
- `static_assert`, compile-time assertions.
- Type traits (*e.g.* `has_virtual_destructor`).
- `using` can replace `typedef`.
- ...

C++17 and C++20

- nested namespaces C++17
- de-structuring bindings
- improved constexpr
- UTF-8 character literals
- std::variant typesafe union
- std::optional
- std::filesystem
- std::byte to avoid implicit conversion hasard
- constexpr, consteval, constinit C++20
- modules replace CPP (no more *#include*)
- concepts make template assumptions explicit
- ranges and views improve on iterators
- python-like string formatting
- coroutines? async, await, yield