Modern C++ (C++11/14)
Geeks Anonymes

Damien Gerard

October 19th, 2016

Introduction

“C++11 feels like a new language.” Bjarne Stroustrup (inventor of C++)

Labels

11 since C++11

14 since C++14

17 since C++17

R that ain't for kids

Range-based for loop

11

`for` statement syntax extended to allow for easy iteration over a range of elements:

```
int myArray[5] = {1, 2, 3, 4, 5};
for(int x : myArray) {
  std::cout << x << std::endl;
}
for(int& x : myArray) {
  x *= 2; // double value and save
}
for(auto& x : myArray) { // auto = int
  x *= 2;
}
```

auto

Constant iterator over a vector:

```
std::vector<int> vec(4,100); // four ints with value 100
for(std::vector<int>::const_iterator it = vec.cbegin();
    it != vec.cend(); ++it)
{
  std::cout << *it << std::endl;
}
```

auto for type deduction:

```
std::vector<int> vec(4,100); // four ints with value 100
for(auto it = vec.cbegin(); it != vec.cend(); ++it) {
  std::cout << *it << std::endl;
}
```

11

Null pointer literal

Constant `0` had the double role of constant integer and null pointer constant.

```
void f(int*);
void f(int);

f(NULL); // calls f(int) or error: ambiguous
```

If `NULL` defined as 0, `f(NULL)` calls `f(int)`. Most programmers would expect `f(int*)` because `NULL` means a *null pointer* in their mind.

Null pointer literal

11

`nullptr` is a pointer literal of type `nullptr_t`, which is implicitly convertible and comparable to any pointer type. It is not implicitly convertible or comparable to integral types, except for `bool`.

```
char* pc = nullptr; // compiles
int* pi = nullptr;  // compiles
bool b = nullptr;   // compiles, b is false
int i = nullptr;    // error

f(nullptr); // calls f(int*), not f(int)
```

`f(nullptr_t)` will actually call `f(int*)` using implicit conversion.

Initializer lists

C++03 inherited the initializer-list feature from C.

```
struct Widget {
  float first;
  int second;
};

Widget w1 = {0.43f, 10}; // first=0.43f and second=10
Widget w2[] = {{13.4f, 3}, {43.2f, 29}};
```

Useful albeit limited to Plain Old Data (POD) structs and classes.

Uniform initialization

11

Initializer-lists for all classes, including standard containers.

```
std::vector<std::string> v = { "Hello", "world" };
std::vector<std::string> v({ "Hello", "world" });
std::vector<std::string> v{ "Hello", "world" };
```

```
int val = 5.2; // automatic narrowing
int val{5.2};  // error or warning: type 'double'
  // cannot be narrowed to 'int' in initializer list
  // insert an explicit static cast static_cast<int>( )
```

```
Widget getWidget() {
  return {0.43f, 10}; // no need for explicit type
}
```

Most vexing parse

```
struct Widget {
  Widget();
};
struct WidgetKeeper {
  WidgetKeeper(Widget w);
};
WidgetKeeper wKeeper(Widget()); // most vexing parse
```

Ambiguous call:

1. new instance of Widget, sent to constructor of WidgetKeeper;
2. function declaration: name "wKeeper", return type WidgetKeeper, single (unnamed) parameter which is a function returning type Widget.

Most vexing parse

Most developers expect the first, but the C++ standard requires it to be interpreted as the second.

Workaround in C++03 to ensure the first interpretation:

```
WidgetKeeper time_keeper( (Widget()) );
```

C++11's uniform initialization syntax:

```
WidgetKeeper time_keeper{Widget{}};
```

11

Returning multiple values

```
void divide(int dividend, int divisor, int& quotient,
            int& remainder)
{
  quotient = dividend / divisor;
  remainder = dividend % divisor;
}
```

Or even worse, return `quotient` through the returned value and `remainder` as a reference variable:

```
int divide(int dividend, int divisor, int& remainder) {
  remainder = dividend % divisor;
  return dividend / divisor;
}
```

Returning multiple values

Or, but rather heavy:

```
struct divideResult {
  int quotient;
  int remainder;
};

divideResult divide(int dividend, int divisor) {
  return { dividend / divisor, dividend % divisor };
}

divideResult result = divide(10, 3);
std::cout << result.quotient << std::endl;
std::cout << result.remainder << std::endl;
```

11

Returning multiple values

```
std::tuple<int,int> divide(int dividend, int divisor) {
  return std::make_tuple(dividend / divisor,
                         dividend % divisor);
}

int quotient, remainder;
std::tie(quotient, remainder) = divide(10, 3);
```

11

C++17's structured bindings (Clang-4.0 only, as of today):

```
auto [quotient, remainder] = divide(10, 3);
std::cout << quotient << std::endl;
std::cout << remainder << std::endl;
```

17

std::unordered_*

11

`std::set`, `map`, `multiset`, `multimap` associative containers are implemented as balanced trees. The `std::unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap` alternatives are implemented as hash tables.

Including hash tables was one of the most recurring requests. It was not adopted in C++03 due to time constraints only. Although hash tables are less efficient than a balanced tree in the worst case (in the presence of many collisions), they perform better in many real applications.

emplace, emplace_back

11

`emplace` and `emplace_back` can construct an element *in-place*.

```
auto employees = std::unordered_map<int, std::string>{};
auto e1 = std::pair<int, std::string>{1, "John Smith"};

employees.insert(e1);
employees.insert(std::make_pair(2, "Mary Jones"));
employees.emplace(3, "James Brown"); // construct in-place

for (const auto& e : employees) {
  std::cout << e.first << ": " << e.second << std::endl;
}
```

In-class initialization

In C++03, in-class initialization on static const members of integral or enumeration type only.

```
struct Widget {
  static const int a = 7;      // compiles
  float b = 7.2;      // error: not static const integral
  const int c = 7;              // error: not static
  static int d = 7;             // error: not const
  static const float e = 7.2; // error: not integral
  const static int arr[] = { 1, 2, 3 };
    // error: must be initialized out of line
};
```

In-class initialization

11

C++11 allows some of them:

```
struct Widget {
  static const int a = 7;
  float b = 7.2;
  const int c = 7;
  static int d = 7;      // still fails
  static float e = 7.2; // still fails
  constexpr static int arr[] = { 1, 2, 3 };
  constexpr static std::complex<double> f = { 1, 2 };
};
```

A non-const static variable still has to be initialized outside the class with

```
int Widget::d = 7;
float Widget::e = 7.2;
```

ODR-use

R

An object is odr-used if its address is taken, or a reference is bound to it. If a const or constexpr static data member is odr-used, a redefinition (no initializer permitted) at namespace scope is required.

```
struct Widget {
  static const int n = 1;
  static constexpr int m = 4;
};
const int& f(const int& r);
// call f(Widget::n) or f(Widget::m) somewhere
// redefinition at namespace scope required:
const int Widget::n, Widget::m;
```

11

C++17 alleviates this constraint for `constexpr`. A static data member declared constexpr is implicitly `inline` and needs not be redeclared at namespace scope.

17

constexpr

Declare something that can be evaluated down to a constant:

```
constexpr double pi() { return std::atan(1) * 4; }
constexpr double getCircleSurface(double r) {
  return pi() * r * r;
}
const double circleSurface = getCircleSurface(0.5);
```

11

or

```
const float oneEighty = degreesToRadians(180.0f);
```

Since C++14, constexpr functions may have more than one line.

14

Call other constructors

In C++03, other constructors could not be called. Workaround: call a common member function

```
class Widget  {
  int number_;
  void construct(int number) { number_ = number; }
public:
  Widget(int number) { construct(number); }
  Widget() { construct(42); }
};
```

Call other constructors

Wrong workaround:

```
class Widget  {
  int number_;
public:
  Widget(int number) : number_(number) { }
  Widget() {
    Widget(42);
    // this->number_ = 0 or something undefined
  }
};
```

What is wanted but does not compile:

```
Widget() : Widget(42) { }
```

Call other constructors

11

In C++11 other peer constructors may be called (“delegation”).

```
class Widget  {
  int number_;
public:
  Widget(int number) : number_(number) { }
  Widget() : Widget(42) { }
};
```

An object is constructed once *any* constructor finishes execution. Since multiple constructors are allowed to execute, each delegating constructor will be executing on a fully constructed object of its own type.

Call or import base class constructors

11

Call base class constructor:

```
struct Base {
  Base(int number);
};
struct Derived : Base {
  Derived(int number) : Base(number) { }
};
```

Import all base class constructors:

```
struct Derived : Base {
  using Base::Base
};
// can call Derived(42);
```

Override a base class method

```
struct Animal  {
  char* say() const { return "??"; }
};
struct Cow : Animal {
  char* say() const { return "moo"; }
};
struct Pig : Animal {
  char* say() const { return "oink"; }
};
std::vector<Animal*> animals{new Cow, new Pig};
for(const Animal* a : animals) {
  std::cout << a->say() << std::endl;
}
```

Prints “??” for both the cow and the pig.

Dynamic polymorphism

```
struct Animal  {
  virtual char* say() const { return "??"; }
};
struct Cow : Animal {
  virtual char* say() const { return "moo"; }
};
struct Pig : Animal {
  virtual char* say() const { return "oink"; }
};
```

Prints “moo” and “oink”. What if `const` is forgotten in the `Pig`’s `say()`? It will compile and print “moo” and “??”.

Typo in derived class

The following code compiles but contains no virtual function overrides.

```
struct Base {
  void f1() const;
  virtual void f2() const;
  virtual void f3(int x);
}

struct Derived : Base {
  void f1() const;
  virtual void f2();
  virtual void f3(unsigned int x);
}
```

overrides

11

Make explicit that a derived class is expected to override a base class:

```
struct Derived : Base {
  void f1() const override;
  virtual void f2() override;
  virtual void f3(unsigned int x) override;
}
```

The compiler will complain about all the overriding-related issues.

final

11

Prevent overriding:

```
struct Base {
  virtual void f() final;
};
struct Derived : Base {
  virtual void f(); // error: 'f' overrides a 'final' function
};
```

Prevent derivation:

```
struct Base final { };
struct Derived : Base { }; // error: 'Base' is marked 'final'
```

This could be achieved in C++03 with `private virtual` inheritance.

Explicitly deleted functions

11

```
struct Widget {
  void f(double i);
  void f(int) = delete;
};
Widget w;
w.f(3.0); // ok
w.f(3);   // error: explicitely deleted
```

Explicitely deleted special member functions:

```
struct Widget {
  Widget() = default;
  Widget(const Widget&) = delete;
  Widget& operator=(const Widget&) = delete;
};
```

Unrestricted unions

11 R

In C++03, unions cannot contain any objects that define a non-trivial constructor or destructor. C++11 lifts some restrictions.

```
struct Widget {
  int x_;
  Widget() {}
  Widget(int x) : x_(x) {}
};

union U {
  double f;
  Widget w; // illegal in C++03, legal in C++11
};
U u; // error: call to implicitely deleted default constructor
// note: default constructor of 'U' is implicitely deleted
// because field 'w' has a non-trivial default constructor
```

Unrestricted unions

11 R

A constructor for the `union` must be manually defined:

```
union U {
  double f;
  Widget w;
  U(int x) : w(x) { }
};
U u(3);
```

Strongly typed enumerations

11

C++03 enumerations are not type-safe. It allows the comparison between two enum values of different enumeration types. Type-safe enumeration:

```
enum class Enum1 {
  Val1 = 100,
  Val2 // = 101
};
```

Override the default underlying type:

```
enum class Enum2 : unsigned long {Val1, Val2};
```

Other examples:

```
enum Enum3; // Underlying type cannot be determined (C++03/11)
enum class Enum4; // Underlying type is int (C++11)
```

Template type deduction for classes

Template class:

```
template<typename T> // "typename" and "class" are synonymous
class Stack {
  std::vector<T> elems_;
public:
  void push(const T& elem) { elems_.push_back(elem); }
  T top() const { return elems_.back(); }
  void pop() { elems_.pop_back(); }
};

Stack<double> myStack; // T = double
Stack<int*> myStack; // T = int*
```

One `Stack` code for `double` and another `Stack` code for `int*`.

Template type deduction for functions

```
template <typename T>
inline T max(T a, T b) {
  return a > b ? a : b;
}
std::cout << max(3.0, 7.2) << std::endl; // T = double
std::cout << max("hello", "world") << std::endl;
```

```
struct Widget {
  int x_;
  Widget(int x) : x_(x) { }
};
Widget w1(3), w2(4);
std::cout << max(w1, w2) << std::endl;
// error: invalid operands to binary expression
```

Template partial specialization for loop unrolling

R

The template may also be a value instead of a class, allowing e.g. for compile-time loop unrolling, with partial specialization:

```
template <unsigned int N>
int factorial() {
  return N * factorial<N - 1>();
}

template <>
int factorial<0>() {
  return 1;
}
std::cout << factorial<4>() << std::endl; // yields 24
```

Since C++11, the same can be performed with `constexpr`.

11

auto for variable type deduction

11

```
std::vector<int> vec{10, 11, 12, 13};
for(auto it = vec.cbegin(); it != vec.cend(); ++it) {
  std::cout << *it << std::endl;
}
```

decltype for return type deduction


```
template<typename T, typename U>
SomeType mul(T x, U y) {
  return x*y;
}
```

The return type is "the type of x*y". How can we write that?

```
template<typename T, typename U>
decltype(x*y) mul(T x, U y) {
  return x*y;
}
```

It does not compile because x and y are not in scope. We can use a trailing return type.

decltype for return type deduction

11

Trailing return type:

```
template<typename T, typename U>
auto mul(T x, U y) -> decltype(x*y) {
  return x*y;
}
```

When `decltype` is read, `x` and `y` are now known to the compiler. `auto` is only meant to tell the compiler that it will find the return type after `->`. `auto` does not deduce anything.

Generalized return type deduction

Since C++14, `auto` can deduce the return type of functions:

```
auto floor(double x) { return static_cast<int>(x); }
```

Multiple returns are allowed, but the type must be the same:

```
auto f() {
  while(something()) {
    if(expr) {
      return foo() * 42;
    }
  }
  return bar.baz();
}
```

Generalized return type deduction

The return type deduction also works for recursive functions, provided the non-recursive return statement is before the recursive call.

```
auto f() { return f(); } // error: return type of f is unknown
```

```
auto sum(int i) {
  if (i == 1)
    return i;          // return type deduced to int
  else
    return sum(i-1)+i; // ok to call it now
}
```

decltype(auto)

14

`auto` always deduces a non-reference type (`int`, `double`...). `auto&` always deduces a reference type (`int&`, `double&`...). But `auto` cannot conditionnaly see if the return type is a reference.

`decltype` can, but requires to write an expression in its parentheses.

We can use the best of both worlds with `decltype(auto)` which can deduce if the type is a reference.

```
template<typename T, typename U>
decltype(auto) mul(T x, U y) {
  return x*y;
}
```

Copy-and-swap idiom

```
class Array {
  int* arr_;
  unsigned int size_;
public:
  Array(unsigned int size = 0) : size_(size),
  arr_(size > 0 ? new int[size]() : nullptr) { }

  Array(const Array& other) : size_(other.size_),
  arr_(other.size_ > 0 ? new int[other.size_] : nullptr)
  {
    std::copy(other.arr_, other.arr_ + size_, arr_);
  }

  ~Array() { delete[] arr_; }
};
```

Copy-and-swap idiom

Naïve implementation of `operator=`:

```
Array& operator=(const Array& other) {
  if (this != &other) {
    delete [] arr_;
    arr_ = nullptr;

    arrSize_ = other.arrSize_;
    arr_ = arrSize_ ? new int[arrSize_] : nullptr;
    std::copy(other.arr_, other.arr_ + arrSize_, arr_);
  }
  return *this;
}
```

Copy-and-swap idiom

Issues:

1. code duplication;
2. `if(this != &other)` mandatory but should not occur;
3. if `new[]` fails, `*this` will have been modified (no strong exception guarantee).

Copy-and-swap idiom

Successful solution:

```
friend void swap(Array& first, Array& second) {
  using std::swap; // enable ADL
  swap(first.size_, second.size_);
  swap(first.arr_, second.arr_);
}
Array& operator=(Array other) {
  swap( *this, other );
  return *this;
}

Array getArray(unsigned int size) { return Array(size); }

Array arr = getArray(4);
```

Copy Elisions, Returned Value Optimization

Distinguish *parameter* and *argument*:

- `other` is the *parameter* of the function;
- the array returned by `getArray(4)` is the *argument*, from which the parameter `other` is instanciated.

If the argument is a temporary object which will be destroyed, why make a copy for `other`? The argument should *become* `other`, thereby avoiding copy.

Most C++03 compilers could grasp this optimization opportunity whenever possible and elude the copy.

Guideline: Do not copy the function arguments. Instead, pass them by value and let the compiler manage the copying.

Lvalue – Rvalue

C++11 formalized everything:

- a temporary object, eligible for copy elision, is an *rvalue*;
- otherwise it is an *lvalue*.

References:

- an integer *lvalue-reference* is denoted `int&`;
- an integer *rvalue-reference* is denoted `int&&`.

Move constructor and move assignment operator

11

There are two new special member functions.

Move constructor:

```
Array(Array&& other) : Array() {
  swap( *this, other );
}
```

Move assignment operator:

```
Array& operator=(Array&& other) {
  swap( *this, other );
  return *this;
}
```

Specialize code for lvalue or rvalue

11

```
void f(int& param) {
  std::cout << "lvalue" << std::endl;
}
void f(int&& param) {
  std::cout << "rvalue" << std::endl;
}

int a;
f(a); // calls void f(int&)
f(std::move(a)); // calls void f(int&&)
```

If you can take a variable's address, it usually is an lvalue. Otherwise it is usually an rvalue.

std::move

11

`std::move` does not move anything. It casts its parameter as an rvalue.

```
struct Widget {
  Array arr_;
  Widget(const Array& param) : arr_(std::move(param)) { }
};
Array arr(3);
Widget w(arr);
```

compiles but calls the copy constructor of `Array` instead of the move constructor. `std::move` returns a `const Array&&` which cannot be casted as a non-const `Array&&`, but can be casted as a `const Array&`.

```
Array(const Array& other); // copy constructor
Array(Array&& other);      // move constructor
```

Distinguish forwarding from rvalue references

A *forwarding reference* is either an lvalue reference or an rvalue reference. A forwarding reference has the same syntax as an rvalue reference, namely "&&".

Rvalue references examples:

```
void f(Widget&& param);

Widget&& w1 = Widget();
```

Forwarding references:

```
template<typename T>
void f(T&& param);

auto&& w2 = w1;
```

Distinguish forwarding from rvalue references

11

Both forwarding references have in common the presence of type deduction. Rvalue references do not.

```
template<typename T>
void f(T&& param) {
  // Here we can take param's address
  // so param is always an lvalue
}
Widget w;
f(w); // param's type is lvalue-reference Widget&
      // param is lvalue
      // T = Widget&
f(std::move(w)); // param's type is rvalue-reference Widget&&
                 // param is lvalue
                 // T = Widget
```

std::forward

11

```
void process(Widget& lValArg);  // for lvalues
void process(Widget&& rValArg); // for rvalues

template<typename T>
void logAndProcess(T&& param) {
  auto now = std::chrono::system_clock::now();
  makeLogEntry("Calling 'process'", now);
  process(param); // always calls process(Widget&)
}

Widget w;
logAndProcess(w);
logAndProcess(std::move(w));
```

Because `param` is always an lvalue, `process(Widget&)` is always called.

std::forward

If `param`'s type is an rvalue-reference, we need to cast the lvalue `param` as an rvalue. This is what `std::forward` does.

`std::move` always casts as an rvalue; `std::forward` conditionally casts as an rvalue.

But how to distinguish? Look at `T`:

1. if param is an lvalue-reference, `T = Widget&`;
2. if param is an rvalue-reference, `T = Widget`.

`T` must be given to `std::forward` so it can decide whether to cast its parameter as an rvalue or not.

```
process(std::forward<T>(param));
```

Subtle forwarding and rvalue references

11 R

```
template<typename T>
class vector {
public:
  void push_back(T&& x);
};
```

is not a forwarding reference. Writing `std::vector<Widget> v;` causes the template to be instanciated as

```
class vector<Widget> {
public:
  void push_back(Widget&& x);
};
```

which involves no type deduction.

Subtle forwarding and rvalue references

11 R

A template forwarding reference must have the form T&&. Hence, the two following references do not qualify for being forwarding references.

```
template<typename T>
void f(const T&& param);

template<typename T>
void f(std::vector<T>&& param);
```

Subtle forwarding and rvalue references

11 R

The variadic templates of `emplace` involve type deduction.

```
template<typename T>
class vector {
public:
  template<typename... Args>
  void emplace(Args&& args);
};
```

Implementation of std::move in C++11

11 R

```
template<typename T>
typename remove_reference<T>::type&& move(T&& param) {
  using ReturnType = typename remove_reference<T>::type&&;
  return static_cast<ReturnType>(param);
}
```

`param` is a forwarding reference:

- if `param` is an lvalue-reference, `T = Widget&`;
- if `param` is an rvalue-reference, `T = Widget`.

Steps:

1. `remove_reference<T>::type` returns `Widget`;
2. `remove_reference<T>::type&&` returns `Widget&&`;
3. `static_cast<Widget&&>` casts `param` as an rvalue.

Implementation of std::move in C++14

14 R

`std::remove_reference_t<T>` replaces `typename remove_reference<T>::type`.

`std::move` can elegantly be written as

```
template<typename T>
decltype(auto) move(T&& param) {
  using ReturnType = remove_reference_t<T>&&;
  return static_cast<ReturnType>(param);
}
```

Lambda expressions


```
auto lambda = [](int x, int y) -> int { return x + y; };
std::cout << lambda(2, 3) << std::endl;
```

Components:

- `(int x, int y)` is the parameters list;
- `int` after `->` is the return type;
- `{...}` is the lambda body.

A lambda can be executed upon declaration with trailing "()":

```
std::cout
  << [](int x, int y) -> int { return x + y; }(2, 3)
  << std::endl;
```

std::for_each and std::transform

Apply a lambda to each element of an iterator:

```
void f(std::vector<int>& v) {
  std::for_each(v.begin(), v.end(),
    [](int p) { std::cout << p << std::endl; });
}
```

Modify each element with a lambda:

```
void f(std::vector<double>& v) {
  std::transform(v.begin(), v.end(), v.begin(),
    [](double d) { return d < 0.00001 ? 0 : d; });
}
```

C++03 already had `std::for_each` and `std::transform`, but cumbersome functors were necessary since lambdas were not available.

std::for_each and std::transform

11

If the body is not a single `return` expression, the type has to be explicitely stated:

```
void f(std::vector<double>& v) {
  std::transform(v.begin(), v.end(), v.begin(),
    [](double d) -> double {
      if (d < 0.0001)
        return 0;
      else
        return d;
    });
}
```

Lambda captures

11

`[]` allows to capture other variables from the scope.

```
void f(std::vector<double>& v, double epsilon) {
  std::transform(v.begin(), v.end(), v.begin(),
    [epsilon](double d) -> double {
      if (d < epsilon)
        return 0;
      else
        return d;
    });
}
```

We can capture by both reference and value.

Generalized lambda captures

An element of the capture can be initialized with =. This allows renaming of variables and to capture by moving.

```
int x = 4;
auto y = [&r = x, x = x+1]() -> int {
  r += 2;
  return x+2;
}(); // Updates ::x to 6, and initializes y to 7
```

```
std::unique_ptr<int> ptr(new int(10));
auto lambda = [value = std::move(ptr)] { return *value; };
```

`std::unique_ptr` can be moved but not copied. Capture by move is the only way to capture a `std::unique_ptr`.

Generic lambdas

14

Lambda function parameters may be declared auto.

```
auto lambda = [](auto x, auto y) { return x + y; };
```

Improved return type deduction

C++14 allows deduced return types for every function, not only those of the form `return expression`.

```
void f(std::vector<double>& v, double epsilon) {
  std::transform(v.begin(), v.end(), v.begin(),
  [epsilon](auto d) {
    if (d < epsilon)
      return 0;
    else
      return d;
  });
}
```

Remaining modern C++ topics

- type traits (std::enable_if, std::is_pointer...);
- smart pointers;
- threading facilities;
- noexcept;
- std::function;
- std::bind and std::ref;
- std::regex;
- extern template;
- inline namespace...