

Modern C++

New features in (mostly) C++11
but also 14, 17, and 20

Author: Alex Reustle

Resources

- ISO C++: <https://isocpp.org/>
- CppReference: <https://en.cppreference.com/w/>
- Cpp Core Guidelines: <https://github.com/isocpp/CppCoreGuidelines>
- *A Tour of C++*: <http://www.stroustrup.com/Tour.html>
- Compiler Explorer: <https://godbolt.org/>
- Conferences and videos:
 - CppCon: <https://www.youtube.com/CppCon>
 - C++Now: www.youtube.com/BoostCon
- Podcasts: CppCast, CppChat
- Blogs: <http://www.fluentcpp.com/>
- GCC Compiler support: <https://gcc.gnu.org/projects/cxx-status.html>



Contents

- **Great Resources**
- **History**
- **Language Features**
 - auto keyword
 - smart pointers
 - rvalue references and move semantics
 - lambda functions
- **Standard Library**
 - `<algorithms>`
 - `<filesystem>`
- **Standard Library**
 - `<regex>`
 - `<thread>`
- **Future C++20**
 - Concepts
 - Modules
 - Ranges
- **Idioms and best practices**
 - RAII
 - Cpp Core Guidelines



C++ History

Early Goals of C++:

(source) <https://isocpp.org/wiki/faq/cpp11#cpp11-goals>

“C++ has from its inception been a general-purpose programming language with a bias towards systems programming that:

- *is a better C*
- *supports data abstraction*
- *supports object-oriented programming*
- *supports generic programming”*



Pre-Modern C++ History

- 1979: Bjarne starts work on “C With Classes starts” - **Classes, public/private, constructors and destructors**
- 1984: Renamed “C++” - **Virtual functions, operator overloading, references, and I/O streams**
- 1985: First commercial release of C++
- 1998: ISO C++ standard - **Templates, exceptions, RAII, namespaces, The STL.**
- 2002: Work on revised standard (C++0x) begins
- 2003: “*Bugfix*” C++03 released



Modern C++ History

- **2011: C++0x becomes C++11, and is finally approved and released after 13 years**
 - Type deduction, smart pointers, move semantics, lambdas, updated memory model, locks, mutex
- **2014: “*Bugfix*” C++14 released. Seen as “completing” C++11**
 - Variable templates, digit separators, generic lambdas, and a few standard-library improvements
- **2017: C++17 release, Cpp Core Guidelines Formalized**
 - Structured bindings, fold expressions, a file system library, parallel algorithms, and variant and optional types
- **2020: C++20 standard to be approved, the next “Major” release like C++11:**
 - Modules, Ranges, Concepts, Contracts
 - Post-Modern C++?



Design Goals of C++11

- Stability: don't break old code
- Prefer libraries to language extensions: Not always successful
- Prefer generality to specialization: focus on improving abstraction mechanisms
- Support both experts and novices
- Increase type safety: Allow programmers to avoid type-unsafe features
- Improve performance and ability to work directly with hardware – make C++ even better for embedded systems programming and high-performance computation
- Retain Zero-Cost abstractions. Don't pay for features you don't use.
- Fit into the real world – consider tool chains, implementation cost, transition problems, ABI issues, teaching and learning, etc.

<https://isocpp.org/wiki/faq/cpp11#cpp11-specific-goals>



C++ Programming Philosophy

- Stack-based scope instead of heap or static global scope.
- Auto type inference instead of explicit type names.
- Smart pointers for ownership instead of raw pointers.
- `std::string` types instead of raw `char[]` arrays.
- Use C++ Standard Library containers like `vector`, `list`, and `map` instead of raw arrays or custom containers.
- C++ Standard Library algorithms instead of manually coded ones.



Modern C++ adds

- **Language Features:**
 - Auto type deduction, smart pointers
 - rvalue references, lambdas
- **Default support for smart Idioms:**
 - Move Semantics, RAI
- **Standard Library Modules:**
 - Algorithms, Filesystem, Concurrency, Regex



New language feature: Keyword Auto



Keyword Auto

- **auto** used as a type specifier.
- The compiler deduces the type.
- Types remain static. Cannot assign to a different type without a cast.
- Zero runtime overhead.



Keyword Auto

```
auto a  = 1 + 2;           // type is int
auto f  = 1.1;             // float 32
auto d  = double {1.1};    // double 64
auto d2 = 3.14d;           // double 64 using type suffix
auto c  = 'c';             // const char
auto s  = std::string{ "Hello World!" }; // std::string
auto s2 = "Hello World"s;  // std::string
```



Keyword Auto

// C++98

```
map<int,string>::iterator i = m.begin();
```

```
double const xlimit = config["xlimit"];
```

```
singleton& s = singleton::instance();
```

// C++11

```
auto i = begin(m);
```

```
auto const xlimit = config["xlimit"];
```

```
auto& s = singleton::instance();
```



Auto enables the range-for loop

```
//C++98  
for(auto i=begin(c); i!=end(c); ++i){  
    use(*i);  
}
```

```
//C++11  
for(auto& e : c){  
    use(e);  
}
```



C++17 adds structured bindings

// C++11/14

- `tuple<int, string> func();`

`auto tup = func();`

`int i = get<0>(tup);`
`string s = get<1>(tup);`

`use(s, ++i);`

// C++17

- `tuple<int, string> func();`

`auto [i, s] = func();`

`use(s, ++i)`



Reasons to use Auto

- **Consistency**
 - Always uses the correct type, even when refactoring
- **Performance**
 - Never narrows on implicit casts
- **Safety**
 - Type mismatches are impossible
- **Convenience**
 - Some types are long or hard to type
 - Some types are only known by the compiler



New language feature: Smart Pointers



Why Smart Pointers?

- **Raw pointers and references are dangerous when used for object lifetime and ownership.**
 - Use after free
 - Double delete
 - Memory leak
- **But raw pointers and references are great for non-owning operations.**



Smart Pointers

- `unique_ptr<T>`
- `shared_ptr<T>`



Smart Pointers

- **Not `auto_ptr`**
- **Never `auto_ptr`**
- **Deprecated in C++11**
- **Removed in C++17**



Smart Pointers

- `unique_ptr<T>`

- Default choice when creating objects.
- Owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope.
- Programmer does not worry about deleting. RAII
- Cannot be copied, it's unique.

```
auto p = make_unique<T>();
```



Smart Pointers

- **shared_ptr<T>**
 - Use only for objects shared by multiple scopes.
 - Several **shared_ptr** objects may own the same object.
 - The object is destroyed and its memory deallocated when the last remaining **shared_ptr** owning the object is destroyed.
 - Reference counted pointer.

```
auto p = make_shared<T>;
```



Reasons to use Smart Pointers

- **unique_ptr** creation and use in scope has zero overhead over raw pointers.
- Avoids many common pitfalls of pointer based ownership.
- **shared_ptr** ensures safe use of a pointer owned by many actors, at a modest cost.



When not to use shared pointers

- **Don't bother when the operation isn't owning**
 - Example: Function parameters
 - Prefer `const T&` for performance
- **Only use `shared_ptr` when needed**
 - Reference counting across scope is expensive



Herb Sutter CppCon 2014 advice for Modern C++ style

Don't Use Owning *, *new*, or *delete*

▶ C++98:

```
widget* factory();  
void caller() {  
    widget* w = factory();  
    gadget* g = new gadget();  
    use( *w, *g );  
    delete g;  
    delete w;  
}
```

red ⇒ now “mostly wrong” 😊

▶ Don't use owning *, *new* or *delete*.

- ▶ Except: Encapsulated inside the implementation of low-level data structures.

▶ Modern C++:

```
unique_ptr<widget> factory();  
void caller() {  
    auto w = factory();  
    auto g = make_unique<gadget>();  
    use( *w, *g );  
}
```

▶ For “new”, use *make_unique* by default, *make_shared* if it will be shared.

▶ For “delete”, write nothing.



New language features:
rvalue references
move semantics



Motivating example

- Code up a function that performs a costly analysis and produces a large object.
- In Python there's essentially one option:

```
def make_big_vector(...)  
  
    result = make_big_vector()
```
- C++ has more control than that!



Motivating example

C++98:

- //option 1: return by value: high cost to copy vector
`vector<int> make_big_vector();`
`vector<int> result = make_big_vector();`
- //option 2: return by pointer: no copy, remember to delete
`vector<int>* make_big_vector();`
`vector<int>* result = make_big_vector();`
- //option 3: pass by reference: no copy, but caller needs a named object
`void make_big_vector(vector<int>& out);`
`vector<int> result;`
`make_big_vector(result);`



Motivating Example

- C++11 makes option 1 fast, safe and efficient.

```
vector<int> make_big_vector();  
auto result = make_big_vector();
```

- **Guaranteed not to copy the vector**
 - As efficient as options 2, 3
- **Type must have a move constructor**
 - Often provided by the compiler by default
 - Can be user defined, takes rvalue reference
 - Called “Move Semantics”



What is an rvalue?

It's all about assignment

First approximation, good for intuition:
left side of assignment [=] right side of assignment

lvalue = rvalue

lvalue <-- rvalue



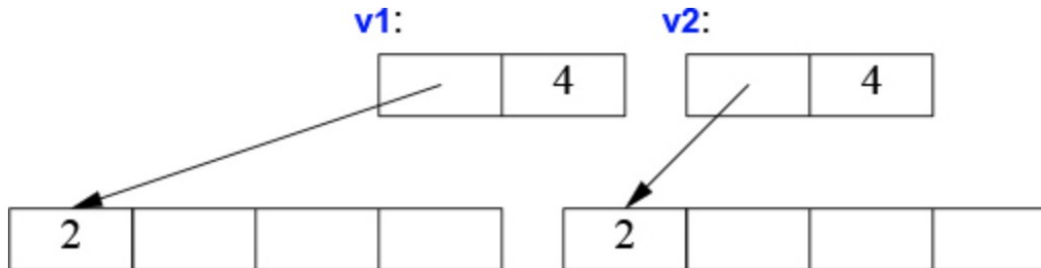
What is an rvalue really?

- An lvalue is a location with an address that can hold an rvalue.
- An rvalue is what will go in the address:
 - the number stored in int
 - a string literal in source code
 - the return value of a function



C++98 version of option 1

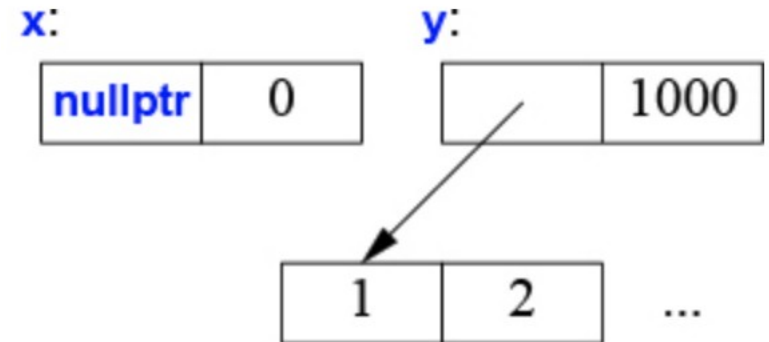
- ```
vector<int> make_big_vector(){
 vector<int> v1 = std::vector<int>(4);
 return v1; // <-I'm an rvalue
};
auto v2 = make_big_vector();
```





# C++11 version of option 1

- ```
vector<int> make_big_vector(){  
    vector<int> x = std::vector<int>(1000);  
    return x;           // <-I'm an rvalue  
};  
auto y = make_big_vector();
```



Move Operations

- ```
class Vector {
 // ...
 Vector(Vector&& a): // move constructor
 elem = a.elem,
 size = a.size
 {
 a.elem = nullptr;
 a.size = 0;
 }
 Vector& operator=(Vector&& a); // move assignment
};
```
- && mean “rvalue reference”, not logical AND.
- Move operations “move” the contents of one object into another and make the source object ready for cleanup.



New language feature: lambda functions



# Lambdas

- C++98 has function objects (aka functors).
- C++11 adds syntactic sugar for building function objects: lambdas
- Lambdas will frequently change the way you write code to make it more elegant and faster.
- Lambdas make the existing STL algorithms roughly 100x more usable.
- Newer C++ libraries increasingly are designed assuming lambdas as available, some even require you to write lambdas to use the library at all



# What's a lambda / function object?

- **C++ version of closure**
- **function + “environment”**
  - collection of variables in shared scope
- **Very similar to a class object**
  - Easier to write
  - Often smaller



## C++98 function objects

```
int outer_scope_int = 42;
```

```
class GreaterThan42 {
 int i = outer_scope_int;
 bool operator()(int x){ return x>i; }
}
```

```
std::cout << GreaterThan42(7);
```



## C++11 Lambdas

```
int i = 42;

auto GreaterThan42 =
 [=](int x){ return x>i; };

std::cout << GreaterThan42(7);
```



# Lambda breakdown

- `[](){} //Basic Lambda`
- `[] //Capture list`
  - How to store variables from the outer scope? Store by value or reference?
  - `[=]` Store everything by value
  - `[&]` Store everything by reference
  - `[&x, =y]` Store x by reference and y by value





# Lambda breakdown

- `[](){} //Basic Lambda`
- `() // Parameter list`
  - Just a function parameter list
- `{ } // Lambda Body`
  - Just a function body
  - Can have multiple statements



Standard Library: <algorithm>



# std::algorithms

- <https://en.cppreference.com/w/cpp/algorithm>
- The algorithms library defines functions for a variety of purposes that operate on ranges of elements.
  - searching
  - sorting
  - counting
  - manipulating
- Note that a range is defined as [first, last) over an arbitrary container.
- Pre-date C++11. Defined for functors.
- Lambdas make them easier to use.



# Lambdas and std::algorithms

```
// C++98
```

```
vector<int>::iterator i = v.begin();
// because we need to use i later
```

```
for(; i != v.end(); ++i) {
 if(*i > x && *i < y) break;
}
```

```
// C++11
```

```
auto i = find_if(begin(v), end(v),
 [=](int i){ return i>x && i<y; }
);
```



## Some std::algorithms

- **find\_if**: iterator to element which satisfies predicate function
- **for\_each**: apply functor to every element
- **stable\_sort**: sorts while preserving order between equal elements
- **binary\_search**: determines if an element exists in a certain range



## More std::algorithms

- <https://en.cppreference.com/w/cpp/algorithm>
- <https://www.fluentcpp.com/2017/01/05/the-importance-of-knowing-stl-algorithms/>
- <https://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning>
- <https://www.fluentcpp.com/2018/07/10/105-stl-algorithms-in-less-than-an-hour/>



# World map of C++ STL Algorithms



Standard Library: <filesystem>





# <filesystem>

- **Provides facilities for performing operations on file systems and their components**
  - paths
  - regular files
  - file\_status and file\_type
  - directories



## <filesystem>

```
path f = "dir/hypothetical.cpp"; // naming a
file

assert(std::exists(f)); // f must exist?

if (std::is_regular_file(f)){ // f is file?
 std::cout << f
 << " is a file; its size is "
 << file_size(f)
 << '\n';
}
```



# Standard Library <regex>



## <regex>

- The regular expressions library provides a class that represents regular expressions.
- C++ supports multiple variants of regex.
- The regular expression syntax and semantics are designed so that regular expressions can be compiled into state machines for efficient execution.
- The regex type performs this compilation at run time.



## <regex>

```
bool is_valid_ZIP(const string& s){
 regex pat {
 R"(\w{2}\s*\d{5}(-\d{4})?)"
 };

 return regex_match(s, pat);
}
```



# Standard Library <thread>



## <thread>

- **C++11 now includes built-in support for:**
  - threads
  - mutual exclusion
  - condition variables
  - futures.
- **C++20 is working on improving this with higher level abstractions**
- **Some <algorithms> are already parallelized**



## <thread>

```
auto p = make_shared<T>();
auto thr = [](std::shared_ptr<T> p){...};

std::thread t1(thr, p), t2(thr, p), t3(thr, p);

p.reset(); // release ownership

t1.join(); t2.join(); t3.join();

std::cout << "All threads completed, ";
 << "the last one deleted the T";
```





# The Future of C++: C++20



# Concepts

- **Template types are currently defined for all types**
- **Concepts limit the scope of permissible types**
  - Sequence Types
  - Arithmetic Types
  - Iterator Types ...



# Concepts

- **template<typename T>**
  - \forall T such that T is a type
- **template<Sequence S, Number N>**
  - \forall S, N such that S is a Sequence and N is a Number



# Concepts

```
//C++17
template<typename Seq, typename Num>
Num sum(Seq s, Num v){
 for (const auto& x : s)
 v+=x;
 return v;
}
```

```
//C++20
template<Sequence Seq, Number Num>
Num sum(Seq s, Num v){
 for (const auto& x : s)
 v+=x;
 return v;
}
```



# Modules

- `#include "header.h"` is old and error-prone
- `#include` is expensive. Must re-parse headers each time they're included even if the code is already compiled and understood.
- The Modules extension add an `import` keyword which will import a module.



# Modules

- A module library is compiled once, **import**ed everywhere.
- Two modules can be imported in either order without changing their meaning.
- If you import something into a module, users of your module do not implicitly gain access to (and are not bothered by) what you imported: **import** is not transitive.
- Improves maintainability and compile-time performance.



# Define a Module

```
// file Vector.cpp:
module; // this compilation will define a module

export module Vector; // defining the module
export class Vector {
 // Vector Declaration
};
Vector::Vector(int s):elem{new double[s]}, sz{s}{}
double& Vector::operator[](int i){return elem[i];}
int Vector::size(){return sz;}
export int size(const Vector& v) {return v.size();}
```

- This defines a module called Vector, exports the class Vector, all its member functions, and the non-member function size().



# Use a Module

```
// file user.cpp:
import Vector; // get Vector's interface
#include <cmath> // get the standard math lib
double sqrt_sum(Vector& v){
 double sum = 0;
 for (int i=0; i!=v.size(); ++i)
 sum+=std::sqrt(v[i]);
 return sum;
}
```





# Ranges

- Current `<algorithm>`s are defined on half open ranges using `begin` and `end` iterators.
- The Range proposal accepted into C++20 would add a `range` type to the standard.
- Rewrite `<algorithm>`s using just the range of a container.
- No more iterators `begin(v)`, `end(v)`

