The road so far

- **Basics**: Syntax, variables, control structures.

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.
- **Compilation**: Libraries, binaries, CMake, Doxygen, Sphinx.

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.
- **Compilation**: Libraries, binaries, CMake, Doxygen, Sphinx.
- **Pointers**: Memory, references, heap and stack.

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.
- **Compilation**: Libraries, binaries, CMake, Doxygen, Sphinx.
- **Pointers**: Memory, references, heap and stack.
- **Classes**: Members, constructors, destructors, visibility.

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.
- **Compilation**: Libraries, binaries, CMake, Doxygen, Sphinx.
- **Pointers**: Memory, references, heap and stack.
- **Classes**: Members, constructors, destructors, visibility.
- **Inheritance**: Derived classes, virtual functions.

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.
- **Compilation**: Libraries, binaries, CMake, Doxygen, Sphinx.
- **Pointers**: Memory, references, heap and stack.
- **Classes**: Members, constructors, destructors, visibility.
- **Inheritance**: Derived classes, virtual functions.
- **Polymorphism**: Dynamic binding, abstract classes.

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.
- **Compilation**: Libraries, binaries, CMake, Doxygen, Sphinx.
- **Pointers**: Memory, references, heap and stack.
- **Classes**: Members, constructors, destructors, visibility.
- **Inheritance**: Derived classes, virtual functions.
- **Polymorphism**: Dynamic binding, abstract classes.
- **Smart pointers**: $\text{shared}_{\text{ptr}}$, $\text{unique}_{\text{ptr}}$, debugging

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.
- **Compilation**: Libraries, binaries, CMake, Doxygen, Sphinx.
- **Pointers**: Memory, references, heap and stack.
- **Classes**: Members, constructors, destructors, visibility.
- **Inheritance**: Derived classes, virtual functions.
- **Polymorphism**: Dynamic binding, abstract classes.
- **Smart pointers**: $\text{shared}_{\text{ptr}}$, $\text{unique}_{\text{ptr}}$, debugging
- **I/O**: istream/ostream, cin/cout, fstream, sstream.

The road so far

- **Basics**: Syntax, variables, control structures.
- **Functions**: Declaration/definition, overload, return values.
- **Compilation**: Libraries, binaries, CMake, Doxygen, Sphinx.
- **Pointers**: Memory, references, heap and stack.
- **Classes**: Members, constructors, destructors, visibility.
- **Inheritance**: Derived classes, virtual functions.
- **Polymorphism**: Dynamic binding, abstract classes.
- **Smart pointers**: $\text{shared}_{\text{ptr}}$, $\text{unique}_{\text{ptr}}$, debugging
- **I/O**: istream/ostream, cin/cout, fstream, sstream.
- **Templates**: Generic programming, type deduction. lambdas.

The road so far

- **STL I**: Containers. `std::vector`, `std::list`, `std::map`, `std::set`...

The road so far

- **STL I**: Containers. `std::vector`, `std::list`, `std::map`, `std::set`...
- **STL II**: Algorithms. `std::sort`, `std::find`, `std::count_if`...

The road so far

- **STL I**: Containers. `std::vector`, `std::list`, `std::map`, `std::set`...
- **STL II**: Algorithms. `std::sort`, `std::find`, `std::count_if`...
- **Error handling**: `try`, `catch`, `throw`.

The road so far

- **STL I**: Containers. `std::vector`, `std::list`, `std::map`, `std::set`...
- **STL II**: Algorithms. `std::sort`, `std::find`, `std::count_if`...
- **Error handling**: `try`, `catch`, `throw`.
- **Technicalities**: `enum class`, `std::any`, `std::variant`, `std::optional`.

The road so far

- **Efficiency**: Cache, SIMD vectorization, branch prediction. `-O3 -march=native`.

The road so far

- **Efficiency**: Cache, SIMD vectorization, branch prediction. `-O3 -march=native`.
- **Concurrency**: `std::thread`, `std::mutex`, `std::async`.

The road so far

- **Efficiency**: Cache, SIMD vectorization, branch prediction. `-O3 -march=native`.
- **Concurrency**: `std::thread`, `std::mutex`, `std::async`.
- **Parallelism**: `std::execution::sequential`, `std::execution::par`, `std::execution::par_unseq`, tbb, thrust.

What was left out

- Concepts: Template type constraints.

What was left out

- Concepts: Template type constraints.
- Coroutines: Asynchronous programming.

What was left out

- Concepts: Template type constraints.
- Coroutines: Asynchronous programming.
- Ranges: STL-like algorithms.

What was left out

- Concepts: Template type constraints.
- Coroutines: Asynchronous programming.
- Ranges: STL-like algorithms.
- Modules: Compilation units.

What was left out

- Concepts: Template type constraints.
- Coroutines: Asynchronous programming.
- Ranges: STL-like algorithms.
- Modules: Compilation units.
- Networking

What was left out

- Concepts: Template type constraints.
- Coroutines: Asynchronous programming.
- Ranges: STL-like algorithms.
- Modules: Compilation units.
- Networking
- Graphics

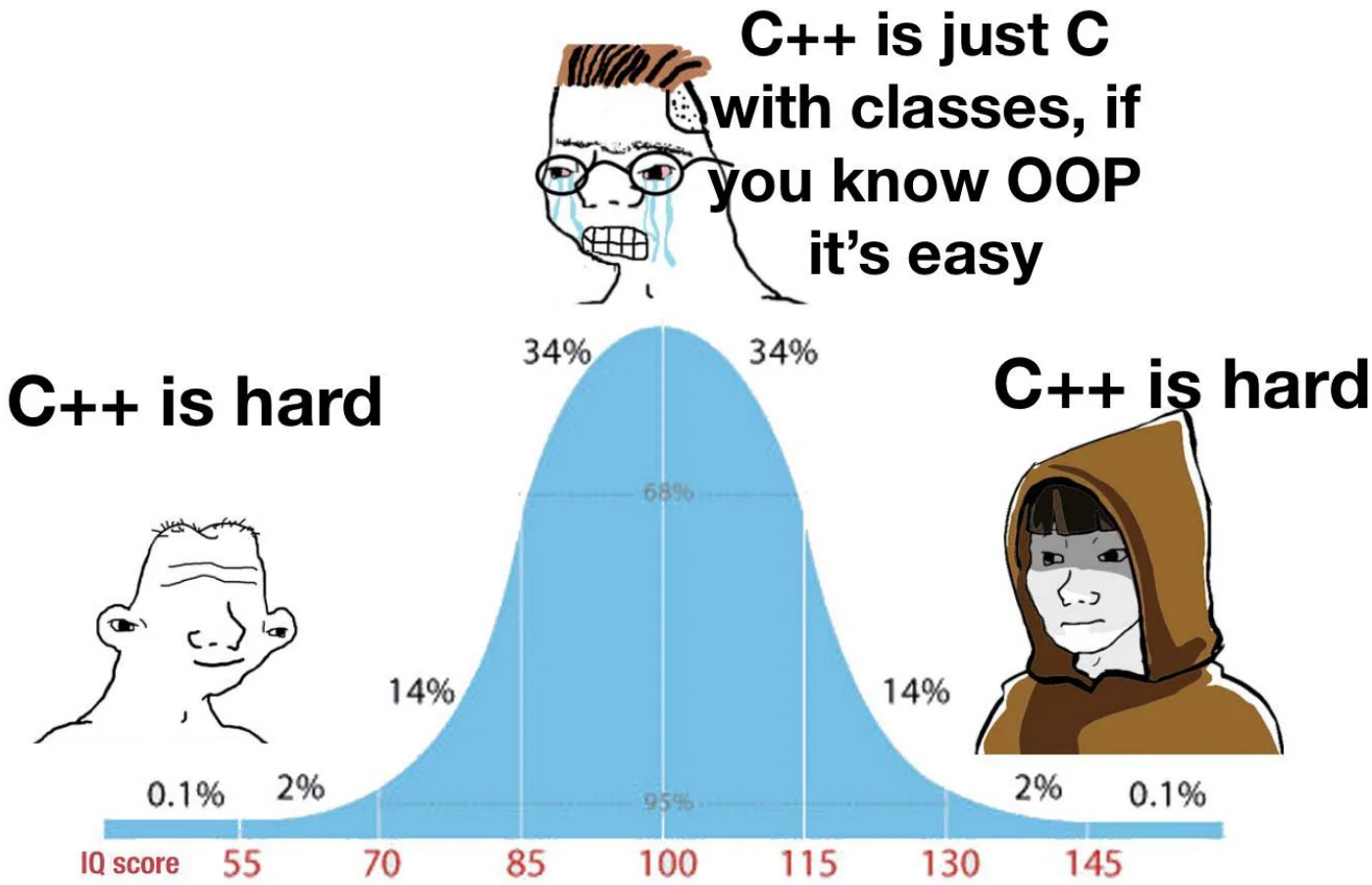
What was left out

- Concepts: Template type constraints.
- Coroutines: Asynchronous programming.
- Ranges: STL-like algorithms.
- Modules: Compilation units.
- Networking
- Graphics
- Suffixes: `auto time = 10s;`

What was left out

- Concepts: Template type constraints.
- Coroutines: Asynchronous programming.
- Ranges: STL-like algorithms.
- Modules: Compilation units.
- Networking
- Graphics
- Suffixes: `auto time = 10s;`
- C++23 and beyond: Contracts, reflection, generators, mdspan, views.

u rn hopefully



Next steps

- Fail forwards: Screw up and learn from it.

Next steps

- Fail forwards: Screw up and learn from it.
- Explore new paradigms: Functional, test-driven, event-driven...

Next steps

- Fail forwards: Screw up and learn from it.
- Explore new paradigms: Functional, test-driven, event-driven...
- Think as high-level as possible.

Next steps

- Fail forwards: Screw up and learn from it.
- Explore new paradigms: Functional, test-driven, event-driven...
- Think as high-level as possible.
- Be allergic to C-like interfaces when writing C++.

Lesson 2: Basic

- variables
- scope
- conditionals (`if`, `switch`)
- loops (`for`, ranged-for, `while`, `do{} while()`)

Lesson 3: Basic II

- Functions (`auto foo(int a, float b=0.0f){return "Hi";}`
- `auto`, `static`
- overloading
- `using` (aliases)
- `namespace`

Lesson 4: Compilation

- Preprocessing, compile and link
- Libraries vs executables
- g++, clang++

Lesson 5: CMake

- CMake (compilation scripts)
- Doxygen (documentation from comments)
- Sphinx (Webpage for docs)
- Breathe (Links sphinx and Doxygen)
- GTest (Unit testing), Test-Driven-Develpment

Lesson 6: Memory

- Pointers, references.
- Passing by reference/value `void foo(int& x)` `void foo2(int x);`
- `const`, `constexpr`
- Casting (`static_cast`, `reinterpret_cast`).
- Stack (`std::array`) vs heap (`std::vector`).

Lesson 8: OOP I

- `class`, `struct`
- `std::tuple`
- methods, members, visibility
- `this`
- nested classes
- `const`, `static`, `friend`

Lesson 9: OOP II

- Constructors, Destructors, `operator`
- RAII
- rule of zero and rule of all

Lesson 10: OOP III

- Inheritance `class A: public B`
- Virtual functions, `virtual`, `override`
- Abstract classes `virtual void foo() = 0;`
- Polymorphism `dynamic_cast`
- User conversions `operator int(){ return some_int;}`

Lesson 12: Advanced memory

- Debugging with gdb/lldb
- Smart pointers, `shared_ptr`, `unique_ptr`
- Smart pointer deleters

Lesson 13: Files and streams

- `istream`, `ostream`
- `fstream`, `stringstream`
- overloading >> and <<
- Formating, `std::format`, `std::hex`, etc

Lesson 14: Templates I

- Function templates
  `template<typename T> T max(T a, T b)`
- Deduction, specialization
- Non-type parameters, `template<int value>`
- Aliases
  `template<typename T> using MyVec = std::vector<T>`
- Variadic templates, `template<typename... Args>`

Lesson 15: Templates II

- Class templates
  `template<typename T> class A{};`
- Deduction, specialization
- Function objects `auto operator()(const T& a, const T& b)`
- Lambdas `auto foo = [](auto a){return 2*a;}`

Lesson 18: STL I

- Containers: Data structures
  `std::vector`, `std::list`, `std::map`, `std::set`
  `std::multimap`, `std::multiset`

```
std::vector<int> vi{1,2,3};
vi[2] = 10;
std::list<int> li{1,2,3};
//li[2] = 10; // Error
li.push_back(4);
std::map<string, int> m{{"one", 1}};
m["two"] = 2;
m["three"] = 3;
std::set<int> s{1,1,2,3}; // 1,2,3
```

Lesson 19: STL II

- Iterators: Pointer generalization
  `begin()`, `end()`, `rbegin()`, `rend()`

```
template <typename T>
T& std::vector<T>::operator[](uint index) {
  return *(begin() + index);
}
```

Lesson 19: STL II

- Algorithms: Functions that operate on containers
  `std::sort`, `std::find`, `std::count_if`

```
std::algorithm_name(begin_iterator,
                    end_iterator,
                    other_parameters);
```

Lesson 20: Error handling

- Exceptions: `try`, `catch`, `throw`

```
try{
  //... something that might fail
  if(error) throw std::runtime_error("Error");
 } catch(std::exception& e){
  // Handle exception or
  //throw; // Rethrow
  // or std::terminate();
 }
```

- Error classes: `std::exception`, `std::runtime_error`, `std::logic_error`

Lesson 20: Error handling

- "Define errors out of existence"

```
void foo(int x){
  if(x < 0)
    throw std::runtime_error("Negative value");
  //do something
}
//...
try{
  foo(-1);
 } catch(std::exception& e){
  std::cerr << e.what();
 }
```

Lesson 20: Error handling

- "Define errors out of existence"

```
void foo(unsigned int x){
  //do something, no need to check
}
```

Lesson 21: Technicalities

```
enum class Color {red, green, blue};
std::optional<Color> foo(std::any a = "hello"){
  std::variant<Color, float> v = 1.0f;
  v = Color::blue;
  if(a.type() != typeid(int))
    return std::nullopt;
  return std::get<Color>(v);
}
```

Lesson 21: Technicalities

```
int main(){
  cout<<"This is NOT printed\n";
  throw "Error";
  // or std::terminate();
}
```

```
int main(){
  cout<<"This is printed"<<endl;
  throw "Error";
}
```

Lesson 22: Efficiency

- Cache: `-O3 -march=native`, `vector<vector<int>>` vs `vector<int>`.
- SIMD
- Branch prediction: `if(x) foo(); else bar();`

Lesson 22: Efficiency

```
using namespace std;
template<class T>
void foo(int nx, int ny){
  vector<T> v1(nx*ny, 0);
  vector<vector<T>> v2(nx, vector<T>(ny, 0));
  T* v3 = new T[nx*ny];
  T** v4 = (T**)malloc(ny*sizeof(T*));
  for(int i = 0; i < ny; ++i)
    v4[i] = (T*)malloc(nx*sizeof(T));
  //...
  delete[] v3;
  for(int i = 0; i < ny; ++i)
    free(v4[i]);
  free(v4);
}
```

- Cache: Increase spatio-temporal data locality.

Lesson 22: Efficiency

```
using namespace std;
void foo(){
  int n = 1000000;
  vector v1(n, 1.0f);
  vector v2(n, 2.0f);
  vector v3(n, 0.0f);
  for(int i = 0; i < n; ++i)
    v3[i] = 2.0*v1[i] + v2[i];
}
```

- SIMD: Group similar operations together.

Lesson 22: Efficiency

```
void foo(){
  auto v = randomVector(1000000);
  float result = 0;
  //sort(v.begin(), v.end());
  for(auto vi: v)
    if(vi > 0) result += vi;
    // result += vi > 0 ? vi : 0;
}
```

- Branch prediction: Use regular patterns.
- Avoid branches if possible

Lesson 23: Concurrency

```
void foo(){
  mutex m;
  auto print = [&m](auto s){
    scoped_lock l{m};
    cout<<s;
  };
  thread t1(print("World"));
  thread t2(print("Hello"));
  t1.join();
  t2.join();
}
```

- thread, atomic, async, future

Lesson 23: Concurrency

```
void foo(){
  atomic c{0};
  auto print = [&](auto a){
    c++;
    return a+c;
  };
  future f1 = async(print, 100);
  future f2 = async(print, -1000);
  cout<<f1.get()<<" "<<f2.get();
}
```

- thread, atomic, async, future