The Best Parts of C++

About my Talks

- Move to the front!
- Please interrupt and ask questions
- This is approximately how my training days look

The Best Parts of C++

We will:

1. Cover ~25 different major C++ features
2. Discuss the advantages of each feature
3. Mention who was involved in the design of this feature.

The information for who was involved is based on information I was able to collect from the standards papers.

This is not a conclusive list of the people involved, it's just the information I had available.

Some people are repeated because they edit proposals or help with wording regularly.

The Best Parts of C++

This talk was inspired by

- Walter Brown's lightning talk from CppCon 2018 "Thank You (I'm sorry it took me so long to say it.)" thanking all of the mentors, teachers and colleagues over the years
- Sean Parent's keynote from Pacific++ 2018 "Generic Programming" about the collaboration of the people involved in the design of the STL

What C++ standard do you use?

- C++98
- C++03
- C++11
- C++14
- C++17
- C++20

#1: The C++ Standard

#1: The C++ Standard

C++ has a standard. It is an internationally agreed to document that specifies the behavior of our compilers and programs.

The value of this should not be underestimated. We can have competing implementations that sort out the *exact* behavior of our programs. Few other languages have this.

- Standard: C++98+
- Credit: Many, many people

Let’s start with this simple code from a beginning C++ magazine I found:

```
1  #include <iostream>
2
3  double pi = 3.141593;
4
5  int main()
6  {
7    double area, radius = 1.5;
8    area = pi * radius * radius;
9    std::cout << area;
10 }
```

Is it possible for `pi` to change?

If only there was some way to specify that the value for `pi` cannot change…

```
1  #include <iostream>
2
3  double pi = 3.141593;
4
5  int main()
6  {
7    double area, radius = 1.5;
8    area = pi * radius * radius;
9    std::cout << area;
10 }
```

`const` tells the compiler that a value cannot change.

```
1  #include <iostream>
2
3  const double pi = 3.141593; ///
4
5  int main()
6  {
7    double area, radius = 1.5;
8    area = pi * radius * radius;
9    std::cout << area;
10 }
```

If we apply `const` consistently we simplify our variable declarations

```
1  #include <iostream>
2
3  const double pi = 3.141593;
4
5  int main()
6  {
7    const double radius = 1.5;
8    const double area = pi * radius * radius;
9    std::cout << area;
10 }
```

#2: `const`

#2: `const`

One of the most important tools to clean code. An object declared `const` or accessed via a `const` reference or `const` pointer cannot be modified. The compiler enforces this.

```
1  const int i = 5; /// must be initialized, cannot change
2  int const j = 6; // or this way around
```

More importantly: it forces us to initialize the value.

- Standard: C++98
- Credit: Stroustrup

We started with a double, let's make a collection of them.

```
1  double *get_data() {
2    double *data = new double[3];
3
4    data[0] = 1.1;
5    data[1] = 2.2;
6    data[2] = 3.3;
7
8    return data;
9  }
```

And to use the data:

```
1  double *get_data() {
2    double *data = new double[3];
3    data[0] = 1.1; data[1] = 2.2; data[2] = 3.3;
4    return data;
5  }
6
7  double sum_data(double *data) {
8    return data[0] + data[1] + data[2];
9  }
10
11 int main() {
12   return sum_data(get_data()); ///
13 }
```

Look good?

Ooops

```
1  double *get_data() {
2    double *data = new double[3];
3    data[0] = 1.1; data[1] = 2.2; data[2] = 3.3;
4    return data;
5  }
6
7  double sum_data(double *data) {
8    return data[0] + data[1] + data[2]; /// uncaught leak
9  }
10
11 int main() {
12   return sum_data(get_data());
13 }
```

Ooops

```
1  double *get_data() {
2    double *data = new double[3];
3    data[0] = 1.1; data[1] = 2.2; data[2] = 3.3;
4    return data;
5  }
6
7  double sum_data(double *data) {
8    const double sum = data[0] + data[1] + data[2];
9
10   delete [] data; /// fixed?
11
12   return sum;
13 }
14
15 int main() {
16   return sum_data(get_data());
17 }
```

If only there was some way to automatically delete things when they are no longer needed…

```
1  struct Double_Data {
2    Double_Data(const std::size_t size)
3      : data(new double[size]) /// allocate
4    {
5    }
6
7    ~Double_Data() { /// destructor
8      delete [] data; // free
9    }
10
11   double *data;
12 };
```

Now we get a cleaner way to use this:

```
1  struct Double_Data {
2    Double_Data(const std::size_t size) : data(new double[size]) { }
3    ~Double_Data() { delete [] data; }
4    double *data;
5  };
6
7  get_data() {
8    Double_Data data(3);
9    data.data[0] = 1.1; data.data[1] = 2.2; data.data[2] = 3.3;
10   return data;
11 }
12
13 double sum_data(const Double_Data &d) {
14   return d.data[0] + d.data[1] + d.data[2];
15 }
16
17 int main() {
18   return sum_data(get_data()); /// no leak, but we'll come back to this
19 }
```

#3: Deterministic Object Lifetime and Destruction

#3: Deterministic Object Lifetime and Destruction

Constructor / destructor pairs (RAII) combined with scoped values give us determinism that removes the need for things like `finally`.

```
1  void some_func()
2  {
3    std::string s{"Hello there world"}; // allocate a string
4  } // automatically free it when scope exits
```

- Standard: C++98
- Credit: Bjarne Stroustrup?

We have a handy data holder that holds `double`s.

```
1  struct Double_Data {
2    Double_Data(const std::size_t size) : data(new double[size]) { }
3    ~Double_Data() { delete [] data; }
4    double *data;
5  };
```

Now one for `int`

```
1  struct Int_Data {
2    Int_Data(const std::size_t size) : data(new int[size]) { }
3    ~Int_Data() { delete [] data; }
4    int *data;
5  };
```

And maybe `float`

```
1  struct Float_Data {
2    Float_Data(const std::size_t size) : data(new float[size]) { }
3    ~Float_Data() { delete [] data; }
4    float *data;
5  };
```

If only there was some way to avoid repeating ourselves with this…

```
1  struct Double_Data {
2    Double_Data(const std::size_t size) : data(new double[size]) { }
3    ~Double_Data() { delete [] data; }
4    double *data;
5  };
6
7  struct Int_Data {
8    Int_Data(const std::size_t size) : data(new int[size]) { }
9    ~Int_Data() { delete [] data; }
10   int *data;
11 };
12
13 struct Float_Data {
14   Float_Data(const std::size_t size) : data(new float[size]) { }
15   ~Float_Data() { delete [] data; }
16   float *data;
17 };
```

templates!

```
1  /// declare a class template that can hold anything we want
2  template<typename Value_Type>
3  struct Data {
4    Data(const std::size_t size)
5      : data(new Value_Type[size])
6    {
7    }
8
9    ~Data() { delete [] data; }
10
11   Value_Type *data;
12 };
13
14 /// declare a function template that takes 3 params of the same type
15 /// and passes that type on to the `Data` template
16 template<typename Value_Type>
17 Data<Value_Type> get_data(const Value_Type &v1, const Value_Type &v2,
18                           const Value_type &v3)
19 {
20   Data<Value_Type> d(3);
21   d.data[0] = v1; d.data[1] = v2; d.data[2] = v3;
22   return d;
23 }
```

#4: Templates

#4: Templates

The ultimate in the DRY principle. You can write a `template` that has types and values filled in at compile-time.

- `template` types are generated by the compiler at compile time
- Do not need any kind of type-erasure (like Java generics do)
- Highly efficient runtime code possible, as good as (or better than) hand writing the various options
- `template` system is Turing complete (not necessarily a good thing)

```
1  template<typename SomeType>
2  struct S { /// struct can do anything it wants with this type
3  };
```

- Standard: C++98
- Credit: Ada Generics?

Does this code bother anyone?

```
1  template<typename Value_Type>
2  struct Data {
3    Data(const std::size_t size) : data(new Value_Type[size]) { }
4    ~Data() { delete [] data; }
5    Value_Type *data;
6  };
7
8  template<typename Value_Type>
9  Data<Value_Type> get_data(const Value_Type &v1, const Value_Type &v2,
10                           const Value_type &v3)
11 {
12   Data<Value_Type> d(3);
13   d.data[0] = v1; d.data[1] = v2; d.data[2] = v3;
14   return d;
15 }
```

Does this code bother anyone?

```
1  template<typename Value_Type>
2  struct Data {
3    Data(const std::size_t size) : data(new Value_Type[size]) { }
4    ~Data() { delete [] data; }
5    Value_Type *data;
6  };
7
8  template<typename Value_Type>
9  Data<Value_Type> get_data(const Value_Type &v1, const Value_Type &v2,
10                           const Value_type &v3)
11 {
12   Data<Value_Type> d(3);
13   d.data[0] = v1; d.data[1] = v2; d.data[2] = v3;
14   return d; ///
15 }
```

It only accidentally works because of the copy elision that compilers have “always” implemented.

If only there was some way to contain a set of values that has already taken care of these issues…

`std::vector` of course.

```
1  #include <vector>
2
3  template<typename Value_Type>
4  std::vector<Value_Type> get_data(const Value_Type &v1, const Value_Type &v2,
5                                   const Value_type &v3)
6  {
7    std::vector<Value_Type> data;
8    data.push_back(v1);
9    data.push_back(v2);
10   data.push_back(v3);
11   return data;
12 }
```

And what about the sum code?

If only there was some utility to automatically add these values up…

```
1  double sum_data(const Double_Data &d) {
2    return d.data[0] + d.data[1] + d.data[2];
3  }
```

```
#include <numeric>

template<typename T>
T sum_data(const std::vector<T> &d) {
  return std::accumulate(d.begin(), d.end(), T());
}
```

#5: Algorithms and Standard Template Library

#5: Algorithms and Standard Template Library

- `set<>`
- `vector<>`
- `for_each`
- `any_of`
- etc

A generic set of composable tools.

- Standard: C++98
- Credit: Alexander Stepanov, Meng Lee et al (see Sean's Talk)

Now we are here:

```
#include <numeric>
#include <vector>

template<typename Value_Type>
std::vector<Value_Type> get_data(const Value_Type &v1, const Value_Type &v2,
                                 const Value_Type &v3)
{
  std::vector<Value_Type> data;
  data.push_back(v1);
  data.push_back(v2);
  data.push_back(v3);
  return data;
}

template<typename T>
T sum_data(const std::vector<T> &d) {
  return std::accumulate(d.begin(), d.end(), T());
}

int main() {
  return sum_data(get_data(1,2,3));
}
```

More generic, safer, no potential memory issues, but still C++98.

But we know the amount of data at compile-time. If only there was some fixed-size container available…

```
#include <numeric>
#include <array>

template<typename Value_Type>
std::array<Value_Type, 3> get_data(const Value_Type &v1, const Value_Type &v2,
                                   const Value_Type &v3)
{
  std::array<Value_Type, 3> data;
  data[0] = v1;
  data[1] = v2;
  data[2] = v3;
  return data;
}
```

No dynamic allocation, win-win scenario with knowing the size of the data structure at compile time. (Lenny Maiorani's negative cost abstraction).

#6: `std::array`

#6: `std::array`

A fixed-size stack-based container. Having the size part of the type information gives more optimization opportunities.

```
1 | std::array<Type, Size> data
```

- Standard: C++11
- Credit: Based on `boost::array`, Nicolai Josuttis

I’d really like a version that takes 2 parameters

```
1  template<typename VT>
2  std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
3  {
4    std::array<VT, 3> data;
5    data[0] = v1; data[1] = v2; data[2] = v3;
6    return data;
7  }
```

And maybe 1 that takes 1

```
template<typename VT>
std::array<VT, 2> get_data(const VT &v1, const VT &v2)
{
  std::array<VT, 2> data;
  data[0] = v1; data[1] = v2;
  return data;
}
template<typename VT>
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
  std::array<VT, 3> data;
  data[0] = v1; data[1] = v2; data[2] = v3;
  return data;
}
```

Simply add the 1 case in…

```
template<typename VT>
std::array<VT, 2> get_data(const VT &v1)
{
  std::array<VT, 2> data;
  data[0] = v1;
  return data;
}
template<typename VT>
std::array<VT, 2> get_data(const VT &v1, const VT &v2)
{
  std::array<VT, 2> data;
  data[0] = v1; data[1] = v2;
  return data;
}
template<typename VT>
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
  std::array<VT, 3> data;
  data[0] = v1; data[1] = v2; data[2] = v3;
  return data;
}
```

Ooops.

```
1  template<typename VT>
2  std::array<VT, 2> get_data(const VT &v1)
3  {
4    std::array<VT, 2> data;  /// oops copy paste error
5    data[0] = v1;
6    return data;
7  }
8  template<typename VT>
9  std::array<VT, 2> get_data(const VT &v1, const VT &v2)
10 {
11   std::array<VT, 2> data;
12   data[0] = v1; data[1] = v2;
13   return data;
14 }
15 template<typename VT>
16 std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
17 {
18   std::array<VT, 3> data;
19   data[0] = v1; data[1] = v2; data[2] = v3;
20   return data;
21 }
```

If only there was a way to initialize the array values in one step…

```
template<typename VT>
std::array<VT, 1> get_data(const VT &v1)
{
  std::array<VT, 1> data;
  data[0] = v1;
  return data;
}
template<typename VT>
std::array<VT, 2> get_data(const VT &v1, const VT &v2)
{
  std::array<VT, 2> data;
  data[0] = v1; data[1] = v2;
  return data;
}
template<typename VT>
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
  std::array<VT, 3> data;
  data[0] = v1; data[1] = v2; data[2] = v3;
  return data;
}
```

List initialization

```
1  template<typename VT>
2  std::array<VT, 1> get_data(const VT &v1)
3  {
4    std::array<VT, 1> data{v1}; /// list initialization
5    return data;
6  }
7  template<typename VT>
8  std::array<VT, 2> get_data(const VT &v1, const VT &v2)
9  {
10   std::array<VT, 2> data{v1, v2};
11   return data;
12 }
13 template<typename VT>
14 std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
15 {
16   std::array<VT, 3> data{v1, v2, v3};
17   return data;
18 }
```

Which becomes:

```
template<typename VT>
std::array<VT, 1> get_data(const VT &v1)
{
  return {v1};
}
template<typename VT>
std::array<VT, 2> get_data(const VT &v1, const VT &v2)
{
  return {v1, v2};
}
template<typename VT>
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
  return {v1, v2, v3};
}
```

#7: List Initialization

#7: List Initialization

C++ initialization comes in many, many, many different flavors. But there's no denying that list-initialization (and in this case direct initialization) has changed the way we use containers.

```
1 | std::vector<int> vec{1,2,3,4};
```

- C++ Standard: C++11
- Credit: Jason Merrill, Daveed Vandevoorde, J. Stephen Adamczyk, Gabriel Dos Reis, and Bjarne Stroustrup

Now to add a 4,5,6 parameter version…

```
1  template<typename VT>
2  std::array<VT, 1> get_data(const VT &v1)
3  {
4    return {v1};
5  }
6  template<typename VT>
7  std::array<VT, 2> get_data(const VT &v1, const VT &v2)
8  {
9    return {v1, v2};
10 }
11 template<typename VT>
12 std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
13 {
14   return {v1, v2, v3};
15 }
```

If only there was a way to avoid all this code duplication…

```
template<typename VT>
std::array<VT, 1> get_data(const VT &v1)
{
  return {v1};
}
template<typename VT>
std::array<VT, 2> get_data(const VT &v1, const VT &v2)
{
  return {v1, v2};
}
template<typename VT>
std::array<VT, 3> get_data(const VT &v1, const VT &v2, const VT &v3)
{
  return {v1, v2, v3};
}
template<typename VT>
std::array<VT, 4> get_data(const VT &v1, const VT &v2, const VT &v3, const VT &v4)
{
  return {v1, v2, v3, v4};
}
template<typename VT>
std::array<VT, 5> get_data(const VT &v1, const VT &v2, const VT &v3, const VT &v4,
const VT &v5)
{
  return {v1, v2, v3, v4, v5};
}
template<typename VT>
std::array<VT, 6> get_data(const VT &v1, const VT &v2, const VT &v3, const VT &v4,
const VT &v5, const VT &v6)
```

````
{
  return {v1, v2, v3, v4, v5, v6};
}```

----


Variadic Templates:

```cpp
/// require at least one parameter and it sets the type
template<typename VT, typename ... Params>
std::array<VT, sizeof...(Params)+1> get_data(const VT &v1, const Params& ...params)
{
  return {v1, params...};
}
````



````

# #8: Variadic Templates

# #8: Variadic Templates

Drastic simplification of code needing to match a variable number of parameters. Absolutely critical for maintainable implementations of things like `std::function`

```
1  template<typename ... P>  // variadic template
2  std::array<VT, sizeof...(P)> get_data(const P & ... params) // param expansion
3  {
4    return {params...}; // pack expansion
5  }
```

- C++ Standard: C++11
- Credit: Douglas Gregor, Jaakko Järvi, Jens Maurer, Jason Merrill, Eric Niebler

# Going back to our original example:

```
1  #include <iostream>
2
3  const double pi = 3.141593;
4
5  int main()
6  {
7    const double radius = 1.5;
8    const double area = pi * radius * radius;
9    std::cout << area;
10 }
```

Is `pi` known at compile time?

# If only there was some way to make a compile-time constant…

```
1  #include <iostream>
2
3  constexpr double pi = 3.141593;
4
5  int main()
6  {
7    const double radius = 1.5;
8    const double area = pi * radius * radius;
9    std::cout << area;
10 }
```

# Or even generate it at compile-time.

```
#include <iostream>

constexpr double calculate_pi() {
  return 22/7; // or something more clever
}

constexpr double pi = calculate_pi();

int main()
{
  const double radius = 1.5;
  const double area = pi * radius * radius;
  std::cout << area;
}
```

# #9: `constexpr`

# #9: `constexpr`

Compile-time generation of code and data.

```
1  // This function can be executed at compile-time
2  constexpr double calculate_pi() {
3    return 22/7;
4  }
5
6  // This value will be available at compile-time
7  constexpr double pi = calculate_pi();
```

- Standard: C++11, relaxed restrictions in C++14.
- Credit: Gabriel Dos Reis, Bjarne Stroustrup, Jens Maurer, Richard Smith, Nicolai Josuttis

What should the type of `pi` be?

1. `float`?
2. `double`?
3. `long double`?
4. I don’t care.
5. It depends

*I* generally either “Don’t Care” or think “It Depends.”

# If only there was some way to indicate that I don't care what the type is…

```
#include <iostream>

constexpr double calculate_pi() {
  return 22/7; // or something more clever
}

constexpr double pi = calculate_pi();

int main()
{
  const double radius = 1.5;
  const double area = pi * radius * radius;
  std::cout << area;
}
```

# auto

```
1  #include <iostream>
2
3  constexpr double calculate_pi() {
4    return 22/7; // or something more clever
5  }
6
7  constexpr auto pi = calculate_pi();
8
9  int main()
10 {
11   const auto radius = 1.5;
12   const auto area = pi * radius * radius;
13   std::cout << area;
14 }
```

# #10: `auto`

# #10: `auto`

Automatic deduction of value types.

- Standard: C++11
- Credit: Jaakko Järvi, Bjarne Stroustrup, Gabriel Dos Reis

*The `auto` feature has the distinction to be the earliest to be suggested and implemented: I had it working in my Cfront implementation in early 1984, but was forced to take it out because of C compatibility problems. Those compatibility problems disappeared when C++98 and C99 accepted the removal of "implicit `int`"; that is, both languages require every variable and function to be defined with an explicit type. The old meaning of auto ("this is a local variable") is now illegal. Several committee members trawled through millions of lines of code finding only a handful of uses – and most of those were in test suites or appeared to be bugs.*

Bjarne Stroustrup C++11 FAQ

# An `auto` restrospective

This is perfectly valid C++98. It means “the lifetime of the `int i` shall be automatically managed.” This is the default for all scoped variables. Hence the keyword being redundant.

```
1  int main() {
2    auto int i = 5;
3  }
```

Pre C-99, this meant “an automatically managed `int`” because `int` was the default type.

```
1  main() { // implicitly returns an int
2    auto i = 5;
3  }
```

Is there any way to adjust the previous code to say I *really* don't care about the types?

```
1  #include <iostream>
2
3  constexpr auto calculate_pi() { ///
4    return 22/7; // or something more clever
5  }
6
7  constexpr auto pi = calculate_pi();
8
9  int main()
10 {
11   const auto radius = 1.5;
12   const auto area = pi * radius * radius;
13   std::cout << area;
14 }
```

# #11: Return type deduction for normal functions.

# #11: Return type deduction for normal functions.

Not to be underestimated, this allows for code like:

```
auto get_thing() {
  struct Thing {};

  return Thing{};
}
```

Allows for the creation of powerful higher order function constructs, among other things.

- Standard: C++14
- Credit: Jason Merrill

Now I'd like to print a set of key, value pairs from a `map`. In C++98/03:

```
template<typename Map>
void print_map(const Map &map, const std::string &key_desc = "key",
                               const std::string &value_desc = "value")
{
  for (typename Map::const_iterator data_itr = map.begin();
       data_itr != map.end();
       ++data_itr)
  {
    std::cout << key_desc << ": '" << data_itr->first << "' "
              << value_desc << ": '" << data_itr->second << "'\n";
  }
}
```

We know `auto` could help, but we're in C++03. Of course, best practices (and Sean Parent) tell us…

No raw loops! So let's use an algorithm.

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string key_desc = "key",
3                                 const std::string value_desc = "value")
4  {
5    for_each(map.begin(), map.end(), /* something */);
6  }
```

What do we need?

- Something that is `Callable` (`operator()`)
- A way to capture the `key_desc` and `value_desc`

```
1  struct Map_Value_Printer {
2    Map_Value_Printer(const std::string &key_desc,
3                      const std::string &value_desc) /// Capture
4      : m_key_desc(key_desc), m_value_desc(value_desc)
5    { }
6
7    template<typename Value>
8    void operator()(const Value &value) { /// Callable
9      std::cout << m_key_desc << ": '" << value.first << "' "
10               << m_value_desc << ": '" << value.second << "'\n";
11   }
12
13   const std::string &m_key_desc;   /// captured values
14   const std::string &m_value_desc;
15 };
16
17 template<typename Map>
18 void print_map(const Map &map, const std::string key_desc = "key",
19                                const std::string value_desc = "value")
20 {
21   for_each(map.begin(), map.end(), Map_Value_Printer(key_desc, value_desc));
22 }
```

If only there was some handy way of creating a callable thing for use with an algorithm…

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string key_desc = "key",
3                                 const std::string value_desc = "value")
4  {
5    for_each(begin(map), end(map),
6             [&](const typename Map::value_type &data) { /// lambda!
7               std::cout << key_desc << ": '" << data.first << "' "
8                         << value_desc << ": '" << data.second << "'\n";
9             }
10   );
11 }
```

# #12: Lambdas

# #12: Lambdas

Lambdas allow us to create unnamed function objects which may or may not have captures. We are not allowed to know the name of the type of a lambda.

```
1 | auto lambda = [/*captures*/](int param1){ return param1 * 10; };
```

- Standard: C++11
- Credit: Jaakko Järvi, John Freeman, Lawrence Crowl, Peter Dimov, Daveed Vandevoorde

This is a bit wordy.

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string key_desc = "key",
3                                 const std::string value_desc = "value")
4  {
5    for_each(begin(map), end(map),
6             [&](const typename Map::value_type &data) { /// Here
7               std::cout << key_desc << ": '" << data.first << "' "
8                         << value_desc << ": '" << data.second << "'\n";
9             }
10   );
11 }
```

If only there was some way to automatically deduce the types of lambda parameters…

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string key_desc = "key",
3                                 const std::string value_desc = "value")
4  {
5    for_each(begin(map), end(map),
6             [&](const auto &data) { ///
7               std::cout << key_desc << ": '" << data.first << "' "
8                         << value_desc << ": '" << data.second << "'\n";
9             }
10   );
11 }
```

# #13: Generic And Variadic Lambdas

# #13: Generic And Variadic Lambdas

Create implicit templates by simply using the `auto` keyword.

```
1  auto lambda = [/*captures*/](auto ... params){
2    return std::vector<int>{params...};
3  };
```

- Standard: C++14
- Credit: Faisal Vali, Herb Sutter, Dave Abrahams with thanks to Jens Maurer, Douglas Gregor, Richard Smith, Christof Meerwald, John Spicer, Jason Merrill

Let’s go back to that pre-algorithm version for a minute:

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string &key_desc = "key",
3                                 const std::string &value_desc = "value")
4  {
5    for (typename Map::const_iterator data_itr = map.begin();
6         data_itr != map.end();
7         ++data_itr)
8    {
9      std::cout << key_desc << ": '" << data_itr->first << "' "
10               << value_desc << ": '" << data_itr->second << "'\n";
11   }
12 }
```

And simplify out the template for just a moment:

Sans template:

```
1  void print_map(const std::map<int, std::string> &data,
2                 const std::string &key_desc = "key",
3                 const std::string &value_desc = "value")
4  {
5    for (std::map<int, std::string>::const_iterator data_itr = map.begin();
6         data_itr != map.end();
7         ++data_itr)
8    {
9      std::cout << key_desc << ": '" << data_itr->first << "' "
10               << value_desc << ": '" << data_itr->second << "'\n";
11   }
12 }
```

This simple construct, iterating over a container in C++, was one of the most difficult things for me when learning C++.

# Applying `auto`:

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string &key_desc = "key",
3                                 const std::string &value_desc = "value")
4  {
5    for (auto data_itr = map.begin(); ///
6         data_itr != map.end();
7         ++data_itr)
8    {
9      std::cout << key_desc << ": '" << data_itr->first << "' "
10               << value_desc << ": '" << data_itr->second << "'\n";
11   }
12 }
```

If only there was some simple way to iterate over all the values in a container…

```
template<typename Map>
void print_map(const Map &map, const std::string &key_desc = "key",
                               const std::string &value_desc = "value")
{
  for (auto data_itr = map.begin();
       data_itr != map.end();
       ++data_itr)
  {
    std::cout << key_desc << ": '" << data.first << "' "
              << value_desc << ": '" << data.second << "'\n";
  }
}
```

# range-based `for` loops:

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string &key_desc = "key",
3                                 const std::string &value_desc = "value")
4  {
5    for (const auto &data : map) ///
6    {
7      std::cout << key_desc << ": '" << data.first << "' "
8                << value_desc << ": '" << data.second << "'\n";
9    }
10 }
```

# #14: range-based `for` loop

# #14: range-based `for` loop

Iterates over all elements in a container. Works with anything that has `begin()` and `end()` members/functions and C-style arrays.

```
1  for (const auto &value : container) {
2    // loops over each element in the container
3  }
```

- Standard: C++11
- Credit: Douglas Gregor, Beman Dawes

Now if only there was some way to make this `data.first`, `data.second` nonsense more readable…

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string &key_desc = "key",
3                                 const std::string &value_desc = "value")
4  {
5    for (const auto &data : map)
6    {
7      std::cout << key_desc << ": '" << data.first << "' "
8                << value_desc << ": '" << data.second << "'\n";
9    }
10 }
```

# Structured Bindings:

```
1  template<typename Map>
2  void print_map(const Map &map, const std::string &key_desc = "key",
3                                 const std::string &value_desc = "value")
4  {
5    for (const auto &[key, value] : map) /// structured binding
6    {
7      std::cout << key_desc << ": '" << key << "' "
8                << value_desc << ": '" << value << "'\n";
9    }
10 }
```

# #15: Structured Bindings

# #15: Structured Bindings

Used to decompose a structure or array into a set of identifiers. You *must* use `auto`, and the number of elements must match. There's no way to skip an element.

```
1 | const auto &[elem1, elem2] = some_thing;
```

- Standard: C++17
- Credit: Jens Maurer, Herb Sutter, Bjarne Stroustrup, Gabriel dos Reis

The whole `template` syntax is a bit bulky. Is there any way to simplify that?

```
1  // C++20's `auto` concept or further constrained to something that
2  // has values that can be destructured into 2 elements.
3  void print_map(const auto &map, const std::string &key_desc = "key",
4                                  const std::string &value_desc = "value")
5  {
6    for (const auto &[key, value] : map) /// structured binding
7    {
8      std::cout << key_desc << ": '" << key << "' "
9                << value_desc << ": '" << value << "'\n";
10   }
11 }
```

Say we want to have two versions of a function, one takes a Floating Point, the other an Integral value.

If we want to do this in C++11 it looks like this:

```
1  template<typename T,
2           typename std::enable_if<std::is_floating_point<T>::value, int>::type = 0>
3  auto func(T f) -> decltype(f * 3) { return f * 3; }
4
5  template<typename T,
6           typename std::enable_if<std::is_integral<T>::value, int>::type = 0>
7  auto func(T i) -> decltype(i + 3) { return i + 3; }
```

By the time we get to C++17 it looks like this:

```
template<typename T,
         std::enable_if_t<std::is_floating_point_v<T>, int> = 0>
auto func(T f) { return f * 3; }

template<typename T,
         std::enable_if_t<std::is_integral_v<T>, int> = 0>
auto func(T i) { return i + 3; }
```

With concepts (relying on someone to have defined the appropriate concepts for us):

```
auto func(FloatingPoint auto f) { return f * 3; }
auto func(Integral auto i) { return i + 3; }
```

# The concept definitions themselves can be quite simple:

```
1  template <class T> concept Integral = std::is_integral<T>::value;
2
3  template <class T> concept FloatingPoint = std::is_floating_point<T>::value;
```

# #16: Concepts

# #16: Concepts

Allow us to specify the requirements for a type, implicitly creating a template that constrains how a function can be used.

- Standard: C++20
- Credit: Andrew Sutton, Casey Carter, Eric Niebler (plus many others, there is a long history to concepts).

But there's a potential inefficiency hiding here.

```
1  void print_map(const auto &map, const std::string &key_desc = "key",
2                                  const std::string &value_desc = "value")
3  {
4    for (const auto &[key, value] : map)
5    {
6      std::cout << key_desc << ": '" << key << "' "
7                << value_desc << ": '" << value << "'\n";
8    }
9  }
10
11 int main()
12 {
13   print_map(get_some_map(), "index", "location");
14 }
```

We are constructing a `std::string` from a `const char *` for no particular reason.

If only there was a way to observe string-like things without actually constructing a `std::string`...

```
1  void print_map(const auto &map, std::string_view key_desc = "key",
2                                  std::string_view value_desc = "value")
3  {
4    for (const auto &[key, value] : map)
5    {
6      std::cout << key_desc << ": '" << key << "' "
7                << value_desc << ": '" << value << "'\n";
8    }
9  }
10
11 int main()
12 {
13   print_map(get_some_map(), "index", "location");
14 }
```

# #17: `std::string_view`

# #17: `std::string_view`

A non-owning “view” of a string like structure.

```
1 | std::string_view sv{some_string_like_thing}; // no copy
```

- Standard: C++17
- Credit: Andrew Sutton, Casey Carter, Eric Niebler

`std::cout` is quite verbose, relatively slow, and difficult to reason about. If only there was some easier way of formatting our output…

```
1  void print_map(const auto &map, const std::string_view key_desc = "key",
2                                  const std::string_view value_desc = "value")
3  {
4    for (const auto &[key, value] : map)
5    {
6      std::cout << std::format("{}: '{}' {}: '{}'\n",
7                               key_desc, key, value_desc, value);
8    }
9  }
```

# #18: Text Formatting

# #18: Text Formatting

A subset of the excellent {fmt} library is being worked on for C++20, allowing for formatting of strings with positional, named, and python/printf style formatting options.

```
1 std::string s = fmt::format("I'd rather be {1} than {0}.", "right", "happy");
2 // s == "I'd rather be happy than right."
```

- Standard: C++20 (Hopefully)
- Credit: Victor Zverovich

Bringing together algorithms, text formatting, concepts, etc, our map printing routine might look something like this in C++20:

```
1  void print_map(const auto &map, const std::string_view key_desc = "key",
2                                  const std::string_view value_desc = "value")
3  {
4    const auto print_key_value = [&](const auto &data) {
5      const auto &[key, value] = data;
6      std::cout << std::format("{}: '{}' {}: '{}'\n",
7                               key_desc, key, value_desc, value);
8    };
9
10   for_each(begin(map), end(map), print_key_value);
11 }
```

Now if only there was some way to not have to do this `begin(map)`, `end(map)` stuff when using algorithms…

```
1  void print_map(const auto &map, const std::string_view key_desc = "key",
2                                  const std::string_view value_desc = "value")
3  {
4    const auto print_key_value = [&](const auto &data) {
5      const auto &[key, value] = data;
6      std::cout << std::format("{}: '{}' {}: '{}'\n",
7                               key_desc, key, value_desc, value);
8    };
9
10   for_each(map, print_key_value);
11   // map | for_each(print_key_value);
12 }
```

# #19: Ranges

# #19: Ranges

Example from cppreference.com:

```
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
  std::vector<int> ints{0,1,2,3,4,5};
  auto even = [](int i){ return 0 == i % 2; };
  auto square = [](int i) { return i * i; };

  for (int i : ints | std::view::filter(even) | std::view::transform(square)) {
    std::cout << i << ' ';
  }
}
```

- Standard: C++20
- Credit: Eric Niebler, Casey Carter

# Let’s get back to our data creator:

```
1  /// require at least one parameter and it sets the type
2  template<typename VT, typename ... Params>
3  std::array<VT, sizeof...(Params)+1> get_data(const VT &v1, const Params &...params)
4  {
5    return {v1, params...};
6  }
```

If we apply `auto` return types we get:

```
1  /// require at least one parameter and it sets the type
2  template<typename VT, typename ... Params>
3  auto get_data(const VT &v1, const Params & ... params)
4  {
5    return std::array<VT, sizeof...(Params) + 1>{v1, params...};
6  }
```

This is… less than helpful.

If only there was a way to automatically deduce the type of the `array` being returned…

```
1  template<typename VT, typename ... Params>
2  auto get_data(const VT &v1, const Params & ... params)
3  {
4    return std::array{v1, params...}; /// auto deduce size/type
5  }
```

# #20: Class Template Argument Deduction

# #20: Class Template Argument Deduction

Just how function template arguments have always been deduced, `class` templates can be as of C++17.

```
1 | std::vector vec{1,2,3}; // now possible
```

For types like `array` this has an even bigger impact.

- Standard: C++17
- Credit: Mike Spertus, Faisal Vali, Richard Smith

And now we can simplify the template arguments:

```
1  template<typename ... Params>
2  auto get_data(const Params & ... params)
3  {
4    return std::array{params...};
5  }
```

(note: This might make using this function harder as it would result in a compile-time error if the types differ.)

And with C++20's `auto` concept we might get:

```
1  auto get_data(const auto & ... params)
2  {
3    return std::array{params...};
4  }
```

The only problem now is that this code requires the types to be copyable.

If only there was a way to forward arguments and avoid copies…

```
1  auto get_data(auto && ... params)
2  {
3    return std::array{std::forward<decltype(params)>(params)...};
4  }
```

No unnecessary copies of params, guaranteed copy/move elision of return value.

Admittedly this does not make C++ easier to teach, but it does allow for a high level of efficiency. But at this point one questions if `get_data()` is even necessary.

# #21: rvalue References

# #21: rvalue References

Can be difficult to teach, but undeniably fix a few holes in the language and allow us to more accurately reason about the lifetime of objects.

- Standard: C++11
- Credit: Howard Hinnant, Douglas Gregor, David Abrahams, Bjarne Stroustrup, Lawrence Crowl

# #22: Guaranteed Copy Elision

# #22: Guaranteed Copy Elision

Compilers have “always” done copy elision on return values, but C++17 guarantees it in some situations:

```
1  struct S {
2    S() = default;
3    S(&&) = delete;
4    S(const S &) = delete;
5  };
6
7  auto s_factory() {
8    return S{}; // compiles in C++17, neither a copy nor a move
9  }
```

This lets us teach simply “avoid naming return values” and you get the optimal code in all situations.

- Standard: C++17
- Credit: Richard Smith

# We promised we'd come back to this one:

```
1  struct Double_Data {
2    Double_Data(const std::size_t size) : data(new double[size]) { }
3    ~Double_Data() { delete [] data; }
4    double *data;
5  };
6
7  Double_Data get_data() {
8    Double_Data data(3);
9    data.data[0] = 1.1; data.data[1] = 2.2; data.data[2] = 3.3;
10   return data;
11 }
12
13 double sum_data(const Double_Data &d) {
14   return d.data[0] + d.data[1] + d.data[2];
15 }
16
17 int main() {
18   return sum_data(get_data()); /// no leak, but we'll come back to this
19 }
```

How is this broken (C++98)?

This was a copy on return in C++98 (But all compilers implemented copy elision, so it worked).

```
1  struct Double_Data {
2    Double_Data(const std::size_t size) : data(new double[size]) { }
3    ~Double_Data() { delete [] data; }
4    double *data;
5  };
6
7  Double_Data get_data() {
8    Double_Data data(3);
9    data.data[0] = 1.1; data.data[1] = 2.2; data.data[2] = 3.3;
10   return data; /// copy, which means what?
11 }
12
13 double sum_data(const Double_Data &d) {
14   return d.data[0] + d.data[1] + d.data[2];
15 }
16
17 int main() {
18   return sum_data(get_data());
19 }
```

We get a double free :(

In C++11 it's a move, but by defining a destructor we made it a copy, (But all compilers implemented copy elision, so it worked).

```
1  struct Double_Data {
2    Double_Data(const std::size_t size) : data(new double[size]) { }
3    ~Double_Data() { delete [] data; }
4    double *data;
5  };
6
7  Double_Data get_data() {
8    Double_Data data(3);
9    data.data[0] = 1.1; data.data[1] = 2.2; data.data[2] = 3.3;
10   return data; /// move, which means what?
11 }
12
13 double sum_data(const Double_Data &d) {
14   return d.data[0] + d.data[1] + d.data[2];
15 }
16
17 int main() {
18   return sum_data(get_data());
19 }
```

We get a double free :( Default move constructor will copy the pointer.

# If only there was some way to disable problematic operations…

```
1  struct Double_Data {
2    Double_Data(const std::size_t size) : data(new double[size]) { }
3    ~Double_Data() { delete [] data; }
4    Double_Data(Double_Data &&) = delete;
5    Double_Data(const Double_Data &) = delete;
6    Double_Data &operator=(Double_Data &&) = delete;
7    Double_Data &operator=(const Double_Data &) = delete;
8    double *data;
9  };
```

Not very useful at the moment, but we know it’s safe.

# #23: Defaulted and Deleted Functions

# #23: Defaulted and Deleted Functions

- Any special member function can be explicitly `= default`.
- Any function can be `= delete`.
- This can make your API harder to use wrong.

```
1  struct S {
2    // by providing a constructor we have implicitly
3    // disabled the default constructor
4    S(int) {}
5
6    // explicitly default the default constructor
7    S() = default;
8  };
```

- Standard: C++11
- Credit: Lawrence Crowl

If only there was a way of not even having to think about heap allocated memory (other than containers mentioned previously, use those of course!)…

```
1  struct Double_Data {
2    Double_Data(const std::size_t size)
3      : data(std::make_unique<double[]>(size))
4    {
5    }
6
7    std::unique_ptr<double[]> data;
8  };
```

# #24: `std::unique_ptr` and `std::make_unique`

# #24: `std::unique_ptr` and `std::make_unique`

Automatic, safe, very difficult to use incorrectly.

Interestingly, relies on:

- `= delete`d special member functions
- destructors
- r-value references

And plays nicely with guaranteed copy/move elision for factory functions.

- Standard: C++11/14
- Credit: … ?

# Let’s make a quick sum function.

```
1  template<typename ... T>
2  auto sum(const T & ... t)
3  {
4    // how do we sum this?
5  }
```

We can do it recursively:

```
1  template<typename T>
2  auto sum(const T &t)
3  {
4    return t;
5  }
6
7  template<typename First, typename Second, typename ... T>
8  auto sum(const First &first, const Second &second, const T & ... t)
9  {
10   return first + sum(second, t...);
11 }
```

But this can be horribly slow at compile time, difficult to optimize, and difficult to debug.

We can do it with an interesting `initializer_list` hack...

```
1  template<typename First, typename ... T>
2  auto sum(const First &first, const T & ... t)
3  {
4    // trick was first published by Sean Parent and Eric Niebler.
5    auto result = first;
6    (void)std::initializer_list<int>{ (result += t, 0) ... };
7    return result;
8  }
```

Avoids the recursion but fixes the result type as the type of the first parameter.

# Corrected type version:

```
1  template<typename First, typename ... T>
2  auto sum(const First &first, const T & ... t)
3  {
4    typename std::common_type<first, T...>::type result = first;
5    (void)std::initializer_list<int>{ (result += t, 0) ... };
6    return result;
7  }
```

If only there was some way to ask the compiler to sum up this parameter set for us…

```
1  template<typename ... T>
2  auto sum(const T & ... t)
3  {
4    return (t + ...);
5  }
```

# #25: Fold Expressions

# #25: Fold Expressions

Fold expressions are for use in variadic expansions and can be used with any common operator.

- Standard: C++17
- Credit: Andrew Sutton, Richard Smith

# Key C++98 Features

1. A C++ Standard
2. `const`
3. Deterministic Object Lifetime and Destruction
4. Templates
5. Algorithms and Standard Template Library

# Key C++11 Features

1. std::array
2. List Initialization
3. Variadic Templates
4. `constexpr`
5. `auto`
6. Lambdas
7. range-based `for` loops
8. rvalue References
9. Defaulted and Deleted Functions
10. `std::unique_ptr`

# Key C++14 Features

1. relaxed `constexpr`
2. generic and variadic lambdas
3. return type deduction for normal functions
4. `std::make_unique`

# Key C++17 Features

1. Structured Bindings
2. `std::string_view`
3. Class Template Argument Deduction
4. Guaranteed Copy Elision
5. Fold Expressions

# Key C++20 Features

1. Concepts
2. Text Formatting
3. Ranges
4. Contracts, but we ran out of time!

# Bonus Feature

Remember this code?

```
1 constexpr double pi() {
2   return 22/7;
3 }
```

What's the return value?

3.

# Bonus Feature

How can we catch something like this?

```
1 constexpr double pi() {
2   return 22/7;
3 }
```

Tools!

# Bonus Feature

We are surrounded by amazing tools that can catch many errors in our code:

- sanitizers
- compilers
- static analyzers
- testing frameworks
- fuzzers

And update it:

- clang-tidy with modernize

## What do we see?

- Common idioms get easier to express
- Language semantics get more consistent (`...` in more places, `auto` in more places)
- Memory management issues almost *go away*
- We have to think less about the lifetime of our objects
- It's easier to construct libraries that are hard to use wrong (don't underestimate this).
- And we haven't even hit on the power of contracts or properly constraining function arguments with concepts

The features we mentioned here are thanks to:

Alexander Stepanov, Andrew Sutton, Beman Dawes, Bjarne Stroustrup, Casey Carter, Christof Meerwald, Dave Abrahams, Daveed Vandevoorde, David Abrahams, Douglas Gregor, Eric Niebler, Faisal Vali, Gabriel Dos Reis, Herb Sutter, Howard Hinnant, J. Stephen Adamczyk, Jaakko Järvi, Jason Merrill, Jens Maurer, John Freeman, John Spicer, Lawrence Crowl, Meng Lee, Mike Spertus, Nicolai Josuttis, Peter Dimov, Richard Smith, Victor Zverovich

Plus many other names I couldn’t find.

# Jason Turner

- Co-host of CppCast https://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Projects
  - https://chaiscript.com
  - https://cppbestpractices.com
  - https://github.com/lefticus/cpp_box
  - https://coloradoplusplus.info
- Microsoft MVP for C++ 2015-present

# Jason Turner

Independent and available for training or contracting

- https://articles.emptycrate.com/idocpp

# Upcoming Events

- NDC TechTown - Sept 2-3, 2019 - Moving to C++
- CppCon - Sept 21, 2019 - Applied `constexpr` - Doing More At Compile-Time