Back to Basics: Templates – Part 2

Bob Steagall
CppCon 2021

KEWB COMPUTING

Recap: Templates

- C++ supports generic programming with templates
  - A template is a *parametrized description* of a family of some facility
- A template is not a *thing* – it is a recipe for making *things*
- C++ provides six kinds of templates
  - Function templates
  - Class templates
  - Member function templates
  - Alias template
  - Variable templates
  - Lambda templates

Recap: Translation Units

- In C++, translation is performed in nine well-defined stages
- Phases 1 through 6 perform lexical analysis (the pre-processor)
- The output of Phase 6 is a **translation unit**
- A translation unit is defined [5.1] as
  - A source file
  - Plus all the headers and source files included via `#include` directives
  - Minus any source lines skipped by conditional inclusion preprocessing directives (#ifdef)
  - And all macros expanded

Recap: Declarations and Definitions

- A **name** is an identifier that denotes an entity
  - Every template has a name
  - Every template specialization has a name, formally known as a **template-id**

- A **declaration** introduces one or more **names** into a translation unit
  - A declaration may also *re-introduce* a name into a translation unit

- A **definition** is a declaration that fully defines the entity being introduced

The One-Definition Rule (ODR)

- A program must contain exactly one definition of every non-inline variable or function that is used in the program

- For an inline variable or an inline function, a definition is required in every translation unit that uses it
  - `inline` evolved to mean "multiple definitions are permitted"

- Exactly one definition of a class must appear in any translation unit that uses it in such a way that the class must be complete

- The rules for inline variables and functions **also apply to templates**

Recap: Template Parameters and Template Arguments

```
template<class T1, class T2>
struct pair
{
    T1  first;
    T2  second;
    ...
};

template<class T>
T const&  max(T const& a, T const& b)
{ ... }
```

Template Parameters

```
pair<string, double>  my_pair;

double d = max<double>(0, 1);

string s1 = ...;
string s2 = ...;
string s3 = max(s1, s2);
```

Template Arguments

- **Template parameters** are the names that come after the `template` keyword in a template declaration

- **Template arguments** are the concrete items **substituted** for template parameters to create a template **specialization**

Recap: Templates and Specializations

```
template<class T1, class T2>
struct pair
{
    T1  first;
    T2  second;
    ...
};

template<class T>
T const&  max(T const& a, T const& b)
{ ... }
```

Template

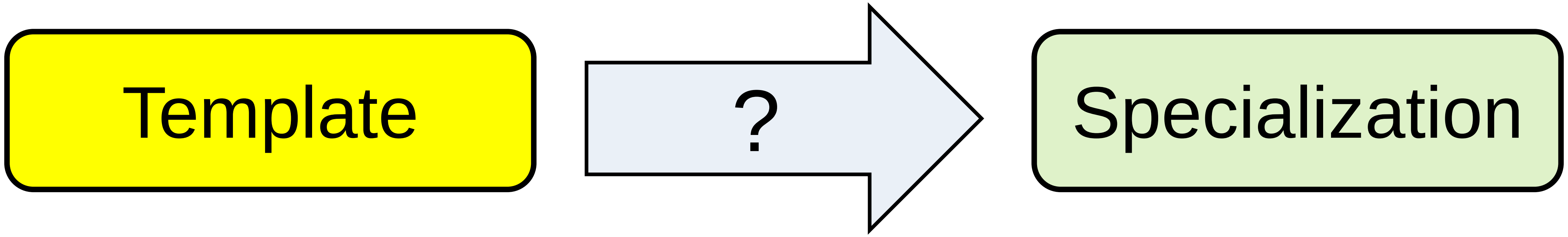
```
pair<string, double>  my_pair;

double d = max<double>(0, 1);

string s1 = ...;
string s2 = ...;
string s3 = max(s1, s2);
```

Specialization

- **Template parameters** are the names that come after the `template` keyword in a template declaration

- **Template arguments** are the concrete items **substituted** for template parameters to create a template **specialization**

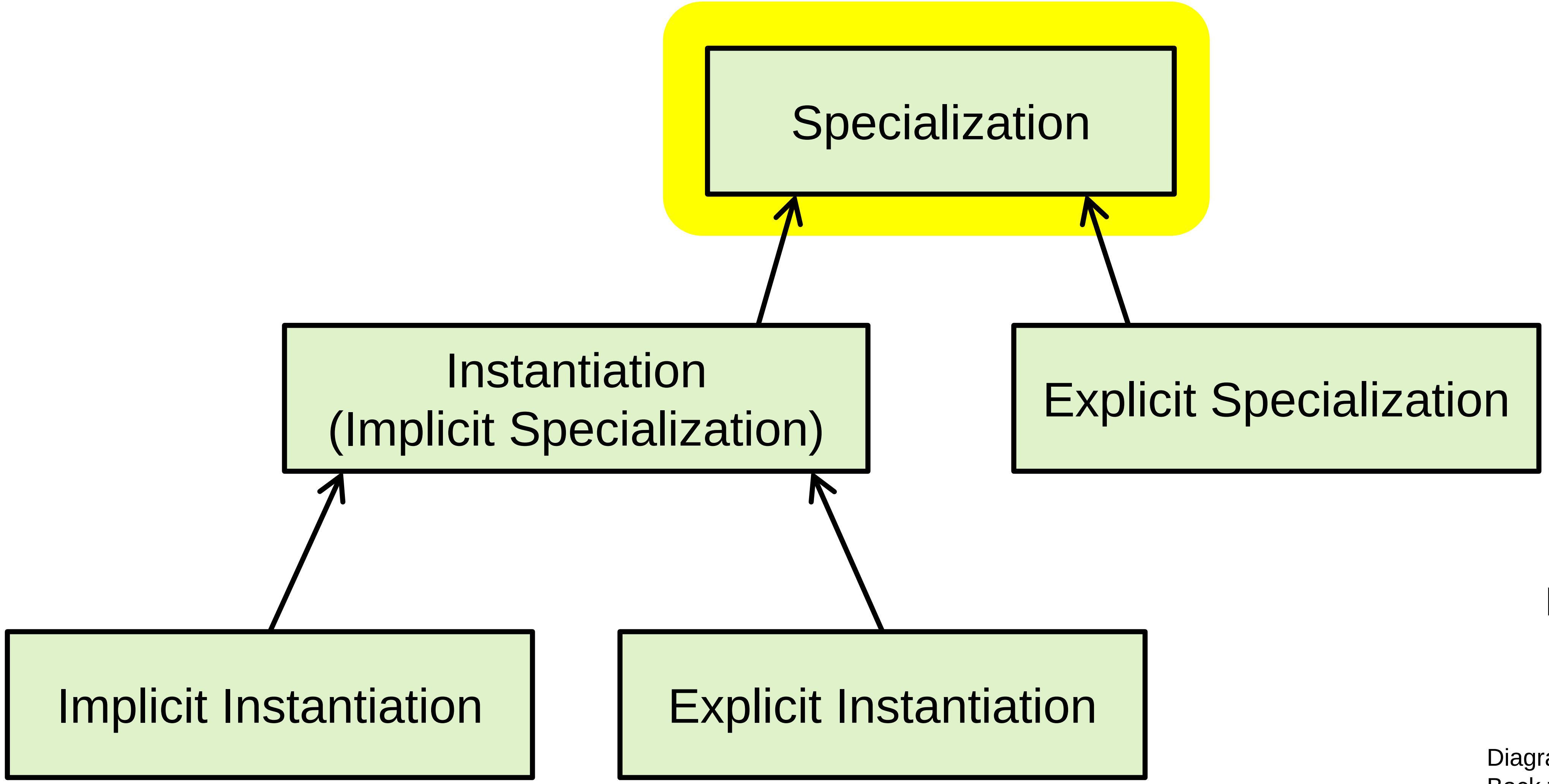
Recap: From Template to Specialization



- The template is a recipe that tells how to generate something useful
- A specialization is the useful thing built from that recipe
- We get from template to specialization in one of two ways:
  - **Instantiation**
  - **Explicit specialization**

Recap: Instantiation and Specialization

- A **specialization** is what results when template arguments are substituted for template parameters

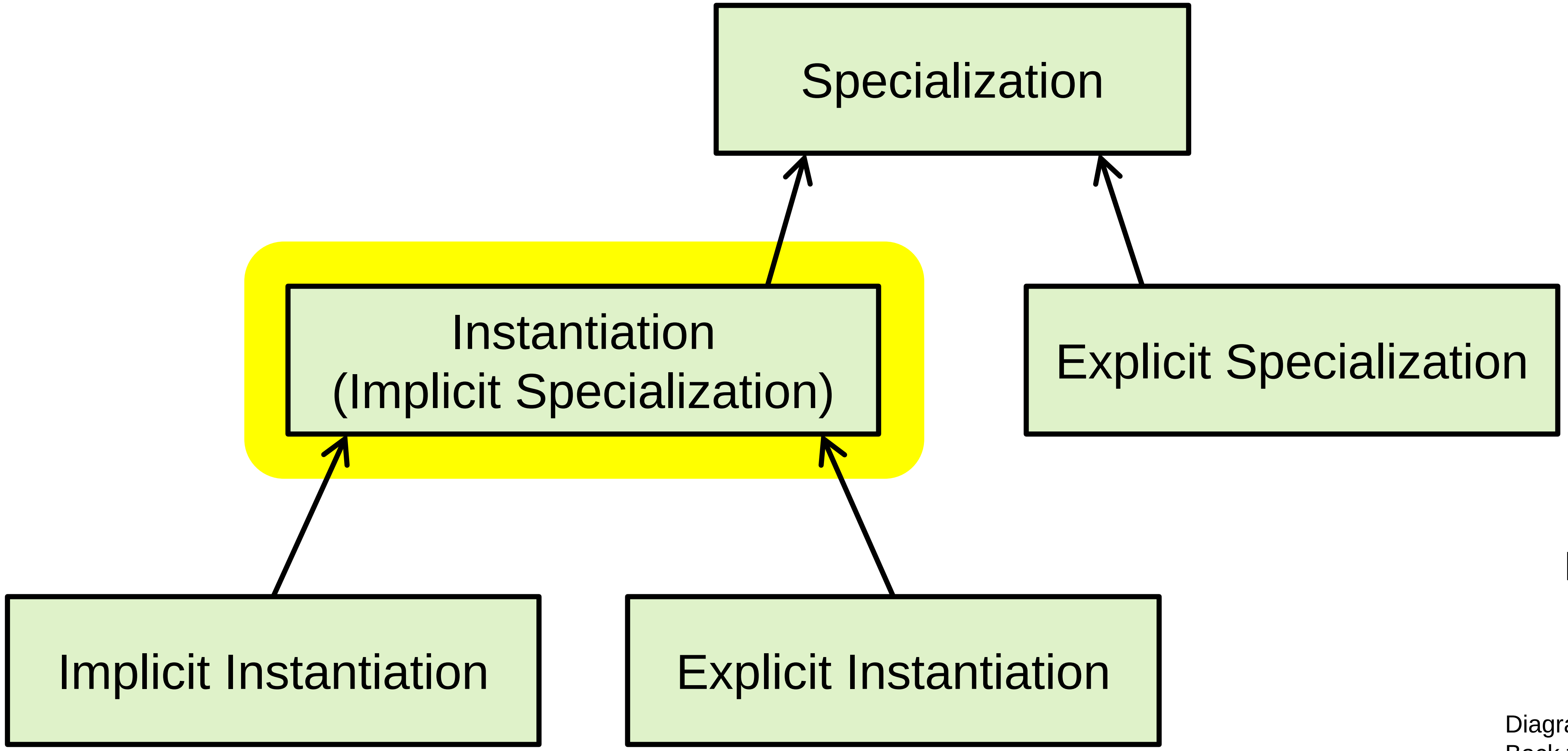


Note: arrow means *Is-A*

Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Recap: Instantiation and Specialization

- **Instantiation** occurs when the *compiler* generates a specialization from a template
  - *Instantiation* is a synonym for *compiler-generated specialization*

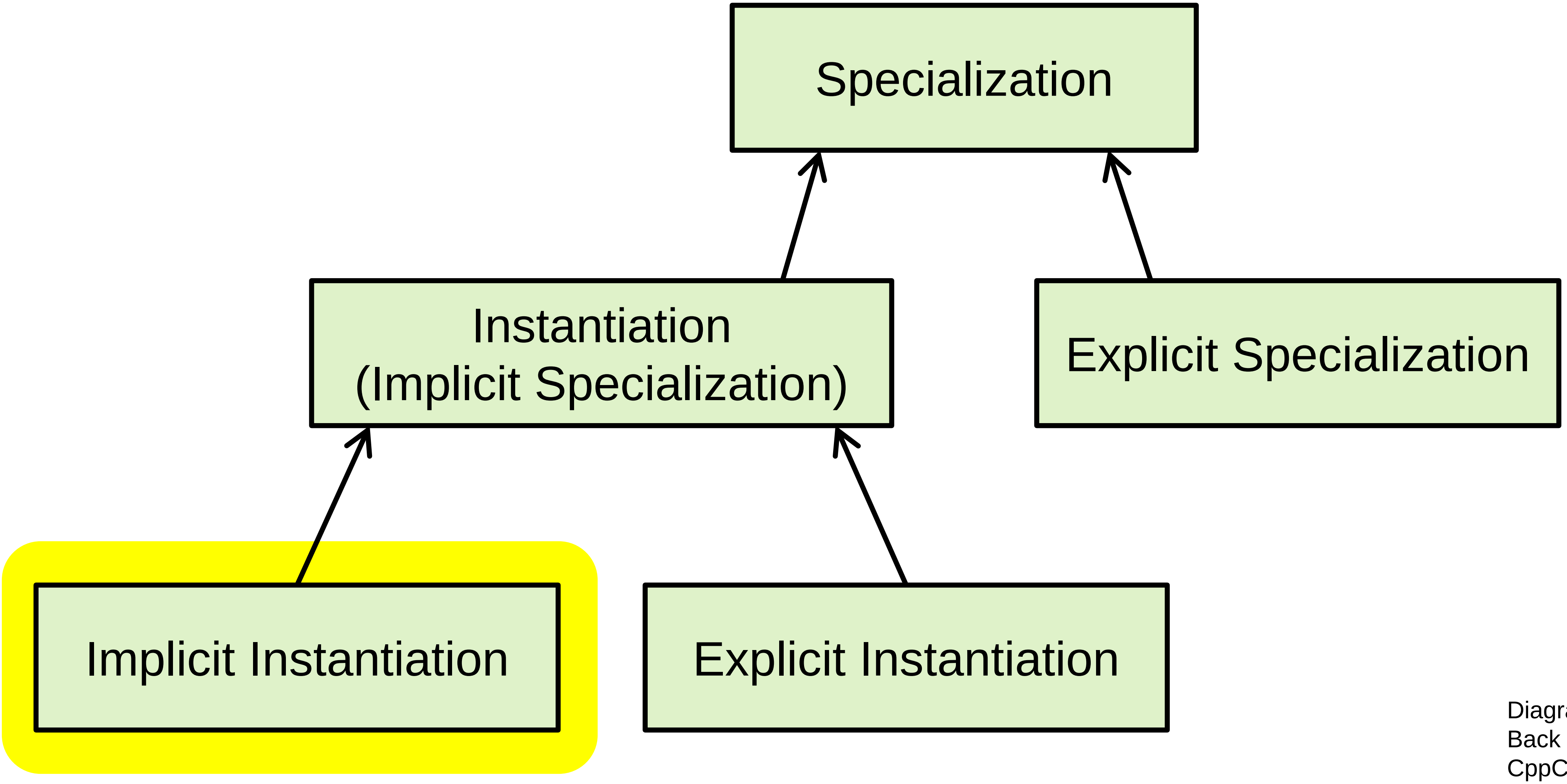


Note: arrow means *Is-A*

Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Recap: Instantiation and Specialization

- **Implicit instantiation** occurs when the compiler automatically generates a specialization upon seeing its use in code

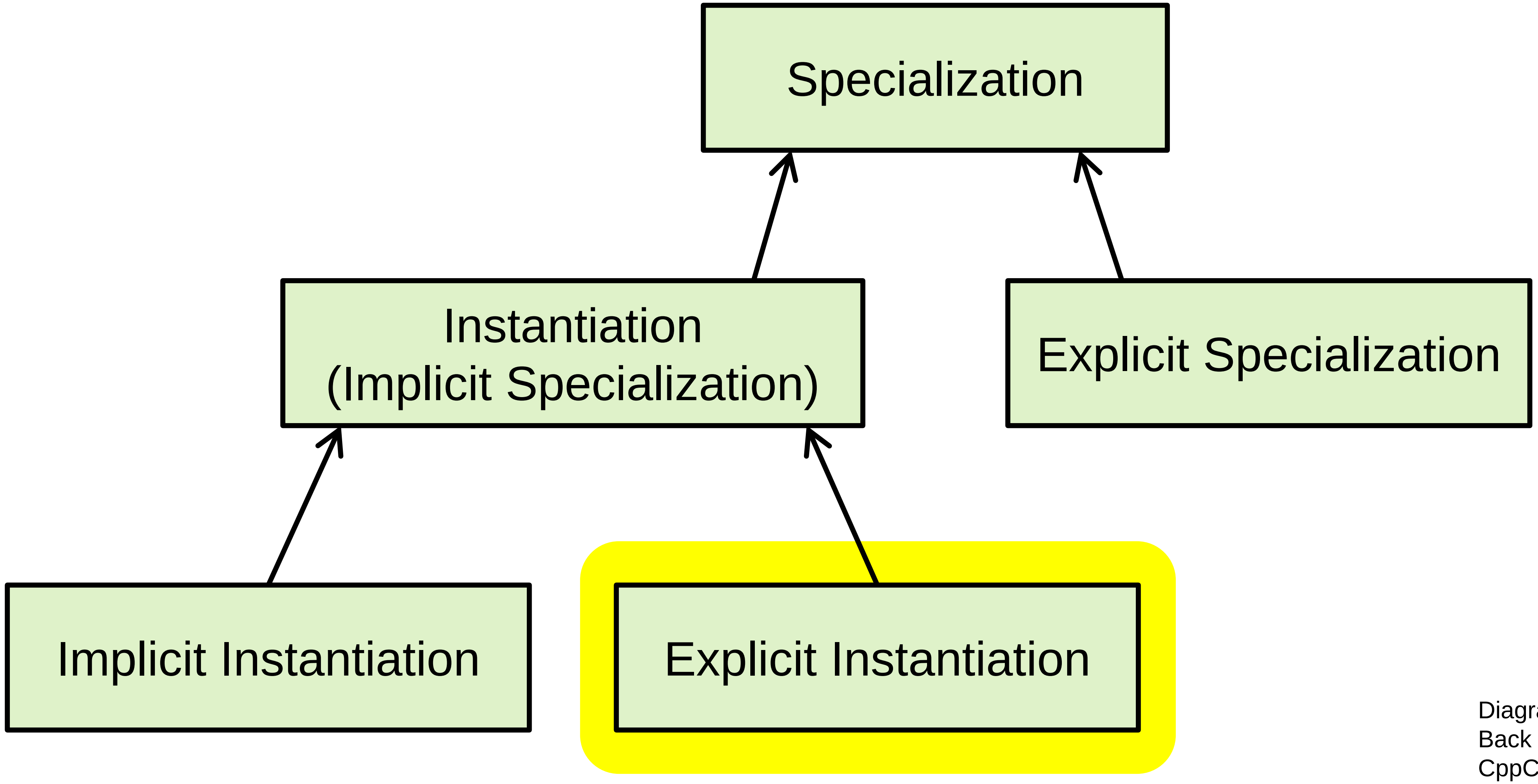


Note: arrow means *Is-A*

Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Recap: Instantiation and Specialization

- **Explicit instantiation** occurs when the compiler encounters an explicit directive in source code to force instantiation

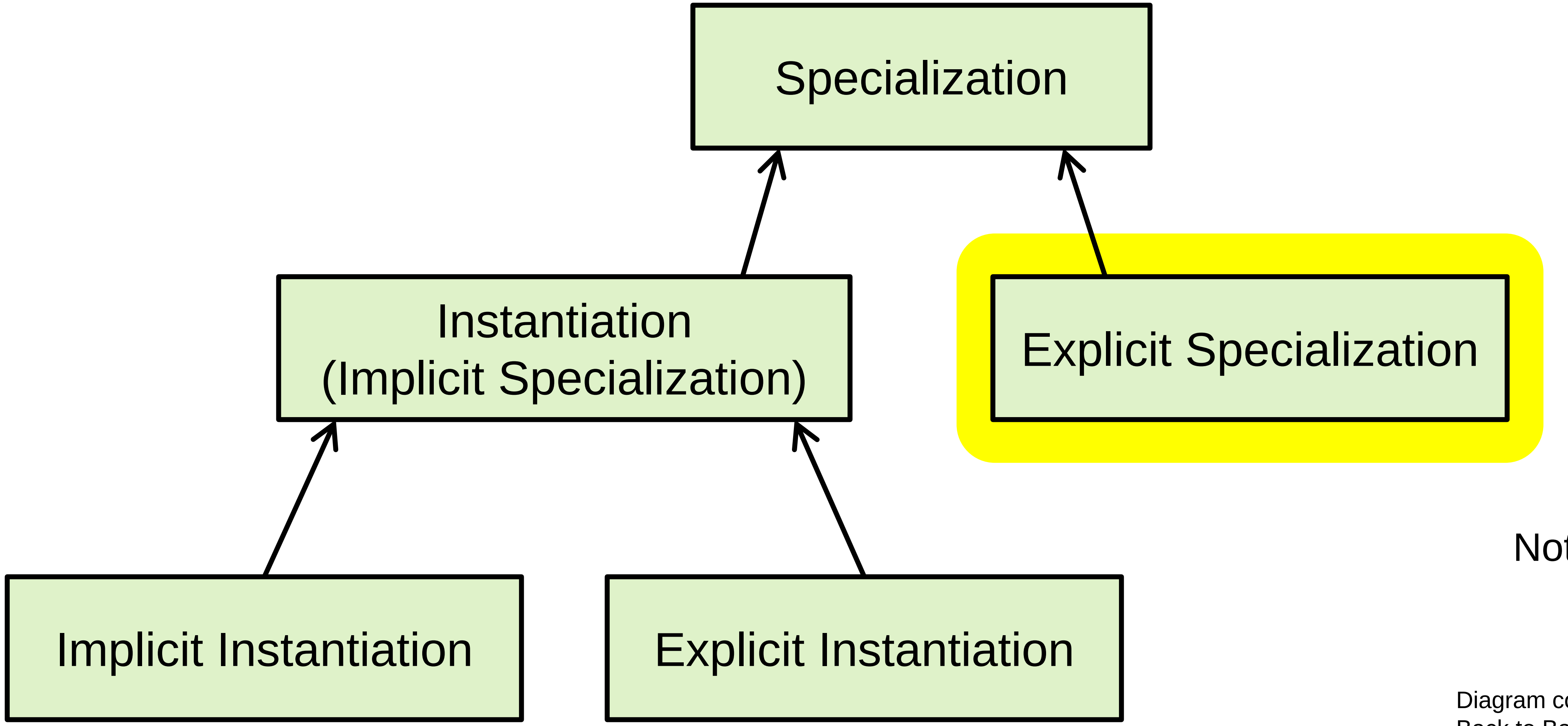


Note: arrow means *Is-A*

Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Recap: Instantiation and Specialization

- An **explicit specialization** is a user-provided implementation of a template with all template parameters fully substituted



Note: arrow means *Is-A*

Diagram courtesy of Dan Saks
Back to Basics: Function and Class Templates
CppCon 2019

Two-Phase Translation

- Templates are processed in two phases

- At the site of its definition, a template is checked for correctness, ignoring template parameters
  - Syntax errors are discovered
  - Names that don't depend on template parameters are looked up
  - Static assertions that don't depend on template parameters are checked

- At instantiation time
  - The template arguments are substituted for the parameters
  - Names depending on template parameters are looked up
  - All parts depending on template parameters are checked again

Two-Phase Translation

- Templates are processed in two phases
- At the site of its definition, a template is checked for correctness, ignoring template parameters
  - Syntax errors are discovered
  - Names that don't depend on template parameters are looked up
  - Static assertions that don't depend on template parameters are checked
- At instantiation time
  - The template arguments are substituted for the parameters
  - Names depending on template parameters are looked up
  - All parts depending on template parameters are checked again

Two-phase name lookup

Complete and Incomplete Types

- Types can be *complete* or *incomplete*
  - Related to the distinction between a definition and a declaration
  - Some situations require complete types, other situations work with incomplete types

- Incomplete types are
  - A class type declared but not defined
  - An array type with unspecified bound
  - `void`, and some others…

- Sometimes incomplete types will work as template parameters, sometimes not

```
template<class T>
struct node_base
{
   T* p_next;
};
```

```
template<class T>
struct node_val : public node_base<T>
{
   T  value;
};
```

Structuring Code That Uses Templates

- At some point we'll use the recipe and make an actual *thing*
- The compiler must be able to see the recipe to make the *thing*
- How do we ensure the recipe is visible when the compiler needs to see it?
- (1) We put our template code (recipe) in one or more headers
- (2) We include those headers in source files that need to use the recipe

Structuring Code That Uses Templates

```cpp
//- foobar.h
#ifndef FOOBAR_H_INC
#define FOOBAR_H_INC
#include <vector>

template<class T>
class foobar
{
    std::vector<T> stuff;

  public:
    foobar(int n=0);
    ...
    void add(T const& t)
    {
       stuff.push_back(t);
    }
};

void f1();
void f2();

#endif
```

```cpp
//- file_1.cpp
#include "foobar.h"
#include <string>
#include <complex>

using std::string;
using std::complex;

using cxfloat = complex<float>;

void f1(int n)
{
    foobar<string>   fbs(n);
    foobar<cxfloat>  fbc(n*s);

    ...

    fbs.add("Hello, world");
    fbc.add(cxfloat{1.0f, 2.0f});

    ...
}
```

```cpp
//- file_2.cpp
#include "foobar.h"
#include <string>

using std::string;

void f2(int n)
{
    foobar<string>  fbs(n);
    ...
}
```

```cpp
//- main.cpp
#include "foobar.h"

int main()
{
    f1();
    f2();
    return 0;
}
```

Structuring Code That Uses Templates

```
//- foobar.h
#ifndef FOOBAR_H_INC
#define FOOBAR_H_INC
#include <vector>

template<class T>
class foobar
{
    std::vector<T> stuff;

  public:
    foobar(int n=0);
    ...
    void add(T const& t)
    {
       stuff.push_back(t);
    }
};

void f1();
void f2();

#endif
```

```
//- file_1.cpp
#include "foobar.h"
#include <string>
#include <complex>

using std::string;
using std::complex;

using cxfloat = complex<float>;

void f1(int n)
{
    foobar<string>   fbs(n);
    foobar<cxfloat>  fbc(n*s);

    ...

    fbs.add("Hello, world");
    fbc.add(cxfloat{1.0f, 2.0f});
```

```
//- file_2.cpp
#include "foobar.h"
#include <string>

using std::string;

void f2(int n)
{
    foobar<string>  fbs(n);
    ...
}
```

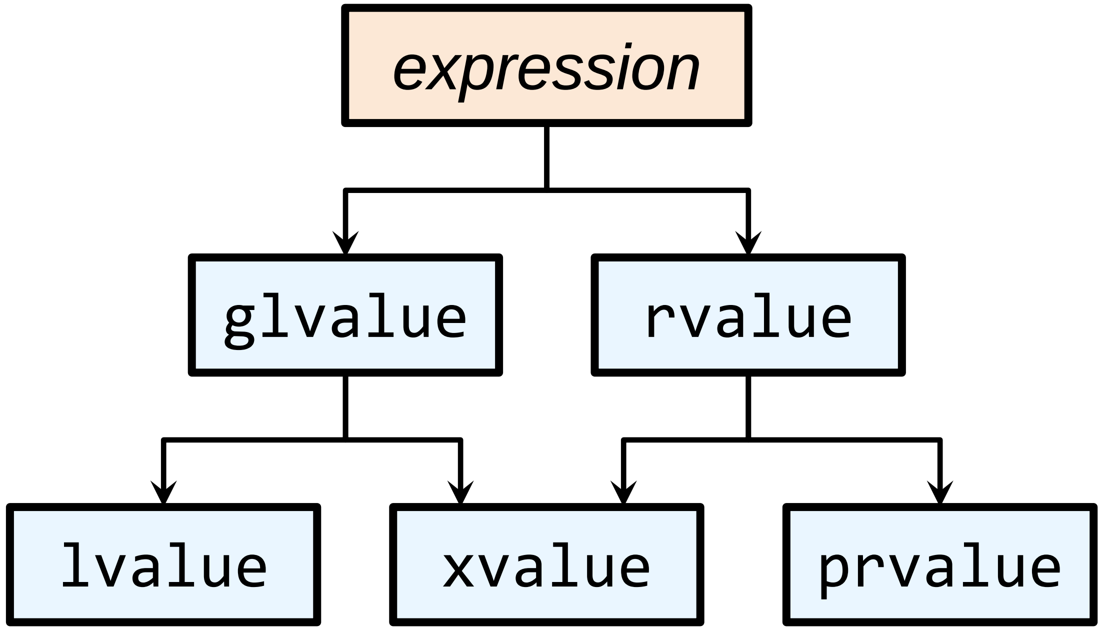
```
//- main.cpp
#include "foobar.h"

int main()
{
    f1();
    f2();
```

With GCC, you might build with something like:

```
$ g++ -std=c++17 file_1.cpp file_2.cpp main.cpp -o foobar
```

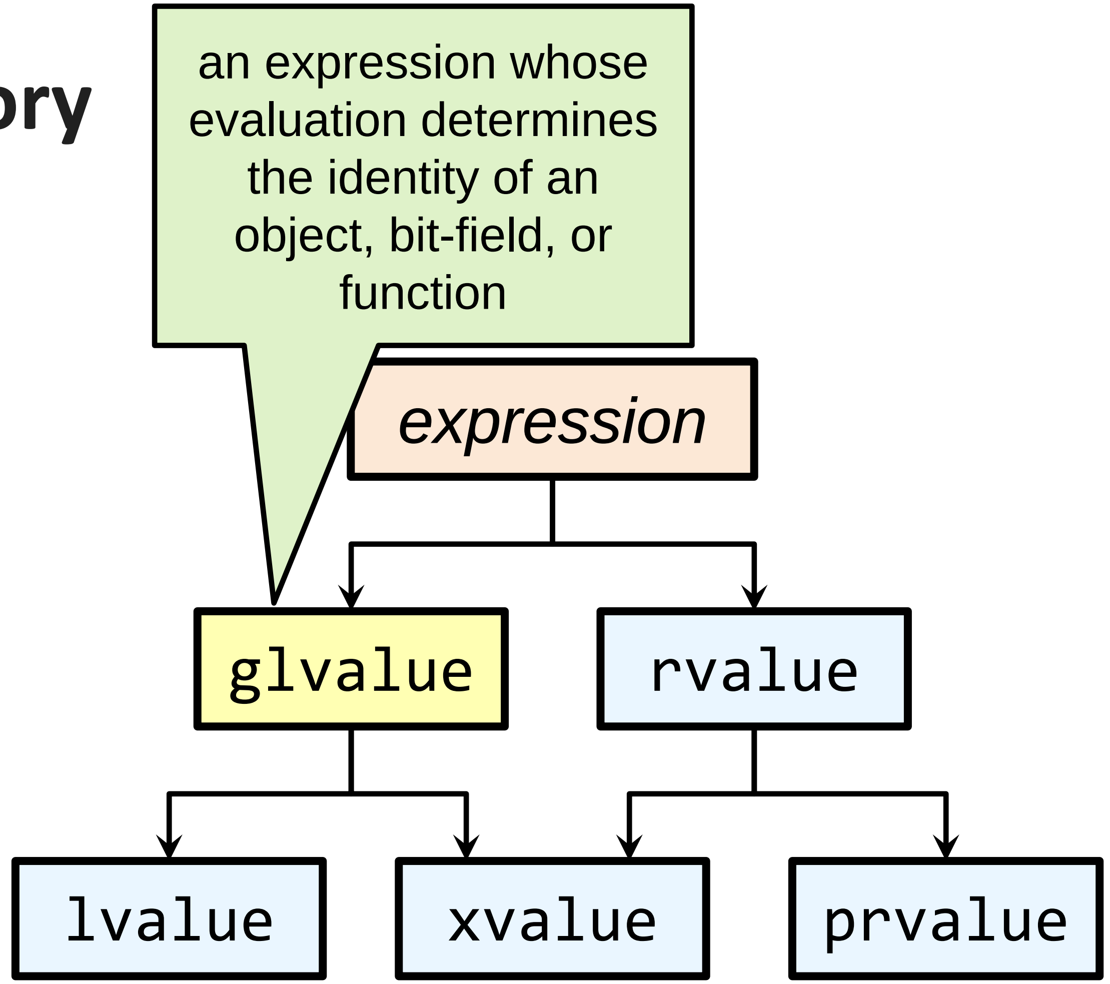
Value Categories: lvalues and rvalues

- Every C++ expression has an associated **type**
- Every C++ expression belongs to a **value category**
- The standard classifies all expressions into one of five value categories
  - Two (*glvalue* and *rvalue*) are *composite* categories
  - Three (*lvalue*, *xvalue*, and *prvalue*) are *core* categories



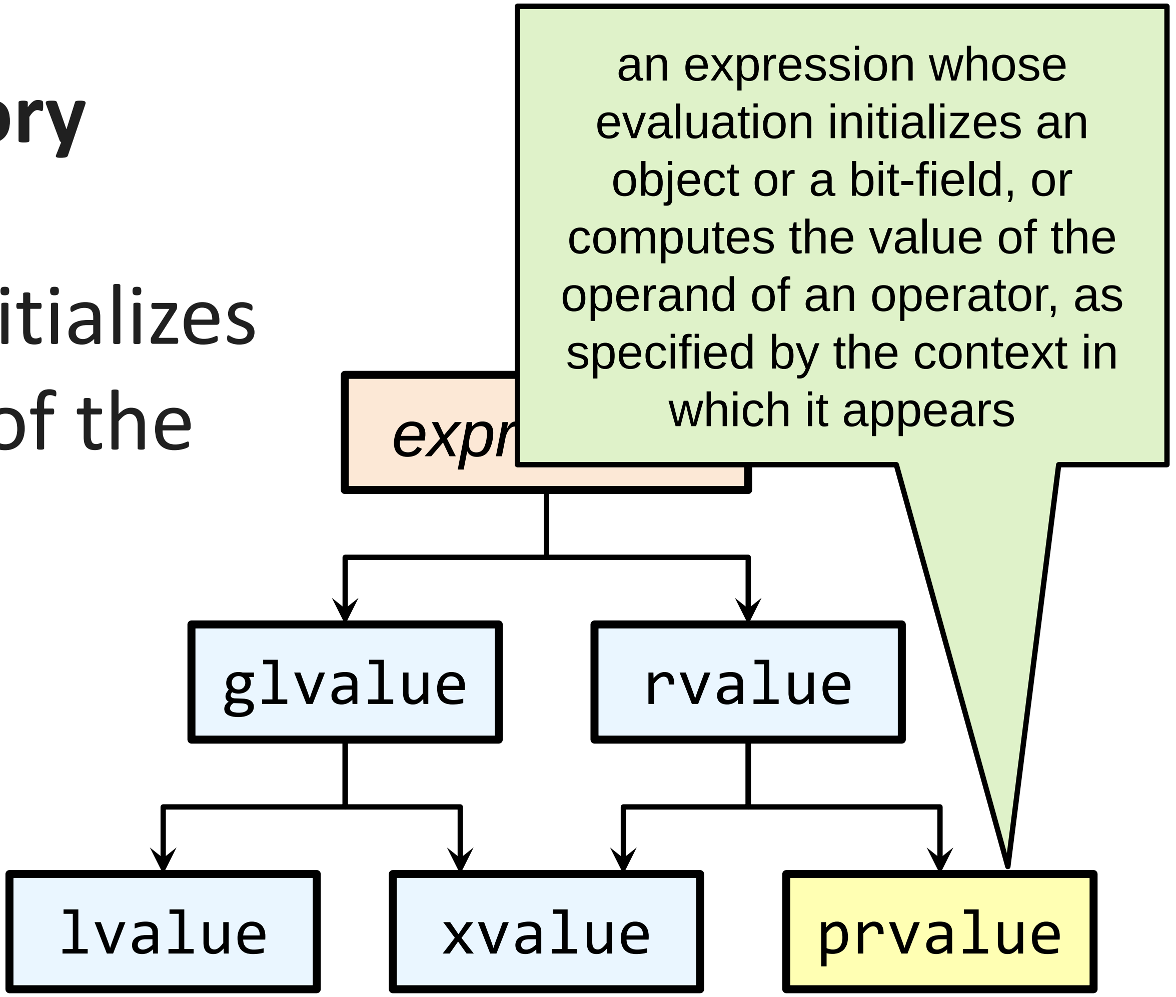
Value Categories: lvalues and rvalues

- Every C++ expression has an associated **type**
- Every C++ expression belongs to a **value category**
- A **glvalue** is an expression whose evaluation determines the *identity* of an object, bit-field, or function
  - Has storage
  - Has a name (directly or indirectly)
  - Has an address that can be taken*
  - Non-const glvalues can be assigned to*



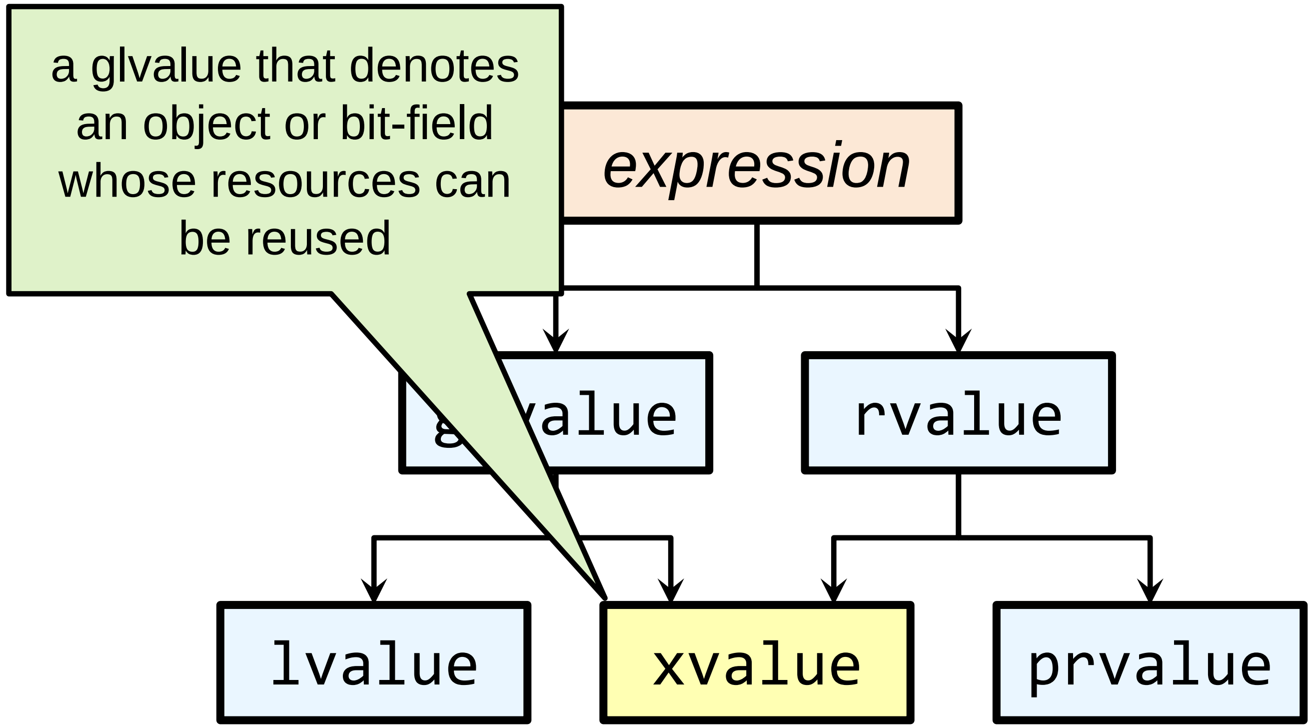
Value Categories: lvalues and rvalues

- Every C++ expression has an associated **type**
- Every C++ expression belongs to a **value category**
- A **prvalue** is an expression whose evaluation initializes an object or a bit-field, or computes the value of the operand of an operator
  - Has no name
  - Has no storage*



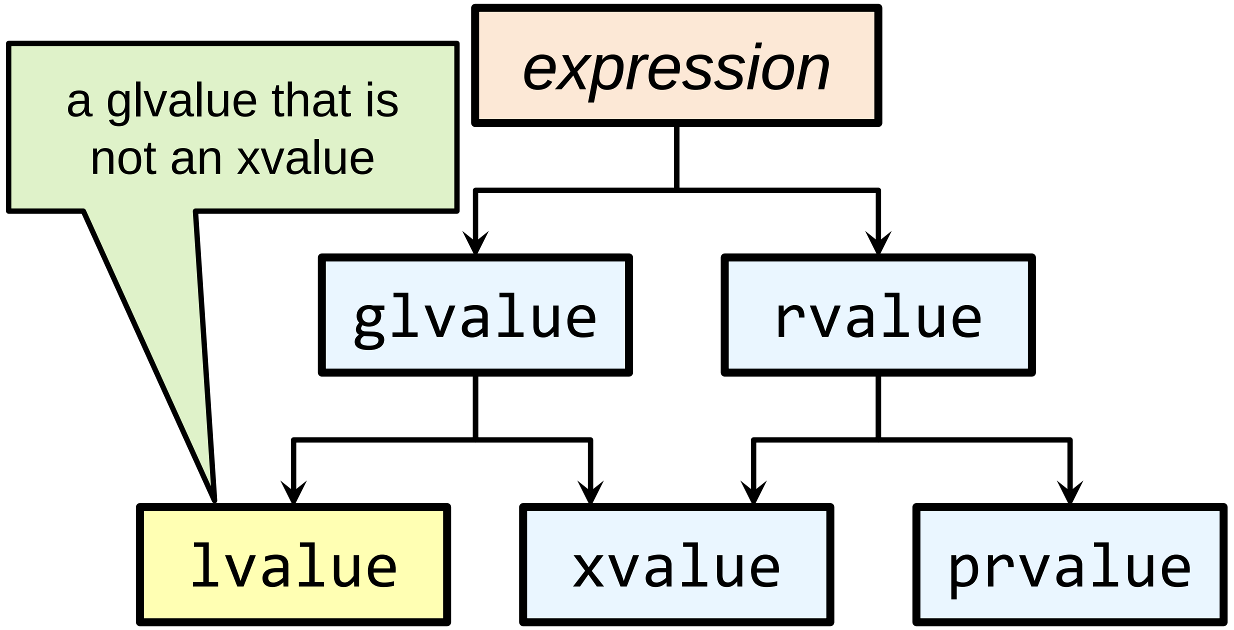
Value Categories: lvalues and rvalues

- Every C++ expression has an associated **type**
- Every C++ expression belongs to a **value category**
- An **xalue** is an expression denoting an object or bit-field whose resources can be reused
  - The "x" originally came from "eXpiring value"
  - Represents a *glvalue* whose value will soon not matter



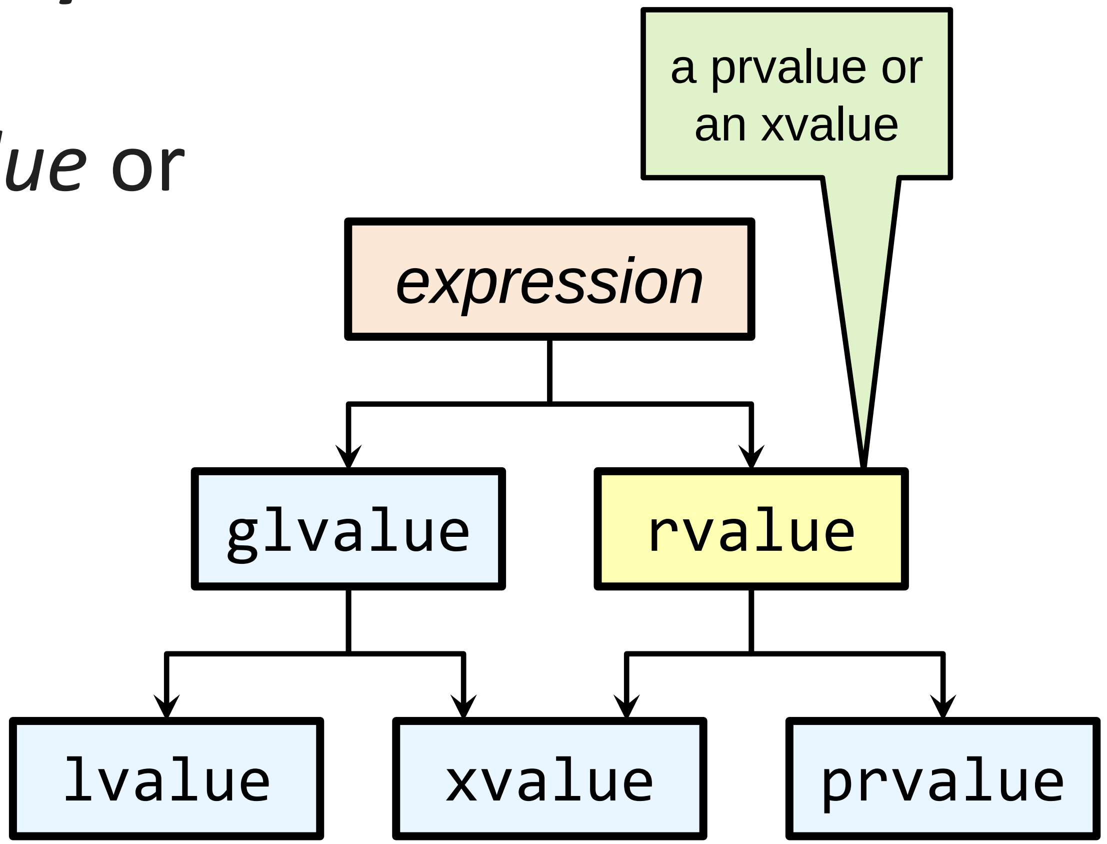
Value Categories: lvalues and rvalues

- Every C++ expression has an associated **type**
- Every C++ expression belongs to a **value category**
- An **lvalue** is a *glvalue* that is not an *xvalue*



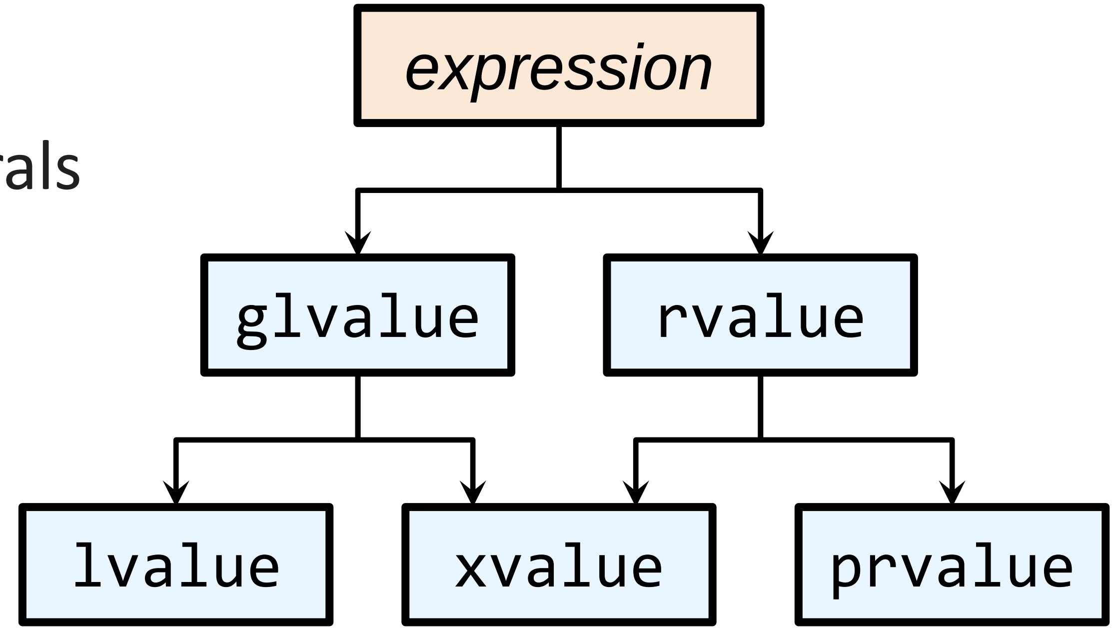
Value Categories: lvalues and rvalues

- Every C++ expression has an associated **type**
- Every C++ expression belongs to a **value category**
- An **rvalue** is an expression that is either a *prvalue* or an *xvalue*



Value Categories: lvalues and rvalues

- *lvalues* are things like
  - Expressions that designate variables or functions
  - Class data members
  - A call to a function that returns an *lvalue* reference
- *prvalues* are things like
  - Literals that are not user-defined literals or string literals
  - Application of built-in arithmetic operators
  - A call to a function with a non-reference return type
- *xvalues* are things like
  - A cast to an *rvalue* reference to an object type
  - A call to a function that returns an *rvalue* reference to an object type



Template Categories in Detail

More Detail – Function Templates

- A function template is a recipe for generating a parametrized family of functions

```
template<class T>
T const&  min(T const& a, T const& b)  //- Definition of function template min
{
    return (b < a) ? b : a;
}

string  s0 = "foo";
string  s1 = "bar";
string  s2 = min<string>(s0, s1);     //- Explicitly specify template argument is string
string  s3 = min(s0, s1);             //- Template argument is deduced as string

double  d1 = 3.14;
double  d2 = min<double>(2.78, d1);   //- Explicitly specify template argument is double
double  d3 = min(2.78, d1);           //- Template argument is deduced as double
```

Function Templates

- A function template is a recipe for generating a parametrized family of functions

```
template<class T>
T const&  min(T const& a, T const& b)  //- Definition of function template min
{
    return (b < a) ? b : a;
}

int     i = 42;
double  d = 3.14;
auto    x = min(d, i);                //- Error!  Ambiguity – is T int or double?
auto    y = min<double>(i, d);        //- OK, force T to be double
```

Template Argument Deduction

- Consider an example

```
template<class T, class U, class V = double>
int  f(T& t, U u, V const& v) { ... }

int     i = 0;
string  s = "hi";

int r = f(i, 1.0, s);
```

- How does the compiler determine the types of T, U, and V?

- **Template argument deduction** is the process by which the compiler determines the type of function template arguments based on function argument types

Template Argument Deduction

- Let's simplify to the case of a single parameter

```
template<class T>
void f(ParameterType t);

invoke_f(SomeExpression);  //- Type deduction occurs here
```

- Given the form of *`invoke_f`* and the type of *`SomeExpression`*, the compiler must determine
  - *`ParameterType`*, the type of the function argument `t`
  - `T`, the type of the template argument
  - *`ParameterType`* and `T` are not necessarily the same (e.g., `string const&` and `string`)

- To do so, the compiler performs a sort of pattern matching

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f<SomeTemplateArgument>(SomeExpression);     //- Type deduction occurs here
```

- Explicitly-specified template arguments are fixed as specified

```
template<class T>
void f(T t);

double  d = 2.78;

//- T is double
//  ParameterType is double
//
f<double>(d);
f<double>(3.14);
f<double>(42);
```

```
template<class T>
void f(T const& t);

double  d = 2.78;

//- T is double
//  ParameterType is double const&
//
f<double>(d);
f<double>(3.14);
f<double>(42);
```

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f(SomeExpression);          //- Type deduction occurs here
```

- Otherwise, template argument types must be inferred from the context

```
template<class T>
void f(T t);

double        d   = 2.78;
double&       rd  = d;
double const& crd = d;
string  g();

f(3.14);
f(d);
f(rd);
f(crd)
f(g());
```

```
template<class T>
void f(T const& t);

double        d   = 2.78;
double&       rd  = d;
double const& crd = d;
string  g();

f(3.14);
f(d);
f(rd);
f(crd)
f(g());
```

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

- What are the possible forms that *`ParameterType`* could have?
  - `T`
  - `T*`
  - `T const*`
  - `T&`
  - `T const&`
  - `T&&`
  - `T const&&`

- What are the possible types that *`SomeExpression`* could have (assuming `int` is involved)?
  - `int`
  - `int const`
  - `int*`
  - `int const*`
  - `int&`
  - `int const&`
  - `int &&`
  - `int const&&`

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

What is the deduced type T?

Form of *ParameterType*

Type of *SomeExpression*	T	T const*	T const&	T const&&	T*	T&	T&&
int const	int	-	int	int	-	-	int
int const*	int const*	int	int const*	int const*	int const	-	int const*
int const&	int	-	int	-	-	int const	int const&
int const&&	int	-	int	int	-	int const	int const
int	int		int	int	-	-	int
int*	int*	int	int*	int*	int	-	int*
int&	int	-	int	-	-	int	int&
int&&	int	-	int	int	-	-	int

Template Argument Deduction

For all cases, when substituting template parameters, reference-collapsing rules apply

& + & → &
& + && → &
&& + & → &
&& + && → &&

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

What is the deduced type T?

Form of *ParameterType*

Type of *SomeExpression*

	T	T const*	T const&	T const&&	T*	T&	T&&
int const	int	-	int	int	-	-	int
int const*	int const*	int	int const*	int const*	int const	-	int const*
int const&	int	-	int	-	-	int const	int const&
int const&&	int	-	int	int	-	int const	int const
int	int		int	int	-	-	int
int*	int*	int	int*	int*	int	-	int*
int&	int	-	int	-	-	int	int&
int&&	int	-	int	int	-	-	int

Template Argument Deduction – Reference Collapsing

- You can't directly declare a reference to a reference

```
int const&   x  = 1;
int const& & rx = x;  //- Error: reference to reference is invalid
```

- But when composing types during template parameter substitution, it is allowed according to certain rules

& + & → &
& + && → &
&& + & → &
&& + && → &&

```
using RI = int&;

int        x   = 42;
RI         rx  = x;     //- rx is int&
RI const& rrx = rx;    //- "int& const&", drop outer const, & + & → &, rrx is int&

using RCI = int const&;
RCI&&      rcx = x;     //- "int const& const&&", no outer cv-qualifier, & + && → &,
                        //  rcx is int const&

using RRI = int&&
RRI const&&  rrcx = x; //- "int&& const&&", drop outer const, && + && → &&, rrcx is int&&
```

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

For all cases, when substituting template parameters, reference-collapsing rules apply
& + & → &
& + && → &
&& + & → &
&& + && → &&

Forwarding reference, a special matching rule applies for T&&: T&& deduces T

What is the deduced type T?

Form of *ParameterType*

Type of *SomeExpression*

	T	T const*	T const&	T const&&	T*	T&	T&&
int const	int	-	int	int	-	-	int
int const*	int const*	int	int const*	int const*	int const	-	int const*
int const&	int	-	int	-	-	int const	int const&
int const&&	int	-	int	int	-	int const	int const
int	int		int	int	-	-	int
int*	int*	int	int*	int*	int	-	int*
int&	int	-	int	-	-	int	int&
int&&	int	-	int	int	-	-	int

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

For all cases, when substituting template parameters, reference-collapsing rules apply
& + & → &
& + && → &
&& + & → &
&& + && → &&

Forwarding reference, a special matching rule applies for T&&: T&& deduces T

What is the deduced type T?

Form of *ParameterType*

Type of *SomeExpression*

	T	T const*	T const&	T const&&	T*	T&	T&&
int const	int	-	int	int	-	-	int
int const*	int const*	int	int const*	int const*	int const	-	int const*
int const&	int	-	int	-	-	int const	int const&
int const&&	int	-	int	int	-	int const	int const
int	int		int	int	-	-	int
int*	int*	int	int*	int*	int	-	int*
int&	int	-	int	-	-	int	int&
int&&	int	-	int	int	-	-	int

Template Argument Deduction

For all cases, when substituting template parameters, reference-collapsing rules apply

& + & → &

& + && → &

&& + & → &

&& + && → &&

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

What is the deduced type T?

Forwarding reference, a special matching rule applies for T&&: T&& deduces T

Form of *ParameterType*

Type of *SomeExpression*

	`T`	`T const*`	`T const&`	`T const&&`	`T*`	`T&`	`T&&`
`int const`	`int`	-	`int`	`int`	-	-	`int`
`int const*`	`int const*`	`int`	`int const*`	`int const*`	`int const`	-	`int const*`
`int const&`	`int`	-	`int`	-	-	`int const`	`int const&`
`int const&&`	`int`	-	`int`	`int`	-	`int const`	`int const`
`int`	`int`		`int`	`int`	-	-	`int`
`int*`	`int*`	`int`	`int*`	`int*`	`int`	-	`int*`
`int&`	`int`	-	`int`	-	-	`int`	`int&`
`int&&`	`int`	-	`int`	`int`	-	-	`int`

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

For all cases, when substituting template parameters, reference-collapsing rules apply
& + & → &
& + && → &
&& + & → &
&& + && → &&

Forwarding reference, a special matching rule applies for T&&: T&& deduces T

What is the deduced type T?

Form of *ParameterType*

Type of *SomeExpression*

	T	T const*	T const&	T const&&	T*	T&	T&&
int const	int	-	int	int	-	-	int
int const*	int const*	int	int const*	int const*	int const	-	int const*
int const&	int	-	int	-	-	int const	int const&
int const&&	int	-	int	int	-	int const	int const
int	int		int	int	-	-	int
int*	int*	int	int*	int*	int	-	int*
int&	int	-	int	-	-	int	int&
int&&	int	-	int	int	-	-	int

Template Argument Deduction

For all cases, when substituting template parameters, reference-collapsing rules apply
& + & → &
& + && → &
&& + & → &
&& + && → &&

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

What is the deduced type T?

Forwarding reference, a special matching rule applies for T&&: T&& deduces T

Form of *ParameterType*

Type of *SomeExpression*

	T	T const*	T const&	T const&&	T*	T&	T&&
int const	int	-	int	int	-	-	int
int const*	int const*	int	int const*	int const*	int const	-	int const*
int const&	int	-	int	-	-	int const	int const&
int const&&	int	-	int	int	-	int const	int const
int	int		int	int	-	-	int
int*	int*	int	int*	int*	int	-	int*
int&	int	-	int	-	-	int	int&
int&&	int	-	int	int	-	-	int

KEWB COMPUTING

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

For all cases, when substituting template parameters, reference-collapsing rules apply
& + & → &
& + && → &
&& + & → &
&& + && → &&

Forwarding reference, a special matching rule applies for T&&: T&& deduces T

What is the deduced type T?

Form of *ParameterType*

Type of *SomeExpression*

	T	T const*	T const&	T const&&	T*	T&	T&&
int const	int	-	int	int	-	-	int
int const*	int const*	int	int const*	int const*	int const	-	int const*
int const&	int	-	int	-	-	int const	int const&
int const&&	int	-	int	int	-	int const	int const
int	int		int	int	-	-	int
int*	int*	int	int*	int*	int	-	int*
int&	int	-	int	-	-	int	int&
int&&	int	-	int	int	-	-	int

Template Argument Deduction

```
template<class T>
void f(ParameterType t);

f(SomeExpression);
```

For all cases, when substituting template parameters, reference-collapsing rules apply
& + & → &
& + && → &
&& + & → &
&& + && → &&

Forwarding reference, a special matching rule applies for T&&: T&& deduces T

What is the deduced type T?

Form of *ParameterType*

Type of *SomeExpression*	T	T const*	T const&	T const&&	T*	T&	T&&
int const	int	-	int	int	-	-	int
int const*	int const*	int	int const*	int const*	int const	-	int const*
int const&	int	-	int	-	-	int const	int const&
int const&&	int	-	int	int	-	int const	int const
int	int		int	int	-	-	int
int*	int*	int	int*	int*	int	-	int*
int&	int	-	int	-	-	int	int&
int&&	int	-	int	int	-	-	int

Template Argument Deduction

- We've seen how individual arguments are deduced, but there's more to it; informally* the steps are
  - Explicitly-specified template arguments are fixed as such and don't participate in argument deduction
  - The remaining function arguments contribute to the deduction process for their respective template parameter
  - All deductions occur in parallel – the deduction for a given argument does not affect the deduction of any other argument
  - If a template argument couldn't be deduced, but has a default argument, then the default argument is fixed as the deduced argument
  - The compiler verifies that each argument that was not explicitly specified and was not fixed by a default argument has been deduced at least once, and that all deductions agree
  - Any function argument that participated in deduction must match its argument type exactly

*Paraphrased from Arthur O'Dwyer's
Template Normal Programming
CppCon 2016

Return Type Deduction

- The compiler can sometimes determine the return type
  - If the return type depends on template parameters

```
template<lass T>
T   min(T a, T b)
{
    return (b < a) ? b : a;
}

template<class T1, class T2>
auto  min(T1 a, T2 b)
{
    return (b < a) ? b : a;
}

template<class T1, class T2>
std::common_type_t<T1,T2>  min(T1 a, T2 b)
{
    return (b < a) ? b : a;
}
```

Function Template Explicit Specialization

- Function templates can be explicitly specialized
  - A user-provided implementation with all template parameters fully substituted

```
template<class T>
T const&  min(T const& a, T const& b)              //- Primary template
{
    return (a < b) ? a : b;
}

template<>
char const*  min(char const* pa, char const* pb)    //- Full specialization; this is only
{                                                   //  valid if a function template min
    return (strcmp(pa, pb) < 0) ? pa : pb;          //  has already been declared
}

char const*  p0 = "hello";
char const*  p1 = "world";
char const*  pr = min(p0, p1);   //- What's the answer?
```

More Detail – Class Templates

- A class template is a recipe for generating a parametrized family of classes
  - Let's write a simple stack adaptor with some special features in a header

```
//- header Stack.hpp

template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool      is_empty() const;
    T const&  top() const;

    void    push(T const& t);
    void    pop();
};
```

Class Templates – Template Parameter Scope

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
};
```

The scope of template parameter T begins here ...

... and ends here

Class Templates – Defining Member Functions

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t)
    {
        m_data.push_back(t);
    }
};
```

Class Templates – Defining Member Functions

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t)
};

template<class T> void
Stack<T>::push(T const& t)
{
    m_data.push_back(t);
}
```

The scope of template parameter T begins here ...

... and ends here

Class Templates – Class Scope

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Stack or Stack<T> ?

It can be either inside class scope

Class Templates – Class Scope

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool       is_empty() const;
    T const&   top() const;

    void     pop();
    void     push(T const& t);
    Stack    push_all_from(Stack const& other);
};
```

Class scope begins here ...

... and ends here

Class Templates – Class Scope

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other)
    {
        Stack tmp(*this);
        m_data.insert(m_data.end(), other.m_data.cbegin(), other.m_data.cend());
        return tmp;
    }
};
```

Class Templates – Class Scope

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);

};

template<class T> Stack<T>
Stack<T>::push_all_from(Stack const& other)
{
    Stack  tmp(*this);
    m_data.insert(m_data.end(), other.m_data.cbegin(), other.m_data.cend());
    return tmp;
}
```

Why `Stack<T>` here?

These are not in class scope

But not here?

Class Templates – Class Scope

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);

};

template<class T> Stack<T>
Stack<T>::push_all_from(Stack const& other)
{
    Stack  tmp(*this);
    m_data.insert(m_data.end(), other.begin(), other.end());
    return tmp;
}
```

Class scope begins here ...

... and ends here

Class Templates – Dependent Names

```
template<class T>
class Stack
{
    vector<T>  m_data;

  public:
    const_iterator  begin() const;
    const_iterator  end() const;
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Where does const_iterator come from?

From vector, but how?

Class Templates – Dependent Names

```
template<class T>
class Stack
{
    vector<T>  m

  public:
    using const_iterator = vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool       is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

```
error: missing 'typename' prior to dependent type name 'vector<T>::const_iterator'
using const_iterator = vector<T>::const_iterator;
```

`Stack::const_iterator` is a dependent name – it depends on one or more template parameters

Class Templates – Dependent Names

```
template<class T>
class Stack
{
    vector<T>  m_data;
  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool        is_empty() const;
    T const&    top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Class Templates – Dependent Names

```
template<class T>
class Stack
{
    vector<T>  m_data;
  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const
    {
        return m_data.crbegin();
    }
    const_iterator  end() const;
    bool          is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Class Templates – Dependent Names

```
template<class T>
class Stack
{
    vector<T>  m_data;
  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool        is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};

template<class T> typename Stack<T>::const_iterator
Stack<T>::begin() const
{
    return m_data.crbegin();
}
```

Outside class scope, `const_iterator` is a dependent name, and requires `typename`

Class Templates – Dependent Names

```
template<class T>
class Stack
{
    vector<T>  m_data;
  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool       is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};

template<class T> auto
Stack<T>::begin() const -> const_iterator
{
    return m_data.crbegin();
}
```

Inside class scope, `typename` is not required

Class Templates – Static Data Members

```
template<class T>
class Stack
{
    vector<T>  m_data;
    static int m_count;
  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool       is_empty() const;
    T const&   top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Class Templates – Static Data Members

```
template<class T>
class Stack
{
    vector<T>  m_data;
    static int m_count;
  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool        is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack c
};

template<class T>
int Stack<T>::m_count = 0;
```

Before C++17

Class Templates – Static Data Members

```
template<class T>
class Stack
{
    vector<T>         m_data;
    inline static int m_count = 0;
  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool        is_empty() const;
    T const&    top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Since C++17

Class Templates – Full Specialization

- Suppose we want to customize our stack specifically for `ints`

```
template<class T>
class Stack
{
    vector<T>         m_data;
    inline static int m_count = 0;

  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool      is_empty() const;
    T const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Class Templates – Full Specialization

```
template<>
class Stack<int>
{
    vector<int>  m_data;

  public:
    bool    is_empty() const;
    int     top() const;

    void    pop();
    void    push(int t);
    void    push_from(string const& s);
};
```

Class Templates – Full Specialization

```
template<>
class Stack<int>
{
    vector<int>  m_data;

  public:
    bool    is_empty() const;
    int     top() const;

    void    pop();
    void    push(int t);
    void    push_from(string const& s);
};


void
Stack<int>::push_from(string const& s)
{
    m_data.insert(m_data.end(), s.begin(), s.end());
}
```

Class Templates – Partial Specialization

- Suppose we want to customize our stack for pointers (any kind of pointer)

```
template<class T>
class Stack
{
    vector<T>         m_data;
    inline static int m_count = 0;

  public:
    using const_iterator = typename vector<T>::const_reverse_iterator;

    const_iterator  begin() const;
    const_iterator  end() const;
    bool       is_empty() const;
    T const&   top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Class Templates – Partial Specialization

```
template<class T>
class Stack<T*>
{
    vector<T*>  m_data;

  public:
    bool    is_empty() const;
    T*      top() const;

    T*      pop();
    void    push(T const& t);
};
...

Stack<string*>   pstack;

pstack.push(new string("Hello"));
cout << *pstack.top() << '\n';

delete pstack.pop();
```

Class Templates – Partial Specialization

```
template<class T>
class Stack<T*>
{
    vector<T*>  m_data;

  public:
    bool    is_empty() const;
    T*      top() const;

    T*      pop();
    void    push(T const& t);
};

template<class T>
T*
Stack<T*>::pop()
{
    T*  tmp = m_data.back();
    m_data.pop_back();
    return tmp;
}
```

Class Templates – Partial Specialization

```
template<class T, class U>
struct Pair
{
    T   first;
    U   second;
    Pair(T const& t, U const& u) {...};
    ...
};

template<class T, class U>
struct Pair<T&, U>
{
    T   first;
    U   second;
    Pair(T const& t, U const& u) {...};
    ...
};

template<class T, class U>
struct Pair<T, U&>  { ... };

template<class T, class U>
struct Pair<T&, U&> { ... };
```

```
Pair<int, float>    p0;

Pair<int&, float>   p1;

Pair<int, float&>   p2;

Pair<int&, float&>  p3;


template<class T, class U>
void f(T t, U u)
{
    Pair<T, U>      p(t, u);
    ...
}
```

Class Templates – Type Traits

- What else can we do with partial specialization?
    - We can detect properties of types as predicates

```
template<class T>
struct IsPointer
{
    static constexpr bool value = false;
};

template<class T>
struct IsPointer<T*>
{
    static constexpr bool value = true;
};

template<class T>
inline constexpr
bool  IsPointer_V = IsPointer<T>::value;
```

```
template<class T>
void foo(T t)
{
    if constexpr (IsPointer<T>::value)
        //- Do one thing
    else
        //- Do another thing


    if constexpr (IsPointer_V<T>)
        //- Do one thing
    else
        //- Do another thing

    ...
}
```

Class Templates – Type Traits

- What else can we do with partial specialization?
  - We can detect and normalize properties of types

```
template<class T>
struct RemoveCV
{
    using Type = T;
};

template<class T>
struct RemoveCV<T const>
{
    using Type = T;
};
```

```
template<class T>
struct RemoveCV<T volatile>
{
    using Type = T;
};

template<class T>
struct RemoveCV<T const volatile>
{
    using Type = T;
};

template<class T>
using RemoveCV_T = typename RemoveCV<T>::Type;
```

Class Templates – Type Traits

- What else can we do with partial specialization?
  - We can detect and normalize properties of types

```
template<class T>
struct RemoveRef
{
    using Type = T;
};

template<class T>
struct RemoveRef<T&>
{
    using Type = T;
};
```

```
template<class T>
struct RemoveRef<T&&>
{
    using Type = T;
};


template<class T>
using RemoveRef_T = typename RemoveRef<T>::Type;
```

Class Templates – Type Traits

- What else can we do with partial specialization?
  - We can detect and normalize properties of types

```
#include <type_traits>
using std::is_same_v;

static_assert(is_same_v<int, RemoveCV_T<int>>);
static_assert(is_same_v<int, RemoveCV_T<int const>>);
static_assert(is_same_v<int, RemoveCV_T<int volatile>>);
static_assert(is_same_v<int, RemoveCV_T<int const volatile>>);

static_assert(is_same_v<int const, RemoveRef_T<int const>>);
static_assert(is_same_v<int const, RemoveRef_T<int const&>>);
static_assert(is_same_v<int const, RemoveRef_T<int const&&>>);

static_assert(is_same_v<int, RemoveCV_T<RemoveRef_T<int const&>>>);
```

Class Templates – Type Traits

- We can use traits to (possible) improve our Stack

```
template<class T>
class Stack
{
    using DataType = RemoveCV_T<RemoveRef_T<T>>;
    vector<DataType>  m_data;

  public:
    using const_iterator = typename vector<DataType>::const_reverse_iterator;
    const_iterator  begin() const;
    const_iterator  end() const;

    bool             is_empty() const;
    DataType const&  top() const;

    void    pop();
    void    push(T const& t);
    Stack   push_all_from(Stack const& other);
};
```

Class Templates – Friends

- Class templates can have friends

```
template<class T>
class Stack
{
    ...

    template<class U> friend class Stack<U>;
    friend class FooBar;
    friend T;                  //- Ignored if T is not a class type;

    friend void foo();
    friend void bar<T>();

  public:
    template<class U>
    Stack(Stack<U> const& src);
    ...
};
```

Summary

- C++ templates are a vast topic

- Recommendations
  - Start simple, with the topics in this talk
  - Understand value categories and references
  - Understand function templates, especially type deduction
  - Understand overload resolution and how it applies to function templates
  - Understand class templates, especially partial specialization
  - Write some type traits, try some metaprogramming
  - Read *Vandevoorde, Josuttis, and Gregor*
  - Watch any talk about templates by Walter Brown
  - Use cppreference.com
  - Ask questions and don't give up!

Thank You for Attending!

Talk: `github.com/BobSteagall/CppCon2021`

Blog: `bobsteagall.com`