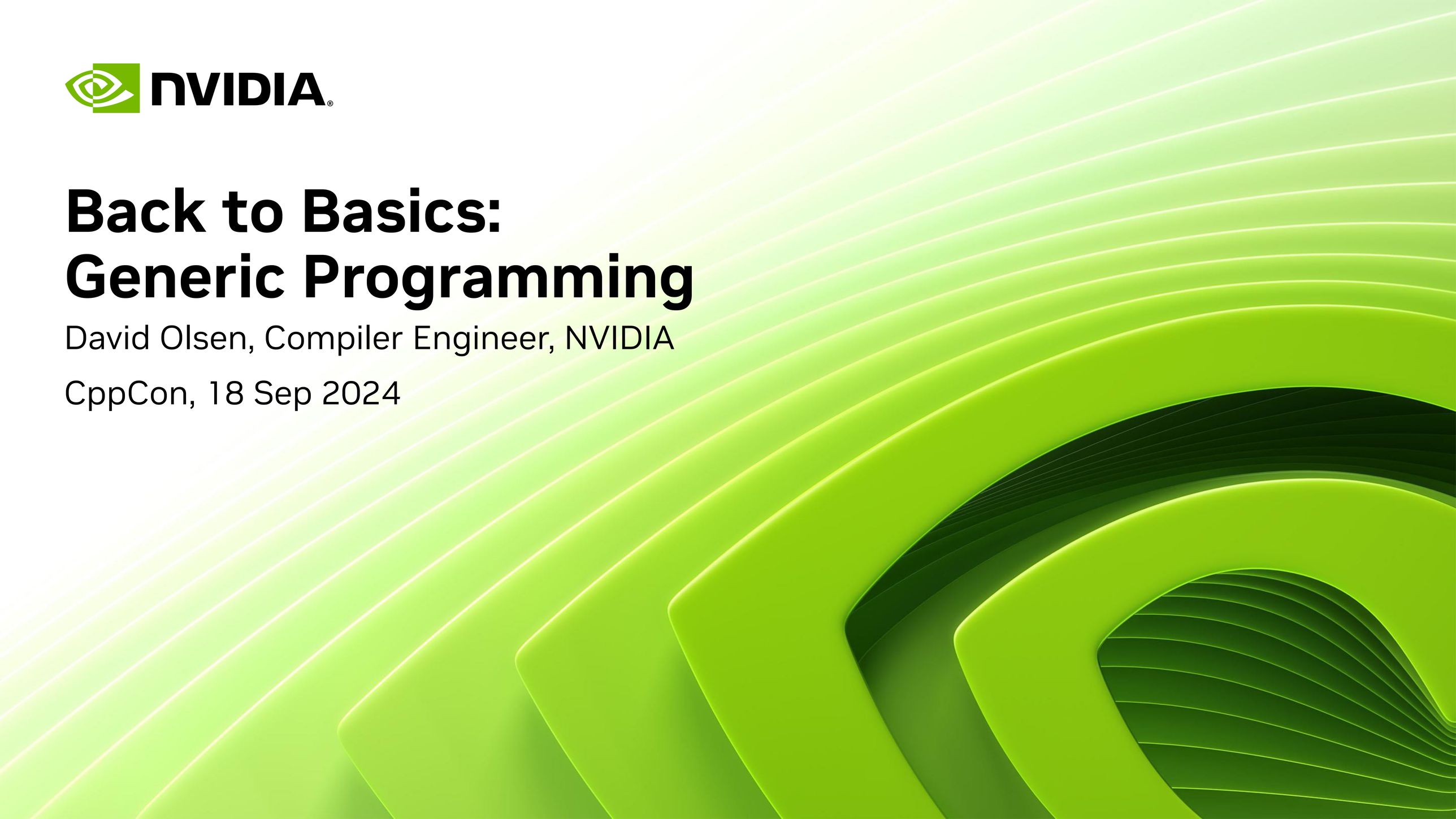
Back To Basics
Generic Programming

DAVID OLSEN



Back to Basics: Generic Programming

David Olsen, Compiler Engineer, NVIDIA

CppCon, 18 Sep 2024

Generic Programming

Generic Programming

Generic Programming

Same code works on different, unrelated types

Generic Programming

Same code works on different, unrelated types

```
T sum(C container) {
    T result = 0;
    for (T value : container) {
        result += value;
    }
    return result;
}
```

Generic Programming

Static polymorphism

Generic Programming

Same code works on different, unrelated types

Static polymorphism

C++ Templates

Same code works on different, unrelated types

Static polymorphism

Basics

Define a Template

`template <`*`template-parameters`***`>`** *declaration*`;`

declaration can be

- class / struct
- function
- type alias
- variable
- concept

template-parameter is

`class` | **`typename`** *identifier* [`=` *default-value*]

Template definition should be in a header file

Class Template Definition

```
template <class T, class U>
class pair {
    T m0;
    U m1;
  public:
    pair() { }
    pair(T v0, U v1) : m0(v0), m1(v1) { }
    T first() const { return m0; }
    U second() const { return m1; }
};
```

https://godbolt.org/z/rejh9YPhK

Class Template Definition

```
template <class T, class U>
class pair {
    T m0;
    U m1;
  public:
    pair() { }
    pair(T v0, U v1) : m0(v0), m1(v1) { }
    T first() const { return m0; }
    U second() const { return m1; }
};
```

https://godbolt.org/z/rejh9YPhK

Class Template Definition

```
template <class T, class U>
class pair {
    T m0;
    U m1;
  public:
    pair() { }
    pair(T v0, U v1) : m0(v0), m1(v1) { }
    T first() const { return m0; }
    U second() const { return m1; }
};
```

https://godbolt.org/z/rejh9YPhK

Class Template Definition

```
template <class T, class U>
class pair {
    T m0;
    U m1;
  public:
    pair() { }
    pair(T v0, U v1) : m0(v0), m1(v1) { }
    T first() const { return m0; }
    U second() const { return m1; }
};
```

https://godbolt.org/z/rejh9YPhK

Class Template Definition

```
template <class T, class U>
class pair {
    T m0;
    U m1;
  public:
    pair() { }
    pair(T v0, U v1) : m0(v0), m1(v1) { }
    T first() const { return m0; }
    U second() const { return m1; }
};
```

https://godbolt.org/z/rejh9YPhK

Class Template Definition

```
template <typename T, typename U>
class pair {
    T m0;
    U m1;
  public:
    pair() { }
    pair(T v0, U v1) : m0(v0), m1(v1) { }
    T first() const { return m0; }
    U second() const { return m1; }
};
```

https://godbolt.org/z/rejh9YPhK

Function Template Definition

```
template <class T>
void swap_pointed_to(T* a, T* b) {
    T temp = *a;
    *a = *b;
    *b = temp;
}
```

https://godbolt.org/z/3Kn6exqf5

Function Template Definition

```
template <class T>
void swap_pointed_to(T* a, T* b) {
    T temp = *a;
    *a = *b;
    *b = temp;
}
```

https://godbolt.org/z/3Kn6exqf5

Function Template Definition

```
template <class T>
void swap_pointed_to(T* a, T* b) {
    T temp = *a;
    *a = *b;
    *b = temp;
}
```

https://godbolt.org/z/3Kn6exqf5

Type Alias Template Definition

```
template <class T> using ptr = T*;

template <class Iter1, class Iter2>
using result_type = typename std::common_type<
    typename std::iterator_traits<Iter1>::value_type,
    typename std::iterator_traits<Iter2>::value_type
  >::type;
```

https://godbolt.org/z/EE9zqYEEe

Variable Template Definition

```
template <class T>
constexpr bool is_big_and_trivial =
    sizeof(T) > 16 &&
    std::is_trivially_copyable<T>::value &&
    std::is_trivially_destructible<T>::value;
```

https://godbolt.org/z/qaEc8z3bn

Template Parameters

Three kinds

Type template parameter

`class`|**`typename`** *identifier*

Non-type template parameter (NTTP)

type|**`auto`** *identifier*

Template template parameter

`template <`*template-parameters*`> class`**|**`typename`** *identifier*

https://godbolt.org/z/zz47Ee89a

Template Parameters

Three kinds

Type template parameter

`class`|**`typename`** *identifier*

Non-type template parameter (NTTP)

type|**`auto`** *identifier*

Template template parameter

`template <`*template-parameters***`> class`**|**`typename`** *identifier*

https://godbolt.org/z/zz47Ee89a

Template Parameters

Three kinds

Type template parameter

`class`|**`typename`** *identifier*

Non-type template parameter (NTTP)

type|**`auto`** *identifier*

Template template parameter

`template <`*template-parameters***`> class`**|**`typename`** *identifier*

https://godbolt.org/z/zz47Ee89a

Template Parameters

Three kinds

Type template parameter

`class`|**`typename`** *identifier*

Non-type template parameter (NTTP)

type|**`auto`** *identifier*

Template template parameter

`template <`*template-parameters*`> class`**|**`typename`** *identifier*

https://godbolt.org/z/zz47Ee89a

Template Parameters

Name is optional

Type template parameter

`class`|**`typename`** [*identifier*]

Non-type template parameter (NTTP)

type|**`auto`** [*identifier*]

Template template parameter

`template <`*template-parameters*`> class`**|**`typename`** [*identifier*]

https://godbolt.org/z/zz47Ee89a

Template Parameters

Default value is optional

Type template parameter

`class`|**`typename`** [*identifier*] [**`=`** *default-value*]

Non-type template parameter (NTTP)

type|**`auto`** [*identifier*] [**`=`** *default-value*]

Template template parameter

`template <`*template-parameters***`> class`**|**`typename`** [*identifier*]
[**`=`** *default-value*]

https://godbolt.org/z/zz47Ee89a

Template Parameters

Variadic

Type template parameter

`class`|**`typename ...`** [*identifier*]

Non-type template parameter (NTTP)

type|**`auto ...`** [*identifier*]

Template template parameter

`template <`*template-parameters*`> class`**|**`typename ...`** [*identifier*]

Non-Type Template Parameters

Example

```
template <class T, std::size_t N>
class array {
    T m_data[N];
  public:
    constexpr std::size_t size() const { return N; }
    // ...
};
```

https://godbolt.org/z/YWjbM3a8o

Non-Type Template Parameters

Example

```
template <class T, std::size_t N>
class array {
    T m_data[N];
  public:
    constexpr std::size_t size() const { return N; }
    // ...
};
```

https://godbolt.org/z/YWjbM3a8o

Non-Type Template Parameters

Example

```
template <class T, std::size_t N>
class array {
    T m_data[N];
  public:
    constexpr std::size_t size() const { return N; }
    // ...
};
```

https://godbolt.org/z/YWjbM3a8o

Class Template Member Functions

In-class definition

```
template <class T> struct A {
    T f() const {
        return T{};
    }
    template <class U> T g(U u) const {
        return static_cast<T>(u);
    }
};
```

https://godbolt.org/z/n69rGKE93

Class Template Member Functions

Out-of-class definition

```
template <class T> struct A {
    T f() const;
    template <class U> T g(U u) const;
};
template <class T> T A<T>::f() const {
    return T{};
}
template <class T> template <class U>
T A<T>::g(U u) const {
    return static_cast<T>(u);
}
```

https://godbolt.org/z/n69rGKE93

Class Template Member Functions

In-class definition

```
template <class T> struct A {
    T f() const {
        return T{};
    }
    template <class U> T g(U u) const {
        return static_cast<T>(u);
    }
};
```

https://godbolt.org/z/n69rGKE93

Using a template

template-name < *template-arguments* >

Template argument kind must match template parameter kind

- type
- compile-time constant of correct type
- template name

Using a template

template-name **<** *template-arguments* **>**

```
pair<int, std::string> id_name = { 123, "Joe" };

swap_pointed_to<int>(p, q);

result_type<decltype(it), int*> result = ...;

static_assert(not is_big_and_trivial<int>);
```

Using a template

template-name < *template-arguments* >

```
pair<int, std::string> id_name = { 123, "Joe" };

swap_pointed_to<int>(p, q);

result_type<decltype(it), int*> result = ...;

static_assert(not is_big_and_trivial<int>);
```

Using a template

template-name **<** *template-arguments* **>**

```
pair<int, std::string> id_name = { 123, "Joe"s };

swap_pointed_to<int>(p, q);

result_type<decltype(it), int*> result = ...;

static_assert(not is_big_and_trivial<int>);
```

Using a template

template-name **<** *template-arguments* **>**

```
pair<int, std::string> id_name = { 123, "Joe"s };

swap_pointed_to<int>(p, q);

result_type<decltype(it), int*> result = ...;

static_assert(not is_big_and_trivial<int>);
```

Using a template

template-name **<** *template-arguments* **>**

```
pair id_name = { 123, "Joe"s };

swap_pointed_to(p, q);

result_type<decltype(it), int*> result = ...;

static_assert(not is_big_and_trivial<int>);
```

Substitution & Instantiation

Substitution vs. Instantiation

Substitution

- Substitute template arguments for template parameters
- Results in class declaration or function declaration
- Checks the correctness of the template arguments

Instantiation

- Full definition of the class or function or type alias or variable
- Happens after substitution, only when full definition is needed
- Checks the correctness of the definition

Substitution vs. Instantiation

Substitution

- Substitute template arguments for template parameters
- Results in class declaration or function declaration
- Checks the correctness of the template arguments

Instantiation

- Full definition of the class or function or type alias or variable
- Happens after substitution, only when full definition is needed
- Checks the correctness of the definition

Substitution vs. Instantiation

Substitution

- Substitute template arguments for template parameters
- Results in class declaration or function declaration
- Checks the correctness of the template arguments

Instantiation

- Full definition of the class or function or type alias or variable
- Happens after substitution, only when full definition is needed
- Checks the correctness of the definition

Substitution

Class templates

Substitution without instantiation in two contexts

- Incomplete type is sufficient
- Class template partial specialization resolution

Results in an incomplete class type

- Contents of the class are not checked
- Only the template arguments are checked

Substitution

Class templates

Substitution without instantiation in two contexts

- Incomplete type is sufficient `A<int, std::string>* ap;`
- Class template partial specialization resolution

Results in an incomplete class type

- Contents of the class are not checked
- Only the template arguments are checked

Substitution

Function templates

- Substitution happens during overload resolution
  - Unselected overloads are not instantiated
- Results in function declaration
  - Only function signature is checked
    - Including parameters, return type, `noexcept` clause
  - Function body is not checked

Instantiation

Class templates

Replace template parameters with template arguments in the class definition

Results in a complete class definition

Member functions are not instantiated until they are used

Instantiation

Class template example

```
template <class T, class U>
class pair {
    T m0;
    U m1;
  public:
    pair() { }
    pair(T v0, U v1) : m0(v0), m1(v1) { }
    T first() const { return m0; }
    U second() const { return m1; }
};
```

Instantiation

```
pair<int, data>
```

```
pair<int, data>
```

Instantiation

pair<int, data>

```
          pair<int, data>
class pairIi4dataE {
    int m0;
    data m1;
  public:
    pairIi4dataE();
    pairIi4dataE(int v0, data v1);
    int first() const;
    data second() const;
};
```

Instantiation

pair<int, data>

```
          pair<int, data>
class pairIi4dataE {
    int m0;
    data m1;
  public:
    pairIi4dataE();
    pairIi4dataE(int v0, data v1);
    int first() const;
    data second() const;
};
```

Compiler's internal name for pair<int, data>

Instantiation

pair<int, data>

```
          pair<int, data>
class pairIi4dataE {
    int m0;
    data m1;
  public:
    pairIi4dataE();
    pairIi4dataE(int v0, data v1);
    int first() const;
    data second() const;
};
```

Instantiation

```
pair<int[10], data>
```

```
pair<int[10], data>
```

Instantiation

pair<int[10], data>

```
          pair<int[10], data>
class pairIA10_i4dataE {
    int m0[10];
    data m1;
  public:
    pairIA10_i4dataE();
    pairIA10_i4dataE(int v0[10], data v1);
    int (first())[10] const;
    data second() const;
};
```

Instantiation

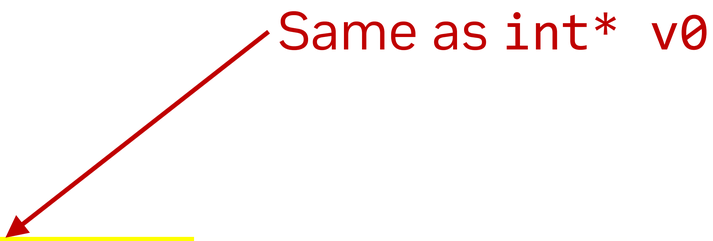
pair<int[10], data>

```
          pair<int[10], data>
class pairIA10_i4dataE {
    int m0[10];
    data m1;
  public:
    pairIA10_i4dataE();
    pairIA10_i4dataE(int v0[10], data v1);
    int (first())[10] const;
    data second() const;
};
```

Instantiation

pair<int[10], data>

```
          pair<int[10], data>
class pairIA10_i4dataE {
    int m0[10];
    data m1;
  public:
    pairIA10_i4dataE();
    pairIA10_i4dataE(int v0[10], data v1);
    int (first())[10] const;
    data second() const;
};
```



Instantiation

pair<int[10], data>

```
          pair<int[10], data>
class pairIA10_i4dataE {
    int m0[10];
    data m1;
  public:
    pairIA10_i4dataE();
    pairIA10_i4dataE(int v0[10], data v1);
    int (first())[10] const;
    data second() const;
};
```

Instantiation

pair<int[10], data>

```
          pair<int[10], data>
class pairIA10_i4dataE {
    int m0[10];
    data m1;
  public:
    pairIA10_i4dataE();
    pairIA10_i4dataE(int v0[10], data v1);
    int (first())[10] const;
    data second() const;
};
```

```
<source>: In instantiation of 'class pair<int [10], data>':
<source>:8:7: error: function returning an array
    8 |     T first() const { return m0; }
      |       ^~~~~
```

Instantiation

Function template

Replace template parameters with template arguments in the function definition

Results in a complete function definition

Instantiation

Function template example

```
template <class T>
void swap_pointed_to(T* a, T* b) {
    T temp = *a;
    *a = *b;
    *b = temp;
}
```

Instantiation

swap_pointed_to<double>

```
          swap_pointed_to<double>
void swap_pointed_toIdEvPT_S1_(double* a, double* b) {
    double temp = *a;
    *a = *b;
    *b = temp;
}
```

SFINAE

Substitution Failure Is Not An Error

A failure during substitution does not fail compilation

Instead, the candidate is discarded

- A function overload that fails substitution is not a viable candidate

This feature is necessary for function templates and class template partial specializations to be useful

SFINAE

Substitution Failure Is Not An Error

A failure during substitution does not fail compilation

Instead, the candidate is discarded

- A function overload that fails substitution is not a viable candidate

This feature is necessary for function templates and class template partial specializations to be useful

Using Class Templates

Class Template Instantiation is a Type

class-template-name < *template-arguments* >

Results in a regular type

Each instantiation is a distinct and unrelated type

Instantiations are Unrelated Types

```
struct A { };
struct B { };

A a;

B b = a;

B* bp = &a;
```

https://godbolt.org/z/PvrTssMeP

Instantiations are Unrelated Types

```
struct A { };
struct B { };

A a;

B b = a;
  error: conversion from 'A' to non-scalar type 'B' requested
B* bp = &a;
  error: cannot convert 'A*' to 'B*' in initialization
```

https://godbolt.org/z/PvrTssMeP

Instantiations are Unrelated Types

```
template <class T> struct D { };

D<int> di;

D<double> dd = di;

D<double>* ddp = &di;
```

https://godbolt.org/z/PvrTssMeP

Instantiations are Unrelated Types

```
template <class T> struct D { };

D<int> di;

D<double> dd = di;
```

```
   error: conversion from 'D<int>' to non-scalar type 'D<double>' requested
```

```
D<double>* ddp = &di;
```

```
   error: cannot convert 'D<int>*' to 'D<double>*' in initialization
```

https://godbolt.org/z/PvrTssMeP

Instantiations are Unrelated Types

```
template <class T> struct D { };

D<int> di;

D<const int> dci = di;

D<const int>* dcip = &di;
```

https://godbolt.org/z/PvrTssMeP

Instantiations are Unrelated Types

```
template <class T> struct D { };

D<int> di;

D<const int> dci = di;
```

```
   error: conversion from 'D<int>' to non-scalar type 'D<const int>' requested
```

```
D<const int>* dcip = &di;
```

```
   error: cannot convert 'D<int>*' to 'D<const int>*' in initialization
```

https://godbolt.org/z/PvrTssMeP

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T, std::size_t N> class std::array;
```

https://godbolt.org/z/9Wh9Pr4cP

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T, std::size_t N> class std::array;

std::array<int, 10> a;
```

https://godbolt.org/z/9Wh9Pr4cP

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T, std::size_t N> class std::array;
```

```
std::array<int, double> b;
```

```
<source>:5:27: error: type/value mismatch at argument 2 in template parameter list for
'template<class _Tp, long unsigned int _Nm> struct std::array’
    5 |     std::array<int, double> b;
      |                           ^
<source>:5:27: note:   expected a constant of type 'long unsigned int', got 'double'
```

https://godbolt.org/z/9Wh9Pr4cP

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T, std::size_t N> class std::array;
```

```
std::array<int, double> b;
```

```
<source>:5:27: error: type/value mismatch at argument 2 in template parameter list for
'template<class _Tp, long unsigned int _Nm> struct std::array’
    5 |     std::array<int, double> b;
      |                           ^
<source>:5:27: note:   expected a constant of type 'long unsigned int', got 'double'
```

https://godbolt.org/z/9Wh9Pr4cP

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T, std::size_t N> class std::array;
```

```
std::array<1, 2> c;
```

```
<source>:6:20: error: type/value mismatch at argument 1 in template parameter list for
'template<class _Tp, long unsigned int _Nm> struct std::array’
    6 |     std::array<1, 2> c;
      |                    ^
<source>:6:20: note:   expected a type, got '1'
```

https://godbolt.org/z/9Wh9Pr4cP

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T, std::size_t N> class std::array;
```

```
std::array<1, 2> c;
```

```
<source>:6:20: error: type/value mismatch at argument 1 in template parameter list for
'template<class _Tp, long unsigned int _Nm> struct std::array’
    6 |     std::array<1, 2> c;
      |                    ^
<source>:6:20: note:   expected a type, got '1'
```

https://godbolt.org/z/9Wh9Pr4cP

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T> struct A { };
template <int N> struct B { };
template <template <class> class X> struct C { };

C<int> wrong_kind;
```

```
<source>:7:6: error: type/value mismatch at argument 1 in template
parameter list for 'template<template<class> class X> struct C’
    7 | C<int> wrong_kind;
      |      ^
<source>:7:6: note:   expected a class template, got 'int'
```

https://godbolt.org/z/eoMY9Y8EE

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T> struct A { };
template <int N> struct B { };
template <template <class> class X> struct C { };

C<int> wrong_kind;
```

```
<source>:7:6: error: type/value mismatch at argument 1 in template
parameter list for 'template<template<class> class X> struct C’
    7 | C<int> wrong_kind;
      |      ^
<source>:7:6: note:   expected a class template, got 'int'
```

https://godbolt.org/z/eoMY9Y8EE

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T> struct A { };
template <int N> struct B { };
template <template <class> class X> struct C { };

C<int> wrong_kind;
```

```
<source>:7:6: error: type/value mismatch at argument 1 in template
parameter list for 'template<template<class> class X> struct C’
    7 | C<int> wrong_kind;
      |      ^
<source>:7:6: note:   expected a class template, got 'int'
```

https://godbolt.org/z/eoMY9Y8EE

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T> struct A { };
template <int N> struct B { };
template <template <class> class X> struct C { };

C<int> wrong_kind;
```

```
<source>:7:6: error: type/value mismatch at argument 1 in template
parameter list for 'template<template<class> class X> struct C’
    7 | C<int> wrong_kind;
      |      ^
<source>:7:6: note:   expected a class template, got 'int'
```

https://godbolt.org/z/eoMY9Y8EE

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T> struct A { };
template <int N> struct B { };
template <template <class> class X> struct C { };

C<A> match;
```

https://godbolt.org/z/eoMY9Y8EE

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T> struct A { };
template <int N> struct B { };
template <template <class> class X> struct C { };

C<B> no_match;
```

```
<source>:7:4: error: type/value mismatch at argument 1 in template
parameter list for 'template<template<class> class X> struct C’
    7 | C<B> no_match;
      |    ^
<source>:7:4: note:   expected a template of type 'template<class> class
X', got 'template<int N> struct B'
```

https://godbolt.org/z/eoMY9Y8EE

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T> struct A { };
template <int N> struct B { };
template <template <class> class X> struct C { };

C<B> no_match;
```

```
<source>:7:4: error: type/value mismatch at argument 1 in template
parameter list for 'template<template<class> class X> struct C’
    7 | C<B> no_match;
      |    ^
<source>:7:4: note:   expected a template of type 'template<class> class
X', got 'template<int N> struct B'
```

https://godbolt.org/z/eoMY9Y8EE

Class Template Argument Kinds

Template argument kind must match template parameter kind

```
template <class T> struct A { };
template <int N> struct B { };
template <template <class> class X> struct C { };

C<B> no_match;

<source>:7:4: error: type/value mismatch at argument 1 in template
parameter list for 'template<template<class> class X> struct C’
    7 | C<B> no_match;
      |    ^
<source>:7:4: note:   expected a template of type 'template<class> class
X', got 'template<int N> struct B'
```

https://godbolt.org/z/eoMY9Y8EE

Using Function Template

No Function Template Arguments

Use function template like a regular function

Let the compiler deduce the arguments

- Unless the function's API requires explicit template arguments

No Function Template Arguments

```
template< class ExecutionPolicy, class ForwardIt,
          class T, class BinaryOp, class UnaryOp >
T transform_reduce( ExecutionPolicy&& policy,
                    ForwardIt first, ForwardIt last,
                    T init, BinaryOp reduce,
                    UnaryOp transform);
```

No Function Template Arguments

```
route_cost find_best_route(int const* distances, int N) {
  return std::transform_reduce(std::execution::par,
      counting_iterator<long>(0L),
      counting_iterator<long>(factorial(N)),
      route_cost(),
      route_cost::min,
      [=](long i) {
        int cost = 0;
        // ... calculate cost ...
        return route_cost(i, cost);
      });
}
```

Constraints

Constraints

Requirements on a template argument

Checked during substitution, not instantiation

Often make use of concepts and requires clauses

Constraints

Examples

```
template <class T>
  int count_one_bits(T arg);
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
template <class T>
  typename std::enable_if<
      std::is_integral<T>::value, int>::type
  count_one_bits(T arg);
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
template <class T,
          class = typename std::enable_if<
              std::is_integral<T>::value>::type>
  int count_one_bits(T arg);
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
template <class T> requires std::is_integral<T>::value
  int count_one_bits(T arg);
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
template <class T> requires std::integral<T>
  int count_one_bits(T arg);
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
template <std::integral T>
  int count_one_bits(T arg);
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
template <class T>
  int count_one_bits(T arg)
    requires std::is_integral<T>::value;
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
template <class T>
  int count_one_bits(T arg)
    requires std::integral<T>;
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
int count_one_bits(std::integral auto arg);
```

https://godbolt.org/z/Kr14fvMYz

Constraints

Examples

```
template <class T> requires (type_is_integral(^^T))
  int count_one_bits(T arg);
```

https://godbolt.org/z/Kr14fvMYz

Constraints

to learn more

"*Back to Basics: Concepts*"

Nicolai Josuttis, today at 14:00 in room Maple 3/4/5

Learn how constraints work

Then use them

Writing Class Templates

KISS Principle

Keep It Simple and Straightforward

No fancy template metaprogramming or type-based metafunctions

Make it easy for your users

Document Requirements

Document expectations for the template parameters

In code if possible

- via constraints

In documentation otherwise

Member functions can have additional requirements

Specialization

Sometimes one instantiation of a class template should behave differently than the others

Define a class to be used in place of the normal instantiation

Specialization can have a completely different interface

- But that is usually a bad idea

Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};
```

https://godbolt.org/z/r6E7Wh675

Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};

template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};
```

```
template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};
```

```
template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};

template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};

template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Sometimes you want to specialize for one template parameter, but not for all of them

Or specialize when one template parameter fits a pattern

Similar to full specialization, but template parameter list is not empty

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};
template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
template <class T>
struct safe_sizeof<T[]> {
    static constexpr std::size_t value = 0;
};
template <class R, class... Args>
struct safe_sizeof<R(Args...)> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};
template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
template <class T>
struct safe_sizeof<T[]> {
    static constexpr std::size_t value = 0;
};
template <class R, class... Args>
struct safe_sizeof<R(Args...)> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};
template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
template <class T>
struct safe_sizeof<T[]> {
    static constexpr std::size_t value = 0;
};
template <class R, class... Args>
struct safe_sizeof<R(Args...)> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};
template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
template <class T>
struct safe_sizeof<T[]> {
    static constexpr std::size_t value = 0;
};
template <class R, class... Args>
struct safe_sizeof<R(Args...)> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = sizeof(T);
};
template <>
struct safe_sizeof<void> {
    static constexpr std::size_t value = 0;
};
template <class T>
struct safe_sizeof<T[]> {
    static constexpr std::size_t value = 0;
};
template <class R, class... Args>
struct safe_sizeof<R(Args...)> {
    static constexpr std::size_t value = 0;
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = 0;
};

template <class T> requires (sizeof(T) > 0)
struct safe_sizeof<T> {
    static constexpr std::size_t value = sizeof(T);
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = 0;
};

template <class T> requires (sizeof(T) > 0)
struct safe_sizeof<T> {
    static constexpr std::size_t value = sizeof(T);
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = 0;
};

template <class T> requires (sizeof(T) > 0)
struct safe_sizeof<T> {
    static constexpr std::size_t value = sizeof(T);
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = 0;
};

template <class T> requires (sizeof(T) > 0)
struct safe_sizeof<T> {
    static constexpr std::size_t value = sizeof(T);
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
struct safe_sizeof {
    static constexpr std::size_t value = 0;
};

template <class T> requires (sizeof(T) > 0)
struct safe_sizeof<T> {
    static constexpr std::size_t value = sizeof(T);
};
```

https://godbolt.org/z/r6E7Wh675

Partial Specialization

Example

```
template <class T>
constexpr std::size_t safe_sizeof = 0;

template <class T> requires (sizeof(T) > 0)
constexpr std::size_t safe_sizeof<T> = sizeof(T);
```

https://godbolt.org/z/sPs8q6Mhd

Specialization Allowed

Class template:	Yes
Variable template:	Yes
Type alias template:	No
Concept:	No
Function template:	*see next section*

Type Alias Specialization Workaround

```
template <class T>
struct remove_pointer {
    using type = T;
};
template <class T>
struct remove_pointer<T*> {
    using type = T;
};
template <class T>
using remove_pointer_t =
    typename remove_pointer<T>::type;
```

https://godbolt.org/z/79d5hqMKz

Type Alias Specialization Workaround

```
template <class T>
struct remove_pointer {
    using type = T;
};
template <class T>
struct remove_pointer<T*> {
    using type = T;
};
template <class T>
using remove_pointer_t =
    typename remove_pointer<T>::type;
```

https://godbolt.org/z/79d5hqMKz

Type Alias Specialization Workaround

```
template <class T>
struct remove_pointer {
    using type = T;
};
template <class T>
struct remove_pointer<T*> {
    using type = T;
};
template <class T>
using remove_pointer_t =
    typename remove_pointer<T>::type;
```

https://godbolt.org/z/79d5hqMKz

Type Alias Specialization Workaround

```
template <class T>
struct remove_pointer {
    using type = T;
};
template <class T>
struct remove_pointer<T*> {
    using type = T;
};
template <class T>
using remove_pointer_t =
    typename remove_pointer<T>::type;
```

https://godbolt.org/z/79d5hqMKz

`typename`

Compiler needs help parsing template definition

Keyword `typename` must precede any qualified type name that depends on a template parameter

typename

Compiler needs help parsing template definition

Keyword typename must precede any qualified type name that depends on a template parameter

```
template <class T>
using remove_pointer_t =
    typename remove_pointer<T>::type;
```

`typename`

Compiler needs help parsing template definition

Keyword `typename` must precede any qualified type name that depends on a template parameter

typename

Compiler needs help parsing template definition

Keyword `typename` must precede any qualified type name that depends on a template parameter

```
A * B(C(D));
```

typename

Compiler needs help parsing template definition

Keyword `typename` must precede any qualified type name that depends on a template parameter

```
A * B(C(D));
```

Expression statement:

multiply `A` and `B(C(D))`

typename

Compiler needs help parsing template definition

Keyword `typename` must precede any qualified type name that depends on a template parameter

```
A * B(C(D));
```

Variable definition:

`B` is a variable of type pointer-to-`A` with initial value `C(D)`

typename

Compiler needs help parsing template definition

Keyword `typename` must precede any qualified type name that depends on a template parameter

```
A * B(C(D));
```

Variable definition:

`B` is a variable of type pointer-to-`A` with initial value `C(D)`

```
A* B = C(D);
```

typename

Compiler needs help parsing template definition

Keyword `typename` must precede any qualified type name that depends on a template parameter

```
A * B(C(D));
```

Function declaration:

> `B` is a function with parameter pointer-to-(function with parameter `D` returning `C`) returning pointer-to-`A`

typename

Compiler needs help parsing template definition

Keyword `typename` must precede any qualified type name that depends on a template parameter

```
A * B(C(D));
```

Function declaration:

`B` is a function with parameter pointer-to-(function with parameter `D` returning `C`) returning pointer-to-`A`

```
A* B(C (*fp)(D arg));
```

typename

Compiler needs help parsing template definition

Keyword typename must precede any qualified type name that depends on a template parameter

```
T::A * B(T::C(T::D));
```

typename

```
template <class T>
void do_something(T t) {
    // ...
    T::value_type* p(T::addr(T::object));
    // ...
}
```

https://godbolt.org/z/K38Kj6f97

typename

```
template <class T>
void do_something(T t) {
    // ...
    T::value_type* p(T::addr(T::object));
    // ...
}
```

```
<source>:4:8: error: dependent-name 'T::value_type' is parsed as a non-
type, but instantiation yields a type
    4 |     T::value_type* p(T::addr(T::object));
      |        ^~~~~~~~~~
<source>:4:8: note: say 'typename T::value_type' if a type is meant
```

https://godbolt.org/z/K38Kj6f97

typename

```
template <class T>
void do_something(T t) {
    // ...
    T::value_type* p(T::addr(T::object));
    // ...
}
```

```
<source>:4:8: error: dependent-name 'T::value_type' is parsed as a non-type, but instantiation yields a type
    4 |     T::value_type* p(T::addr(T::object));
      |        ^~~~~~~~~~
<source>:4:8: note: say 'typename T::value_type' if a type is meant
```

https://godbolt.org/z/K38Kj6f97

typename

```
template <class T>
void do_something(T t) {
    // ...
    T::value_type* p(T::addr(T::object));
    // ...
}
```

```
<source>:4:8: error: dependent-name 'T::value_type' is parsed as a non-
type, but instantiation yields a type
    4 |     T::value_type* p(T::addr(T::object));
      |        ^~~~~~~~~~
<source>:4:8: note: say 'typename T::value_type' if a type is meant
```

https://godbolt.org/z/K38Kj6f97

typename

```
template <class T>
void do_something(T t) {
    // ...
    typename T::value_type* p(T::addr(T::object));
    // ...
}
```

https://godbolt.org/z/b8zdKdn9b

Writing Function Templates

Deducible Template Parameters

Make all template parameters deducible

- Except when you can’t

Deducible Template Parameters

Make all template parameters deducible

Except when you can't

```
template <class Result, class Source>
Result my_fancy_cast(const Source& src) {
    // ...
}
```

https://godbolt.org/z/sa8n6vG7f

Deducible Template Parameters

Make all template parameters deducible

Except when you can't

```
template <class Result, class Source>
Result my_fancy_cast(const Source& src) {
    // ...
}
```

```
my_fancy_cast<Widget>(parts.getFidget())
```

https://godbolt.org/z/sa8n6vG7f

Deducible Template Parameters

Can’t deduce the parent type of a function parameter

Deducible Template Parameters

Can’t deduce the parent type of a function parameter

```
template <class T>
void f(typename T::type arg) { }
```

https://godbolt.org/z/5aEYxWqra

Deducible Template Parameters

Can’t deduce the parent type of a function parameter

```
template <class T> struct A {
    using type = T;
};
template <class T>
void g(typename A<T>::type arg) { }
```

https://godbolt.org/z/5aEYxWqra

Deducible Template Parameters

Can’t deduce the parent type of a function parameter

```
template <class T> struct A { using type = T; };
template <class T> void g(typename A<T>::type arg) { }
```

```
<source>:13:6: error: no matching function for call to 'g(int)'
   13 |     g(2);
      |     ~^~~
<source>:9:6: note:   template argument deduction/substitution failed:
<source>:13:6: note:   couldn't deduce template parameter 'T’
```

https://godbolt.org/z/5aEYxWqra

Overload sets

Avoid complicated overload sets

Overload sets

Avoid complicated overload sets

```
explicit vector( size_type count );
vector( std::initializer_list<T> init );
```

https://en.cppreference.com/w/cpp/container/vector/vector

Overload sets

Avoid complicated overload sets

```
explicit vector( size_type count );
vector( std::initializer_list<T> init );

std::vector<int> a(100);
std::vector<int> b{100};
```

https://godbolt.org/z/sjafa4sxM
https://en.cppreference.com/w/cpp/container/vector/vector

Overload sets

Avoid complicated overload sets

```
explicit vector( size_type count );
vector( std::initializer_list<T> init );

std::vector<int> a(100);
std::vector<int> b{100};
```

100 elements with value 0

1 element with value 100

https://godbolt.org/z/sjafa4sxM
https://en.cppreference.com/w/cpp/container/vector/vector

Overload sets

Avoid complicated overload sets

```
vector( size_type count, const T& value );
vector( std::initializer_list<T> init );
```

```
std::vector<int> a(100, 200);
std::vector<int> b{100, 200};
```

100 elements with value 200

2 elements with values 100, 200

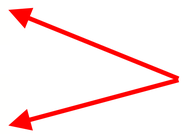
https://godbolt.org/z/sjafa4sxM
https://en.cppreference.com/w/cpp/container/vector/vector

Overload sets

Avoid complicated overload sets

```
explicit vector( size_type count );
vector( std::initializer_list<T> init );

std::vector<int*> a(100);
std::vector<int*> b{100};
```



100 elements with value `nullptr`

https://godbolt.org/z/sjafa4sxM
https://en.cppreference.com/w/cpp/container/vector/vector

Be Choosy

Avoid function templates that accept anything

Especially ones with common names

Be Choosy

Avoid function templates that accept anything

Especially ones with common names

```
template <class T, class U>
void count_stipples_and_report_daces(
          const T& t, const U& u) {
    // ...
}
```

Be Choosy

Avoid function templates that accept anything

Especially ones with common names

```
template <class T, class U>
bool operator==(const T& t, const U& u) {
    // ...
}
```

Function Template Specialization

What is allowed:

- Full specialization of non-member function templates

What is not allowed:

- Partial specialization of non-member function templates
- Any specialization of member function templates

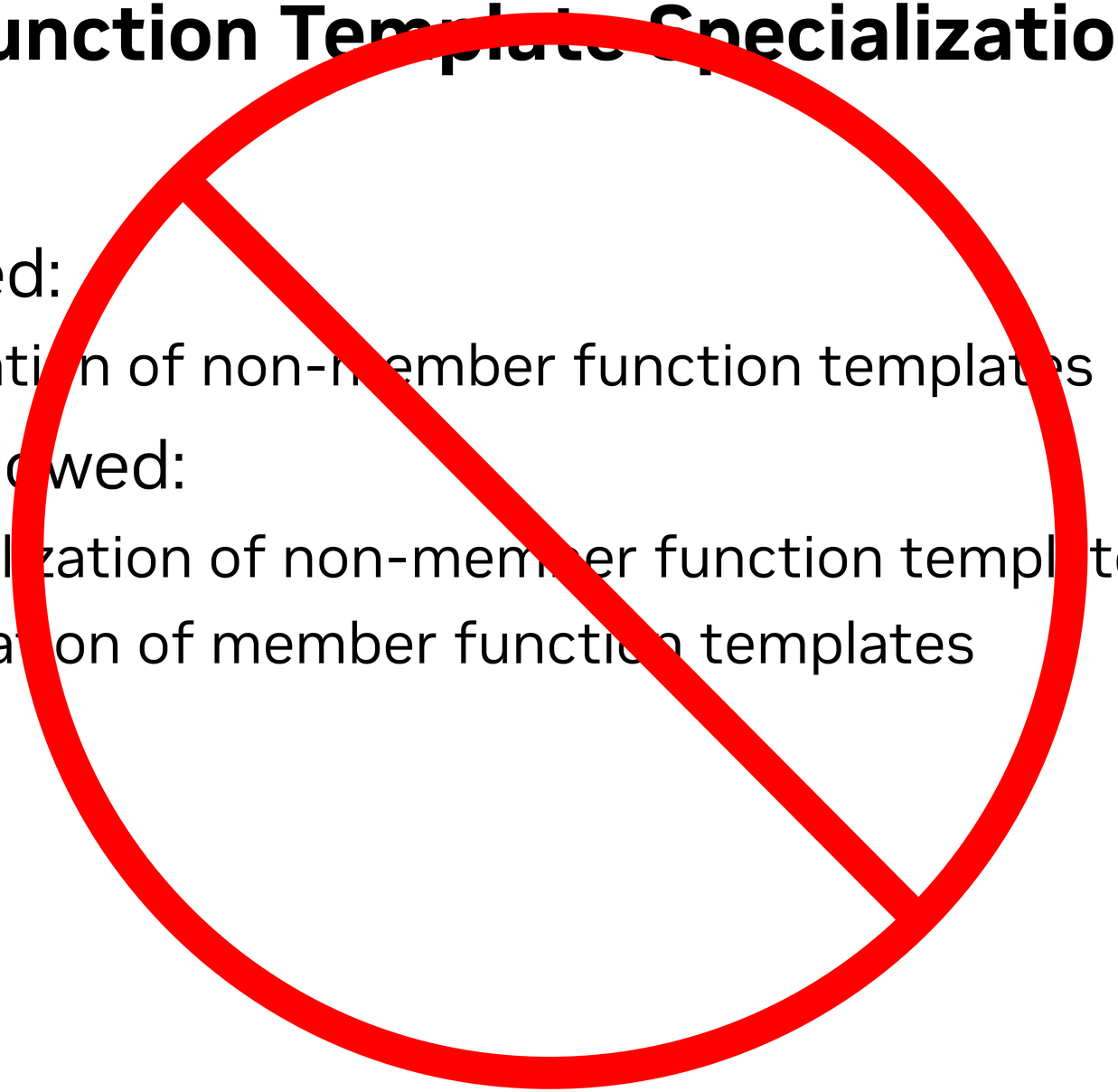
Function Template Specialization

What is allowed:

- Full specialization of non-member function templates

What is not allowed:

- Partial specialization of non-member function templates
- Any specialization of member function templates



Don’t Specialize Function Templates

Don’t Specialize Function Templates

If you want a full specialization,
create a non-template overload

If you want a partial specialization,
create template overload

Don’t Specialize Function Templates

If you want a full specialization,
create a non-template overload

```
template <class T> requires some_condition<T>
void do_something(const T& arg) {
    // ... something
}
```

https://godbolt.org/z/xar4EEMve

Don’t Specialize Function Templates

If you want a full specialization,
create a non-template overload

```
template <class T> requires some_condition<T>
void do_something(const T& arg) {
    // ... something
}
void do_something(const char* s) {
    // ... something different ...
}
```

https://godbolt.org/z/xar4EEMve

Don’t Specialize Function Templates

If you want a partial specialization,
create template overload

```
template <class C> requires is_container<C>
void process_collection(const C& c) {
    // ... processing
}
```

https://godbolt.org/z/rWP76vab1

Don’t Specialize Function Templates

If you want a partial specialization,
create template overload

```
template <class C> requires is_container<C>
void process_collection(const C& c) {
    // ... processing
}

template <class T>
void process_collection(const std::list<T>& c) {
    // ... list processing
}
```

https://godbolt.org/z/rWP76vab1

Conclusion

Resources

"*Modern Template Metaprogramming: A Compendium*"
Walter E. Brown, CppCon 2014, Part I and Part II

https://www.youtube.com/watch?v=Am2is2QCvxY
https://www.youtube.com/watch?v=a0FliKwcwXE

Summary

Define templates in header files

Substitution checks the declaration and template arguments

Instantiation checks the entire definition

SFINAE: Substitution Failure Is Not An Error

Let the compiler deduce arguments for a function template

Constrain your template parameters

Keep it simple