C++ Templates

Alessandro Casalino

Cineca,
Casalecchio di Reno, Bologna, Italy

May 8, 2024

CINECA

Outline

- Function Templates
- Run and Compile-Times
- Class Templates
- Non-datatype Templates
- Template Specialization
- `C++17`: `std::tuple` and Structured Binding
- `C++20`: Template Concepts
- `C++20`: Templated Lambda Functions

Function Templates

Example: Max

```
int max(int x, int y)
{
    return (x < y) ? y : x;
}
```

What if we need `double`?

Note

Function can be used also with `double` input (e.g., `max(2.3,3.4)`), but `-Wconversion` generates warnings. Check: godbolt.org/z/hjbdM1zs8

Example: Max

```
int max(int x, int y)
{
    return (x < y) ? y : x;
}

double max(double x, double y)
{
    return (x < y) ? y : x;
}
```

- Possible approach: overload the function with another one
- But what if we need `unsigned int`?

EXAMPLE: MAX

```
int max(int x, int y)
{
    return (x < y) ? y : x;
}

double max(double x, double y)
{
    return (x < y) ? y : x;
}

unsigned int max(unsigned int x,
                 unsigned int y)
{
    return (x < y) ? y : x;
}
```

Ok-ish, but...

- The code becomes lengthy in no time!
- We are violating **DRY (Don't Repeat Yourself)** rule
- Other datatypes are missing

Warning

Bad things might happen with pure datatype overloads: godbolt.org/z/PETfe789f

Function Templates

```
T max(T x, T y)
{
    return (x < y) ? y : x;
}
```

- We want a function with `T` a **generic** datatype
- This is actually the first step to make a template
- **But this does not compile**: what is `T` for the compiler?

Function Templates

```
T max(T x, T y)
{
    return (x < y) ? y : x;
}
```

- **First step**: substitute the datatypes

```
template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}
```

- **Second step**: declare the template
  - Datatypes in the <>
  - Keyword for datatype is `typename`

Best practice
Use capital letters for datatypes, e.g., `T`, `V`.

Function Templates

```
// Template parameters definition
template<typename T>
// Definition of max<T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}
```

- This is a *template* of the max function
- `T` is a placeholder for any datatype
- Pros:
  - Only one code to maintain
  - Compiler generates code (overloaded function) on-demand
  - Thus we don't have to anticipate (datatype) needs

Best practice

Two equivalent ways to specify datatypes for templates are available: `typename` and `class`. The second is obsolete, thus prefer `typename`.

Example: Max function

From this …

```
int max(int x, int y)
{
    return (x < y) ? y : x;
}

long int max(long int x, long int y)
{
    return (x < y) ? y : x;
}

unsigned int max(unsigned int x,
                 unsigned int y)
{
    return (x < y) ? y : x;
}

unsigned long int max(unsigned long int x,
                      unsigned long int y)
{
    return (x < y) ? y : x;
}


float max(float x, float y)
{
    return (x < y) ? y : x;
}

double max(double x, double y)
{
    return (x < y) ? y : x;
}

long double max(long double x,
                long double y)
{
    return (x < y) ? y : x;
}
```

…to this!

```
template<typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}
```

How to use Templates

```cpp
#include <iostream>

template<typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}

int main() {
    int a {3}, b {2};
    std::cout << max<int>(a, b)
              << std::endl;

    double c {4.}, d {5.};
    std::cout << max<double>(c, d)
              << std::endl;

    return 0;
}
```

Templates are **not** functions: they generate functions when we request a specific version (**template instantiation**).

- Call with `max<datatype> (arg1, arg2)`
- Compiler creates a new `max<int>(arg1, arg2)`

godbolt.org/z/3hWWKE4Y3

How to use Templates

```cpp
#include <iostream>

template <typename T>
T max(T x, T y);

template<>
int max<int>(int x, int y)
{
    return (x < y) ? y : x;
}


int main() {
    int a {3}, b {2};
    std::cout << max<int>(a, b)
              << std::endl;

    return 0;
}
```

Result of **template instantiation**.

- Compiler substitutes `T` with the requested datatype
- Analogously for each datatype requested

Note

If no template instantiation is requested, no function is instantiated in the translation unit.

godbolt.org/z/5ovzbhEqE

How to use Templates

```
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}
int max(int x, int y)
{
    return (x < y) ? y : x;
}

int main() {
    int a {3}, b {2};
    std::cout << max<int>(a, b) << std::endl;
    std::cout << max<>(a, b) << std::endl;
    std::cout << max(a, b)<< std::endl;

    return 0;
}
```

We can also rely on **template argument deduction**.

Question

Which function is called in each case?

Best practice

Unless required, use the standard function call syntax (latter in the example).

godbolt.org/z/n5YT113nK

WARNING!

```
#include <iostream>

template <typename T>
T foo(T x)
{
    return x + 2;
}

int main() {
    std::string str {"Hello world!"};

    std::cout << foo(str) << std::endl;

    return 0;
}
```

With great power comes great responsibility! - Peter Parker

Warning

This does not compile! Templates do not check the validity of propositions with all possible datatype. It is up to the programmer to call templates with meaningful datatypes. Or Template Concepts ...

Plus check the *clean and meaningful* error list from the compiler :-)
godbolt.org/z/aGe9fWjjz

Multiple Template Types

```cpp
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}

int main() {
    std::cout << max(3, 2.5)
              << std::endl;

    return 0;
}
```

Question

Does this code compiles?

godbolt.org/z/3PPerWeqn

Multiple Template Types

```cpp
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}

int main() {
    std::cout << max(3, 2.5)
              << std::endl;

    return 0;
}
```

The compiler

- checks the presence of non-templated `max(int,double)`
- tries to find a suitable template, but only `max<T>(T, T)` is available

Thus: **error**!

godbolt.org/z/3PPerWeqn

Multiple Template Types

```
// Static cast the input value
max(static_cast<double>(3), 2.5)
// Call max with explicit data-type
max<double>(3, 2.5)
```

- Solutions without modifying the template

```
// Or modify the template
// to use two datatypes
template <typename T, typename U>
T max(T x, U y)
{
    return (x < y) ? y : x;
}
```

- Template with two (potentially) different datatypes

Question

Is this safe?
Hint: compile with `-Wconversion`.

godbolt.org/z/Kq1n5eY49

Multiple Template Types

```
#include <iostream>

template <typename T, typename U>
T max(T x, U y)
{
    return (x < y) ? y : x;
}

int main() {
    std::cout << max(3, 2.5)
              << std::endl;

    return 0;
}
```

The issue

- `double` has precedence on `int` (arithmetic conversion rules)
- thus when casting on the condition we obtain `double`
- and in the template instantiation `U` = `double`, different from return type `T` = `int`

Note

The compilation succedes, but there are conversions warnings with `-Wconversion`.
In general, these should be solved!

Multiple Template Types

```
#include <iostream>

template <typename T, typename U>
auto max(T x, U y) // C++14
{
    return (x < y) ? y : x;
}

int main() {
    std::cout << max(3, 2.5)
              << std::endl;

    return 0;
}
```

Solution: let the compiler decide with `auto`!

Warning

Be careful when using `auto`! The `auto` of an operation is decided with the arithmetic conversion rules. And it drops the `const` (use `const auto` when needed).

godbolt.org/z/EP4M57Wv5

Multiple Template Types

C++20

```
#include <iostream>

auto max(auto x, auto y)
{
    return (x < y) ? y : x;
}

int main() {
    std::cout << max(3, 2.5)
              << std::endl;

    return 0;
}
```

- `C++20` feature: `auto` for function args
- Equivalent to the previous
- But this way, `x` and `y` can have different datatypes.

godbolt.org/z/noMq4s9q7

Mixed Templated Functions

```
#include <iostream>

template <typename T>
T add(T x, double y)
{
    return x * static_cast<T>(y);
}

int main() {
    std::cout << add(3.1f, 2.5)
              << std::endl;

    return 0;
}
```

We can mix template and non-template datatypes.

Note

In this dummy example, the `static_cast<T>` is necessary to convert the input and avoid conversion warnings.

godbolt.org/z/36jz8fnno

Templates and Multiple Files

main.cpp

```cpp
#include <iostream>

template <typename T, typename V>
T foo (T i, V j);

int main() {
    std::cout << foo(2,3) << std::endl;
    return 0;
}
```

templates.cpp

```cpp
template <typename T, typename V>
T foo (T i, V j)
{
    return i + j;
}
```

- **This does not compile (linker error)!**
- This is because the forward declaration and the template are in different **translation units**.

TRANSLATION UNIT

From `C++` standard:

> *A translation unit is the basic unit of compilation in C++. It consists of the contents of a single source file, plus the contents of any header files directly or indirectly included by it, minus those lines that were ignored using conditional preprocessing statements.*

Templates and Multiple Files

main.cpp

```
#include <iostream>
#include "templates.hpp"

int main() {
    std::cout << foo(2,3) << std::endl;
    return 0;
}
```

templates.hpp

```
template <typename T, typename V>
T foo (T i, V j)
{
    return i + j;
}
```

- **This now compiles correctly!**
- The template definition and the template call are in the same translation unit.

Note

Suppose the `templates.hpp` is included in several `.cpp`: a function is generated from the template for each include if used in the `.cpp`. This seems a violation of the *one-definition rule* that requires only one definition per program. But templates are exempt from this rule!

Recap

- Templates are used to create prototypes
- Templates are converted to functions on demand (**template instantiation**)
- Multiple template datatypes can be used
- Instantiation calls should be in the same translation unit as the template
- `C++20`: `auto` is equivalent to use templates. But be careful about the downsides!

Run and Compile-Times

Run and Compile-Times

Run-Time

- Operations performed when running the code
- Dynamic (run-time) polymorphism: `virtual` functions
- Approximately 2x slower than standard function calls

Compile-Time (`static`)

- Operations performed during compilation
- Static (compile-time) polymorphism: `templates`
- Functions and operators overloading
- Does not decrease run-time performance
- Keywords: `constexpr`, `consteval` (not `const`!)

EXAMPLE: `assert` AND `static_assert`

```
#include <cassert>

int main() {

    // Note the constexpr for
    // compile-time definitions
    // What happens if we remove it?
    constexpr int a {0};
    constexpr int b {1};

    // This stops compilation
    //static_assert(a == b);

    int c {0};
    int d {1};

    // This kills the code
    assert(c == d);

}
```

- `assert` for run-time
- `static_assert` for compile-time

Best practice

Use `assert` to check conditions at run-time only when really needed, otherwise prefer `static_assert`.

godbolt.org/z/Goeqd7oYq

Class Templates

Example: Class Max

```
#include <iostream>

struct Values {
    int a{}, b{};
};

int max (Values val)
{
    return (val.a > val.b)
           ? val.a : val.b;
}

int main()
{
    Values val {2, 3};
    std::cout << max(val)
              << std::endl;

    return 0;
}
```

What if we need `double`?

Example: Class Max

```cpp
struct Values {
    int a{}, b{};
};

struct Values {
    double a{}, b{};
};

int max (Values val)
{
    return (val.a > val.b)
            ? val.a : val.b;
}

double max (Values val)
{
    return (val.a > val.b)
            ? val.a : val.b;
}
```

This does not compile!
godbolt.org/z/MjejbEch9

Issues:

- Classes can not be overloaded
- The two `max` functions have only different return type
- DRY (Don't Repeat Yourself) rule violated

Class Templates

```
template <typename T>
struct Values {
    T a{}, b{};

    Values(T a, T b)
          : a {a}, b {b} {};
};

int main()
{
    Values<int> vi1 {2, 3};
    Values      vi2 {2, 3};
    Values<double> vd1 {2.1, 3.4};
    Values         vd2 {2.1, 3.4};

    return 0;
}
```

- Class Templates similar to function
- Explicit datatype template calls
- `(C++17)` **Class template argument deduction** (CTAD)

Question

What happens if we define `Values v {2, 3.1}`?

godbolt.org/z/cr6We7fhz

Class Templates C++20

```cpp
// Below only works with C++20!

template <typename T>
struct Values {
    T a{}, b{};

    // No explicit constructor:
    // works only for C++20
};

int main()
{
    Values<int> vi1 {2, 3};
    Values      vi2 {2, 3};
    Values<double> vd1 {2.1, 3.4};
    Values         vd2 {2.1, 3.4};

    return 0;
}
```

Warning

This does not compile in C++17: CTAD fails because no explicit constructor is provided. From C++20 this is possible.

godbolt.org/z/x6z6rTYK6

Template argument deduction guides C++17

```
template <typename T>
struct Values {
    T a{}, b{};
};

// Deduction guide
template <typename T>
Values(T, T) -> Values<T>;

int main()
{
    Values vi1 {2, 3};
    Values vd1 {2.1, 3.4};

    return 0;
}
```

- Either we specify the constructor explicitly, or
- In `C++17` we need to *guide* the compiler with a deduction guide

godbolt.org/z/79PzMhscG

Class Templates

```
template <typename T>
struct Values {
    T a{}, b{};
};

template <typename T>
T foo (Values<T> val)
{
    return doSomething(val);
}
```

- Templated class calls with <>

Warning

Using `Values` instead of `Values<T>` as function arg in this context generates compiler errors.
Check for instance:
godbolt.org/z/ecsdf1e6Y
godbolt.org/z/f1d8b7frY

Example: Class Max

```
#include <iostream>

template <typename T>
struct Values {
    T a{}, b{};
};

template <typename T>
T max (Values<T> val)
{
    return (val.a > val.b)
           ? val.a : val.b;
}

int main()
{
    Values val {2, 3};
    std::cout << max(val) << std::endl;
    Values val2 {2.1, 3.4};
    std::cout << max(val2) << std::endl;
    return 0;
}
```

Complete `max` example with classes.

godbolt.org/z/654Pjo54f

Multiple Template Types

```
#include <iostream>

template <typename T, typename U>
struct Values {
    T a{};
    U b{};
};

int main()
{
    Values val {2, 3.2};

    return 0;
}
```

We can create classes with multiple templated datatypes.

Templates and Headers

This does not compile (linker error)!

fooClass.h

```
#ifndef FOOCLASS_H
#define FOOCLASS_H

template <typename T, typename U>
struct fooClass {
    T i{};
    U v{};

    fooClass(T i, U v)
            : i {i}, v{v} {};

    T foo(T a, U b);
};


#endif //FOOCLASS_H
```

fooClass.cpp

```
#include "fooClass.h"

template <typename T, typename U>
T fooClass<T,U>::foo (T a, U b)
{
    return a + b;
}
```

main.cpp

```
#include <iostream>
#include "fooClass.h"

int main() {
    fooClass foo {2, 3.1};
    std::cout << foo.foo(2,3)
              << std::endl;
    return 0;
}
```

Templates and Headers

Possible solutions

fooClass.h

```
#ifndef FOOCLASS_H
#define FOOCLASS_H
template <typename T, typename U>
struct fooClass {
    T i{};
    U v{};

    fooClass(T i, U v)
            : i {i}, v{v} {};

    T foo(T a, U b)
    {
        return a + b;
    }
};
#endif //FOOCLASS_H
```

```
#ifndef FOOCLASS_H
#define FOOCLASS_H

template <typename T, typename U>
struct fooClass {
    T i{};
    U v{};

    fooClass(T i, U v)
            : i {i}, v{v} {};

    T foo(T a, U b);
};

template <typename T, typename U>
T fooClass<T,U>::foo (T a, U b)
{
    return a + b;
}

#endif //FOOCLASS_H
```

Recap

- Classes can also be templated
- And also here multiple types templates can be used
- Be careful when you create templated classes with methods using the `.cpp`/`.hpp` layout

Non-datatype Templates

Non-Datatype Templates

```
#include <iostream>

template <int N>
int multiply()
{
    return N* N;
}

int main() {
    std::cout << multiply<2>()
              << std::endl;

    return 0;
}
```

- Function can be templated with types.
- Everything is done at compile time!
- Useful when we need to pass `constexpr` values to functions

Warning

Calling `multiply()` (without `<value>`) results in compilation error.

godbolt.org/z/YfzvvW3Yh

Non-Datatype Templates

```
#include <iostream>
#include <cmath>

template <double D>
double getLog()
{
    static_assert(D >= 0.0,
    "getLog(): D must be positive");

    return std::log(D);
}

int main()
{
    std::cout << getLog<5.0>()
              << std::endl;
    std::cout << getLog<-5.0>()
              << std::endl;
    return 0;
}
```

- `static_assert` is an `assert` at compile time
- In this way the check is made at compile time, avoiding run-time errors

Warning

This code does not compile as expected: we are asking for log of negative number.

godbolt.org/z/oK4TrE69E

Template Specialization

Example

```cpp
#include <iostream>

template <typename T>
void print(T value)
{
    std::cout << value << std::endl;
}

int main()
{
    print(5);
    print(6.7);
}
```

What if we want to use scientific notation for `double`?

Example

```
#include <iostream>

template <typename T>
void print(T value)
{
    std::cout << value << std::endl;
}

template <>
void print<double>(double value)
{
    std::cout << std::scientific << value << std::endl;
}

int main()
{
    print(5);
    print(6.7);
}
```

We can use **Template Specialization**.

- The `double` case is treated differently.
- But all other datatypes use the first template definition.

godbolt.org/z/azc8oh8q7

Function partial specialization

```
#include <iostream>

template <typename T, typename U>
void print(T value, U message)
{
    std::cout << message << " : "
              << value << std::endl;
}

template <typename U>
void print<double, U>(double value, U message)
{
    std::cout << std::scientific << message << " : " << value << std::endl;
}

int main()
{
    print(5, "Hello!");
    print(6.7, "Hello!");
}
```

> **Warning**
>
> **This does not compile!**
> Function partial template specialization is not allowed in `C++`.

godbolt.org/z/1dxKdE3Yx

Multiple Template Types Specialization

```
#include <iostream>

template <typename T, typename U>
void print(T value, U message)
{
    std::cout << message << " : " << value << std::endl;
}

template <>
void print<double, const char *>(double value, const char * message)
{
    std::cout << std::scientific << message << " : " << value << std::endl;
}

int main()
{
    print(5, "Hello!");
    print(6.7, "Hello!");
}
```

But we can always specialize **all** datatypes.

godbolt.org/z/1dxKdE3Yx

Delete Template Specializations

```
#include <iostream>

template <typename T>
void print(T value) = delete;

template <>
void print<double>(double value)
{
    std::cout << value << std::endl;
}

int main()
{
    // This generates compiler errors
    // print(5);

    print(6.7);
}
```

- We can remove the possibility to use the function if not `double` (compiler error) with `delete`
- Also the converse can be done (`delete` on specialized template)

Class specialization

```
#include <iostream>

template <typename T, int size = 10>
struct Interface {
    static_assert(
        std::is_same_v<double, T> ||
        std::is_same_v<float, T>,
        "Error.");
    const T v[size];
};

template <int size>
struct Interface<double, size> {
    const double v[size];
    // Here put double stuffs
};

template <int size>
struct Interface<float, size> {
    const float v[size];
    // Here put float stuffs
};
```

Specialization is possible also with classes.

Partial specialization is possible with classes.

godbolt.org/z/1MW9nEGn4

Recap

- Template specialization allows to differentiate template instantiations based on datatypes and non-datatype templates
- Function partial specialization is forbidden
- Classes can also be specialized, and their partial specialization is possible
- `delete` can be useful to prevent instantiation of some templates

`std::tuple` and Structured Binding

TUPLE

```cpp
template <typename T>
struct A { T v; };

template <typename T>
void foo (const A<T>& a, const A<T>& b,
   T & sum, T & diff, std::string & str)
{
    sum = a.v + b.v;
    diff = a.v - b.v;
    str = diff > 0 ? "a" : "b";
}

int main()
{
    A a {1.}, b {2.};
    double sum, diff;
    std::string str;
    foo(a, b, sum, diff, str);
    std::cout << str << " is bigger" << std::endl;
    return 0;
}
```

Example: effective output of `foo` is made of two `double` and one `std::string`.

Is there a better way to manage this output?

Tuple

```
#include <tuple>

template <typename T>
struct A { T v; };

template <typename T>
std::tuple<T, T, std::string>
    foo (const A<T>& a, const A<T>& b)
{
    return {a.v + b.v,
a.v - b.v,
            a.v - b.v > 0 ? "a" : "b"};
}

int main()
{
    A a {1.}, b {2.};
    const auto result = foo(a, b);
    std::cout << std::get<2>(result)
              << " is bigger" << std::endl;
    return 0;
}
```

A way is to use the templated `std::tuple`.

Note

There is also `std::pair`, which accepts only two values. The access methods are the same.

Best practice

Use `std::pair` when you have two output values, `std::tuple` when you have more.

godbolt.org/z/GfreWcT6d

Tuple

```
#include <tuple>

template <typename T>
struct A { T v; };

template <typename T>
std::tuple<T, std::string>
     foo (const A<T>& a, const A<T>& b)
{
    return {a.v + b.v,
            a.v - b.v > 0 ? "a" : "b"};
}

int main()
{
    A a {1.}, b {2.};
    const auto result = foo(a, b);
    std::cout << "sum is " << std::get<double>(result) << std::endl;
    std::cout << std::get<std::string>(result) << " is bigger" << std::endl;
    return 0;
}
```

`std::get` can be used specifying the datatype to extract. It works only if all tuple datatypes are different.

Warning

Do not use this `std::get` when the tuple is templated (as in the example)! Bad things might happen: godbolt.org/z/bq113oKnc

Best practice

In general, use `std::get` with the tuple element number (as in the previous slide).

godbolt.org/z/zrqv4Yb96

Structured Binding

```
#include <tuple>

template <typename T>
struct A { T v; };

template <typename T>
std::tuple<T, T> foo (const A<T>& a, const A<T>& b)
{
    return {a.v + b.v, a.v - b.v};
}

int main()
{
    A a {1.}, b {2.};
    const auto [sum, diff] = foo(a, b);
    std::cout << "sum is " << sum << std::endl;
    return 0;
}
```

From C++17, `std::pair` and `std::tuple` outputs can be redirected to `auto` generated variables. This procedure is **structured binding**.

godbolt.org/z/xd9nsednP

Template Concepts

Template Concepts

C++20

```
#include <iostream>

template <typename T>
concept Integral =
        std::is_integral_v<T>;

template <typename T>
requires Integral<T>
void foo (T a)
{
    std::cout << "Concept met: "
              << a << std::endl;
}

int main()
{
    foo(1);
    foo(static_cast<size_t>(2));
    foo(static_cast<const int>(3));

    //foo(2.1); // Error
}
```

From `C++20`: **Template Concepts**

- Provide standard syntax for template instantiation conditions
- Provide meaningful template errors (previously: a mess!)

Best practice

As for datatypes, use the first capital letter for concepts, e.g., `Integral`.

godbolt.org/z/zEha4PWze

Template Concepts C++20

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

template<typename T>
concept Fundamental
          = std::is_fundamental_v<T>;

template<Fundamental T>
void foo(const T & t){
    std::cout << "T is fundamental: "
              << t << std::endl;
}

int main()
{
    foo(1);
    foo(2.3);

    std::vector v (3,2.);
    //foo(v); Error
}
```

From C++20: **Template Concepts**

Example to check if the datatype is fundamental.

godbolt.org/z/o1W9Mz13W

Template Concepts

C++20

```
#include <iostream>

template<typename T, typename V>
concept HasConditional
= requires (T t, V v) { t > v ? t : v; };

template<typename T, typename V>
requires HasConditional<T,V>
void foo(T t, V v) {
    std::cout << "Conditional: "
              << (t > v ? t : v)
              << std::endl;
}

int main()
{
    foo(1, 2);
    foo("Hello!", "124");

    double * ptr {nullptr};
    // foo(ptr, "124"); Error
}
```

From C++20: **Template Concepts**

Example to check if the datatypes admit a conditional operation between them.

godbolt.org/z/eKrdYeedn

Templated Lambda Functions

Lambda Functions

```cpp
#include <iostream>

int main() {

    // Void Lambda
    auto a = [] () {};
    a();

    // One input lambda
    auto b = [] (int i) { return i; };
    std::cout << b(5) << std::endl;

    return 0;
}
```

Anatomy of a lambda function:

- `[]`: capture clause (not covered in this course)
- `()`: args of the lambda function (as in standard functions)
- {}: body of the lambda function (as in standard functions)

Best practice

Prefer lambdas over standard functions when a small routine used in a limited context is needed.

godbolt.org/z/vd1r6EjPe

Lambda Functions

```
#include <iostream>

int main() {

    int    v[] {1,   3,   4  };
    double g[] {1.2, 3., 4.5};

    auto sum = [] (auto * v) { // C++14
        int sum {0};
        for(size_t i {0}; i < 3; i++){
            sum += v[i];
        };
        return sum;
    };
    std::cout << sum(v) << std::endl;
    std::cout << sum(g) << std::endl;

    return 0;
}
```

Warning

This code compiles, but **there are conversion errors**! We are narrowing the conversion from `double` to `int` when using `sum(g)`.

godbolt.org/z/xMjc65zdM

Templated Lambda Functions C++20

```
#include <iostream>

int main() {

    int    v[] {1,   3,  4  };
    double g[] {1.2, 3., 4.5};

    auto sum = [] <typename T> (T * v) { // C++20
        T sum {0};
        for(size_t i {0}; i < 3; i++) {
            sum += v[i];
        };
        return sum;
    };
    std::cout << sum(v) << std::endl;
    std::cout << sum(g) << std::endl;

    return 0;
}
```

From C++20: **Templated Lambda Functions**

godbolt.org/z/q66rdqb4d

Thank you!

Advanced Template Topics

Template Metaprogramming

```
#include <iostream>

template <unsigned char N>
struct factorial {
    static constexpr unsigned value
        = N * factorial<N-1>::value;
};

template <>
struct factorial<1> {
    static constexpr unsigned value=1;
};

int main() {
    std::cout << factorial<5>::value
              << std::endl;
}
```

Example of *metaprogramming*: factorial at compile time

godbolt.org/z/n5YcG69fh

References Collapsing Rules

Consider `U` a non-reference type. When we use a `typedef T`, their reference type is collapsed to l-values (`U &`) and r-values (`U &&`) according to the following table.

If	Then	and
`T = U`	`T & = U &`	`T && = U &&`
`T = U &`	`T & = U &`	`T && = U &`
`T = U &&`	`T & = U &`	`T && = U &&`