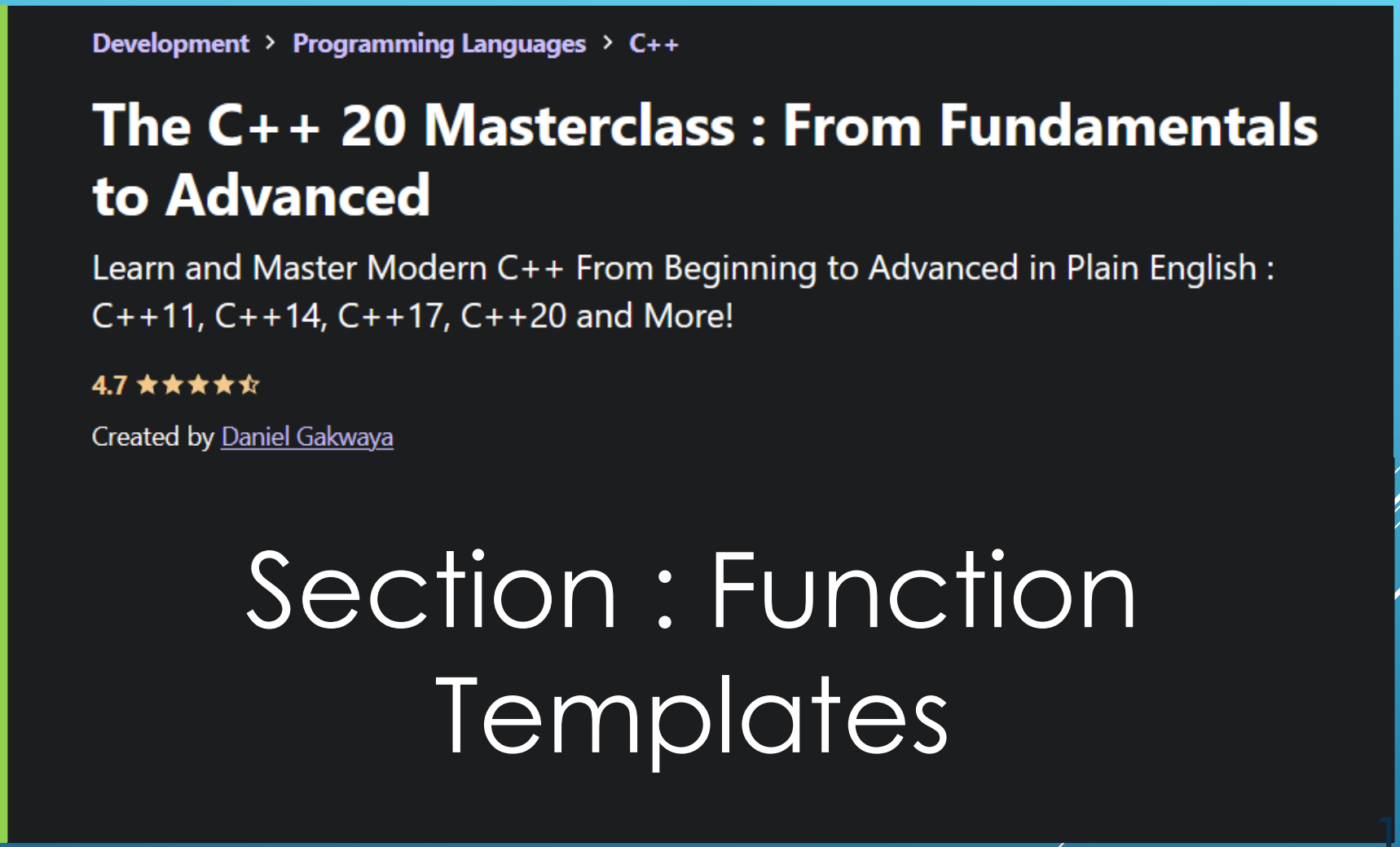
Slides



Section : Function Templates

Slide intentionally left empty

Function templates : Introduction

Function blueprints

Code Repetition

```
int max(int a, int b){
    return (a>b)? a : b;
}

double max(double a, double b){
    return (a>b)? a : b;
}
std::string_view  max(std::string_view a, std::string_view b){
     return (a>b)? a : b;
}
```

Function blue print

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

Slide intentionally left empty

Trying out function templates

Code Repetition

```
int max(int a, int b){
    return (a>b)? a : b;
}

double max(double a, double b){
    return (a>b)? a : b;
}
std::string_view  max(std::string_view a, std::string_view b){
     return (a>b)? a : b;
}
```

```cpp
template <typename T> T maximum(T a,T b);

int main(int argc, char **argv)
{
    int a{10};
    int b{23};

    double c{34.7};
    double d{23.4};

    std::string e{"hello"};
    std::string f{"world"};

    std::cout << "max(int) : " << maximum(a,b) << std::endl; // int version created
    std::cout << "max(double) : " << maximum(c,d) << std::endl;// double version created
    std::cout << "max(string) : " << maximum(e,f) << std::endl;// string version created

    /* ...

    return 0;
}

template <typename T> T maximum(T a,T b){
    return (a > b) ? a : b ; // a and b must support the > operator. Otherwise, hard ERROR
}
```

- Function templates are just blueprints. They're not real C++ code consumed by the compiler. The compiler generates real C++ code by looking at the arguments you call your function template with
- The real C++ function generated by the compiler is called a template instance
- A template instance will be reused when a similar function call (argument types) is issued. No duplicates are generated by the compiler

Summary

- Function templates are just blueprints, they're not real function declaration and definition
- Real function declarations and definitions, aka template instances are created when you call the function with arguments
- If the template parameters are of the same type (T,T), then the arguments you call the function with must also match, otherwise you get a compiler error
- Template instances won't always do what you want. A good example is when you call our maximum function with pointers. DISASTER!
- There are tools like cppinsights.io that can show you template instantiations. You can even use the debugger to infer that information from the activation record of a template function
- The arguments passed to a function template must support the operations that are done in the body of the function

Slide intentionally left empty

Template type deduction and explicit arguments

```cpp
template <typename T> T maximum(T a,T b);

int main(int argc, char **argv)
{
    int a{10};
    int b{23};
    double c{34.7};
    double d{23.4};
    std::string e{"hello"};
    std::string f{"world"};

    maximum(a,b); // int type deduced
    maximum(c,d);// double type deduced
    maximum(e,f) ;// string type deduced

    maximum<double>(c,d); // explicitly say that we want the double
                                        //version called, if an instance is not there
                                        // already, it will be created.
    maximum<double>(a,c); // Works, even if parameters are of different types,
                                          // there is implicit conversion from int to double
                                          // for the first parameter.
    maximum<double>(a,e); // Error : can't convert std::string to double.

    return 0;
}
```

Slide intentionally left empty

Template type parameters by reference

Template type parameters by value

```
template <typename T> T maximum(T a, T b);

int main(int argc, char **argv)
{
    double a {23.5};
    double b {51.2};

    std::cout << "Out - &a: " << &a << std::endl;
    double max1 = maximum(a,b);
    std::cout << "max1 : " << max1 << std::endl;


    return 0;
}

template <typename T> T maximum(T a, T b){
    std::cout << "In - &a: " << &a << std::endl;
    return (a > b ) ? a : b ;
}
```

Template type parameters by reference

```cpp
template <typename T> const T& maximum(const T& a, const T& b);

int main(int argc, char **argv)
{
    double a {23.5};
    double b {51.2};

    std::cout << "Out - &a: " << &a << std::endl;
    double max1 = maximum(a,b);
    std::cout << "max1 : " << max1 << std::endl;


    return 0;
}

template <typename T> const T& maximum(const T& a, const T& b){
    std::cout << "In - &a: " << &a << std::endl;
    return (a > b ) ? a : b ;
}
```

Confuse your compiler!

```
template <typename T> T maximum(T a, T b);
template <typename T> const T& maximum(const T& a, const T& b);

int main(int argc, char **argv)
{
    double a {23.5};
    double b {51.2};

    std::cout << "Out - &a: " << &a << std::endl;
    double max1 = maximum(a,b); // By value? By reference? Compiler  error
    std::cout << "max1 : " << max1 << std::endl;


    return 0;
}
```

Slide intentionally left empty

Template specialization

Template specialization

```
//Function template
template <typename T> T maximum(T a,T b);
//Template specialization
template <>
const char* maximum<const char*> (const char* a , const char* b);

int main(int argc, char **argv)
{
    int a{10};
    int b{23};
    double c{34.7};
    double d{23.4};
    std::string e{"hello"};
    std::string f{"world"};

    int max_int = maximum(a,b); // int type deduced
    int max_double = maximum(c,d);// double type deduced
    std::string max_str = maximum(e,f) ;// string type deduced

    const char* g{"wild"};
    const char* h{"animal"};
    //This won't do what you would expect : BEWARE!
    std::cout << "max(const char*) : " << maximum(g,h) << std::endl;
    return 0;
}
```

Specializing maximum for const char*

```
template <>
const char* maximum<const char*> (const char* a , const char* b){
    //std::strcmp doc : https://en.cppreference.com/w/c/string/byte/strcmp
    return (std::strcmp(a,b) > 0) ? a : b ;
}
```

Slide intentionally left empty

Function templates with overloading

Overloaded functions with templates

```cpp
//Function template
template <typename T> T maximum(T a,T b){
    return (a > b) ? a : b ;
}

//A raw overload will take precedence over any template instance
//if const char* is passed to maximum
const char* maximum(const char* a, const char* b){
    //std::cout << "Raw overload called" << std::endl;
    return (std::strcmp(a,b) > 0) ? a : b ;
}

//Overload through templates. Will take precedence over raw T
//if a pointer is passed to maximum
template <typename T> T* maximum(T* a, T* b){
    //std::cout << "Template overload called" << std::endl;
     return (*a > *b)? a : b;
}


int main(int argc, char **argv)
{
    /* ...
    return 0;
}
```

Slide intentionally left empty

Function templates with multiple parameters

Mysterious return type

```
template <typename T, typename P>  problematic_maximum( T a, P b);
```

BAD design : return type depends on order of parameters

```
template <typename T, typename P> P problematic_maximum( T a, P b);
```

BAD design : return type depends on order of parameters

```
template <typename T, typename P> T problematic_maximum( T a, P b);
```

BAD design : return type depends on order of parameters

```
int int_var{10};
double double_var{20.9};

std::cout << "size of return type : "
   << sizeof(problematic_maximum(int_var,double_var)) << std::endl;
```

A better approach : Separate parameter for return type

```
template <typename ReturnType, typename T, typename P>
ReturnType maximum( T a, P b){
    return (a > b) ? a : b;
}
```

A better approach : Explicit template arguments

```
int max1 = maximum<int,char,long>('c',12L);
std::cout << "max1 : " << max1 << std::endl;

int max2 = maximum<int,char>('d',12L);
std::cout << "max2 : " << max2 << std::endl;

int max3 = maximum<int>('e',14L);
std::cout << "max3 : " << max3 << std::endl;
```

The compiler can't deduce the return type

```
int max4 = maximum('f',15L); // Compiler error : The compiler
                             // doen't know how to deduce the return type.
```

The order of the arguments matters

```
template <typename ReturnType, typename T, typename P>
//Possible calls
    int max1 = maximum<int,char,long>('c',12L); // OK
    int max2 = maximum<int,char>('d',12L);      // OK
    int max3 = maximum<int>('e',14L);           // OK
    int max4 = maximum('f',14L);                // Error : return type can't be deduced

template <typename T,typename ReturnType,  typename P>
//Possible calls
    int max1 = maximum<char,int,long>('c',12L); // OK
    int max2 = maximum<char,int>('d',12L);      // OK
    int max3 = maximum<char>('e',14L);          // Error : return type can't be deduced
    int max4 = maximum('f',14L);                // Error : return type can't be deduced

template <typename T, typename P, typename ReturnType>
//Possible calls
    int max1 = maximum<char,long,int>('c',12L); // OK
    int max2 = maximum<char,long>('d',12L);     // Error : return type can't be deduced
    int max3 = maximum<char>('e',14L);          // Error : return type can't be deduced
    int max4 = maximum('f',14L);                // Error : return type can't be deduced
```

Slide intentionally left empty

Function Template return type deduction with auto

BAD design : return type depends on order of parameters

```
template <typename T, typename P> T problematic_maximum( T a, P b);
```

A better approach : Separate parameter for return type

```
template <typename ReturnType, typename T, typename P>
ReturnType maximum( T a, P b){
    return (a > b) ? a : b;
}
```

Auto return type deduction

```
template <typename T, typename P>
auto maximum(T a, P b){
    return (a > b) ? a : b;
}
```

Largest type is deduced

```
auto max1 = maximum(12.5, 33); // double return type deduced
std::cout << "max1 : " << max1 << std::endl;
std::cout << "size of max1 : " << sizeof(max1) << std::endl;


auto max2 = maximum('b',90); // int return type deduced
std::cout << "max2 : " << max2 << std::endl;
std::cout << "size of max2 : " << sizeof(max2) << std::endl;
```

Explicit template arguments : Force return type on compiler

```
//If you explicitly specify different types, largest will be deduced
auto max3 = maximum<char,char>('b',90);
std::cout << "max3 : " << max3 << std::endl;
std::cout << "size of max3 : " << sizeof(max3) << std::endl;
```

WARNING : Function definition has to come before call

```
template <typename T, typename P>
auto maximum(T a, P b);



int main(int argc, char **argv)
{

    auto max1 = maximum(12.5, 33); //
    std::cout << "max1 : " << max1 << std::endl;
    std::cout << "size of max1 : " << sizeof(max1) << std::endl;


    return 0;
}

template <typename T, typename P>
auto maximum(T a, P b){
    return (a > b) ? a : b;
}
```

Compiler error

Slide intentionally left empty

Decltype and trailing return types

Getting the type of an expression

```
//Decltype : type of an expression
int a {9};
double b{5.5};

std::cout << "size : " << sizeof(decltype((a > b)? a : b)) << std::endl;

decltype((a > b)? a : b) c {67};
std::cout << "c : " << c << std::endl;

auto max1 = maximum(a,b);
std::cout << "max1 : " << max1 << std::endl;
```

decltype as a return type

```
template <typename T, typename P>
decltype((a > b)? a : b)  maximum(T a, P b){
    return (a > b) ? a : b;
}
```

decltype as a "trailing" return type

```
template <typename T, typename P>
auto  maximum(T a, P b)-> decltype((a > b)? a : b){
    return (a > b) ? a : b;
}
```

auto

Not just about return type deduction

Splitting into function declaration and definition

```
template <typename T, typename P>
auto maximum(T a, P b);



int main(int argc, char **argv)
{

    auto max1 = maximum(12.5, 33); //
    std::cout << "max1 : " << max1 << std::endl;
    std::cout << "size of max1 : " << sizeof(max1) << std::endl;


    return 0;
}

template <typename T, typename P>
auto maximum(T a, P b){
    return (a > b) ? a : b;
}
```

Compiler error

Splitting into function declaration and definition

```
template <typename T, typename P>
auto  maximum(T a, P b)-> decltype((a > b)? a : b);

int main(int argc, char **argv)
{

    int a {9};
    double b{5.5};

    auto max1 = maximum(a,b);
    std::cout << "max1 : " << max1 << std::endl;

    return 0;
}

template <typename T, typename P>
auto  maximum(T a, P b)-> decltype((a > b)? a : b){
    return (a > b) ? a : b;
}
```

WORKS

Slide intentionally left empty

Decltype auto

decltype as a "trailing" return type

```
template <typename T, typename P>
auto  maximum(T a, P b)-> decltype((a > b)? a : b){
    return (a > b) ? a : b;
}
```

decltype(auto)

```
//decltype(auto) syntax
template <typename T, typename P>
decltype(auto)  maximum(T a, P b){
    return (a > b) ? a : b;
}

int main(int argc, char **argv)
{
    int a {9};
    double b{5.5};

    auto max1 = maximum(a,b);
    std::cout << "max1 : " << max1 << std::endl;

    return 0;
}
```

Limitation

Can’t split function code into declaration and definition

So many techniques for return type deduction with templates

What should be used

Auto return type deduction

```
template <typename T, typename P>
auto maximum(T a, P b){
    return (a > b) ? a : b;
}
```

- Deduces by value
- Definition has to be in front of main

decltype as a "trailing" return type

```
template <typename T, typename P>
auto  maximum(T a, P b)-> decltype((a > b)? a : b){
    return (a > b) ? a : b;
}
```

- Deduces references
- Keeps the constness
- Can split code into declaration and definition

decltype(auto)

```
//decltype(auto) syntax
template <typename T, typename P>
decltype(auto)  maximum(T a, P b){
    return (a > b) ? a : b;
}

int main(int argc, char **argv)
{
    int a {9};
    double b{5.5};

    auto max1 = maximum(a,b);
    std::cout << "max1 : " << max1 << std::endl;

    return 0;
}
```

- Deduces references
- Keeps the constness
- Definition has to be in front of main

Deduction with auto	• Deduces by value • Definition has to be in front of main
decltype and trailing return type	• Deduces references • Keeps the constness • Can split code into declaration and definition
decltype(auto)	• Deduces references • Keeps the constness • Definition has to be in front of main

Slide intentionally left empty

Default arguments

Default arguments

```
template <typename ReturnType = double, typename T, typename P>
ReturnType maximum ( T a, P b);

//Can also specify the default last, or anywhere really
template < typename T, typename P,typename ReturnType = double>
ReturnType minimum ( T a, P b){
    return (a < b) ? a : b;
}

int main(int argc, char **argv)
{
    /* ...

}

template <typename ReturnType, typename T, typename P>
ReturnType maximum ( T a, P b){
    return (a > b) ? a : b;
}
```

```cpp
auto max1 = maximum(10,20);
std::cout << "max1 : " << max1 << std::endl; // Will be double
std::cout << "size of max1 : " << sizeof(max1) << std::endl;

//If we return type to be int, we can specify explicit template arg
auto max2 = maximum<int>(10,20);
std::cout << "max2 : " << max2 << std::endl;
std::cout << "size of max2 : " << sizeof(max2) << std::endl;

auto min3 = minimum<int,int,int>(10,20);
std::cout << "min3 : " << min3 << std::endl;
std::cout << "size of min3 : " << sizeof(min3) << std::endl;
```

Slide intentionally left empty

Non type template parameters

Non type template parameters

```
template <int threshold,typename T>
bool is_valid(T collection[] ,size_t size)
{
    T sum{};
    for(size_t i{ 0 }; i < size; ++i) {
        sum += collection[i];
    }
    return (sum > threshold) ? true : false;
}

int main(int argc, char **argv)
{
    double temperatures[] {10.0,20.0,30.0,40.0,50.0};
    bool valid = is_valid<200,double>(temperatures,std::size(temperatures));
    std::cout << std::boolalpha << "valid : " << valid << std::endl;
    return 0;
}
```

- As of C++20, non type template parameters can be of any basic built in type (bool,double, float, int, ...), enumeration type, pointer type, or reference type.

- Any class type that has only public members can also be used as a non type template parameters(As of C++20). Such class types can be nested and be used as non type template parameters as well.

- In C++17 and below, only int like types could be used as non type template parameters

- Floating point and class type non type template parameters can come in handy in some applications.

* We will learn about class types later in the course.

Floating point non type template parameters

```
//Support for C++20 NTTP is still not fully supported
//Function below doesn't compile on gcc10 but does on gcc11 with wandbox
//Wandbox : https://wandbox.org/permlink/FqJzUV00c5MC2ie2
template <typename T, double coeff>
T process (T a, T b){
    return a*b-coeff;
}

int main(int argc, char **argv)
{
    /* ...
    std::cout << process <double,5.5>(10.0,2.0) << std::endl;
    return 0;
}
```

Slide intentionally left empty

Auto function templates

Non type template parameters

```cpp
#include <iostream>

/*
template <typename T, typename P>
decltype(auto) func( T a, P b){
    return a + b;
}
*/

auto func (auto a , auto b){
    return a + b;
}

int main(){

    auto result = func(10,20.0);
    std::cout << "result : " << result << std::endl;

    return 0;
}
```

Slide intentionally left empty

Named template parameters for lambdas

```
int main(){

    auto func = [] (auto a, auto b){
        return a + b;
    };

    std::cout << func(10,20.5) << std::endl;

    return 0;
}
```

```cpp
int main(){

    auto func = [] <typename T> (T a, T b){
        return a + b;
    };

    std::cout << func(10,20.5) << std::endl;

    return 0;
}
```

```
int main(){

    auto func = [] <typename T,typename P> (T a, P b){
        return a + b;
    };

    std::cout << func(10,20.5) << std::endl;

    return 0;
}
```

Slide intentionally left empty

Type traits

A mechanism to query information about a (templated) type at compile time

Some type traits

```
int a {45};
std::cout << std::boolalpha;
std::cout << "is_integral<int> : " << std::is_integral<int>::value << std::endl;
std::cout << "is_integral<double> : " << std::is_integral<double>::value << std::endl;
std::cout << "is_floating_point<int> : " << std::is_floating_point<int>::value << std::endl;
std::cout << "is_integral<a> : " << std::is_integral<a>::value std::endl;//ERROR : have to pass in a
                                                                          //type not a variable.
```

Conditional compile time programming

```
template <typename T>
void print_number(T n){
    static_assert(std::is_integral<T>::value , "Must pass in an integral argument");
    std::cout << "n : " << n << std::endl;
}
```

_v styntax with type traits

```
template <typename T>
void print_number(T n){
    //static_assert(std::is_integral<T>::value , "Must pass in an integral argument");
    static_assert(std::is_integral_v<T> , "Must pass in an integral argument");
    std::cout << "n : " << n << std::endl;
}


int main(int argc, char **argv)
{
    std::cout << "is_integral<int> : " << std::is_integral_v<int> << std::endl;
    std::cout << "is_integral<double> : " << std::is_integral_v<double> << std::endl;
    std::cout << "is_floating_point<int> : " << std::is_floating_point_v<int> << std::endl;
    return 0;
}
```

Slide intentionally left empty

constexpr if

Conditional compilation made easier and more flexible

Function dispatch

```
void func_floating_point  (double d) {
    std::cout << "floating point func..." << std::endl;
}
void func_integral(int i) {
    std::cout << "integral algo..." << std::endl;
}


template <typename T>
void func(T t)
{
    if constexpr(std::is_integral_v<T>)
        func_integral(t);
    else if constexpr(std::is_floating_point_v<T>)
        func_floating_point(t);
    else
        static_assert(std::is_integral_v<T> || std::is_floating_point_v<T>,
            "Argument must be integral or floating point");
}
```

Parts discarded by constexpr if aren't included in the template instance

Slide intentionally left empty

Function templates : Summary

Function blueprints

Code Repetition

```
int max(int a, int b){
    return (a>b)? a : b;
}

double max(double a, double b){
    return (a>b)? a : b;
}
std::string_view  max(std::string_view a, std::string_view b){
     return (a>b)? a : b;
}
```

Function blue print

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

- Setting up your first function template
- Template type deduction and explicit template arguments
- Template parameters by reference
- Template specialization
- Overloads and Template specialization
- Multiple template parameters
- Return type deduction with auto
- Declytpe and trailing return types
- decltype(auto)
- default arguments
- Non type template parameters
- auto function templates
- Named template parameters for lambda functions
- type traits and static asserts
- constexpr if

Slide intentionally left empty