Templates

Function templates

C++ supports 3 different kinds of template parameters:

- Type template parameters (where the template parameter represents a type).
- Non-type template parameters (where the template parameter represents a constexpr value).
- Template template parameters (where the template parameter represents a template).
  Type template parameters are by far the most common, so we'll be focused on those. We'll cover non-type template parameters in the chapter on arrays.

```
template <typename T> // this is the template parameter declaration
T max(T x, T y) // this is the function template definition for max<T>
{
    return (x < y) ? y : x;
}
```

best practise: Use a single capital letter starting with T (e.g. T, U, V, etc…) to name type template parameters that are used in trivial or obvious ways.

If the type template parameter has a non-obvious usage or meaning, then a more descriptive name is warranted. (e.g. Allocator or TAllocator).

Function template instantiation

```
#include <iostream>

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

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function
max<int>(int, int)

    return 0;
}
```

Using function templates in multiple files

```
#ifndef MAX_H
#define MAX_H

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

#endif
```

Templates that are needed in multiple files should be defined in a header file, and then #included wherever needed. This allows the compiler to see the full template definition and instantiate the template when needed.

Function templates with multiple template types

```
#include <iostream>

template <typename T, typename U> // We're using two template type
parameters named T and U
T max(T x, U y) // x can resolve to type T, and y can resolve to type U
{
    return (x < y) ? y : x; // uh oh, we have a narrowing conversion problem
here
}

int main()
{
    std::cout << max(2, 3.5) << '\n'; // resolves to max<int, double>

    return 0;
}
```

auto return type

However, the above code still has a problem: using the usual arithmetic rules (10.5 -- Arithmetic conversions), double takes precedence over int, so our conditional operator will return a double. But our function is defined as returning a T -- in cases where T resolves to an int, our double return value will undergo a narrowing conversion to an int, which will produce a warning (and possible loss of data).

Making the return type a U instead doesn't solve the problem, as we can always flip the order of the operands in the function call to flip the types of T and U.

How do we solve this? This is a good use for an **auto** return type -- we'll let the compiler deduce what the return type should be from the return statement:

```
#include <iostream>

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

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}
```

Abbreviated function templates (C++ 20)

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

is the same as:

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

Function templates may be overloaded

```
#include <iostream>

// Add two values with matching types
template <typename T>
T add(T x, T y)
{
    return x + y;
}

// Add two values with non-matching types
// As of C++20 we could also use auto add(auto x, auto y)
template <typename T, typename U>
T add(T x, U y)
{
    return x + y;
}

// Add three values with any type
// As of C++20 we could also use auto add(auto x, auto y, auto z)
template <typename T, typename U, typename V>
T add(T x, U y, V z)
{
    return x + y + z;
}

int main()
{
    std::cout << add(1.2, 3.4) << '\n'; // instantiates and calls
add<double>()
    std::cout << add(5.6, 7) << '\n';   // instantiates and calls
add<double, int>()
    std::cout << add(8, 9, 10) << '\n'; // instantiates and calls add<int,
int, int>()

    return 0;
}
```

Non-type template parameters

A non-type template parameter can be any of the following types:

- An integral type
- An enumeration type
- std::nullptr_t
- A floating point type (since C++20)
- A pointer or reference to an object
- A pointer or reference to a function
- A pointer or reference to a member function

- A literal class type (since C++20)

```
#include <iostream>

template <int N> // declare a non-type template parameter of type int named
N
void print()
{
    std::cout << N << '\n'; // use value of N here
}

int main()
{
    print<5>(); // 5 is our non-type template argument

    return 0;
}
```

What are non-type template parameters useful for?

As of C++20, function parameters cannot be constexpr. This is true for normal functions, constexpr functions (which makes sense, as they must be able to be run at runtime), and perhaps surprisingly, even consteval functions.

```
#include <cmath> // for std::sqrt
#include <iostream>

template <double D> // requires C++20 for floating point non-type parameters
double getSqrt()
{
    static_assert(D >= 0.0, "getSqrt(): D must be non-negative");

    if constexpr (D >= 0) // ignore the constexpr here for this example
        return std::sqrt(D); // strangely, std::sqrt isn't a constexpr
function (until C++26)

    return 0.0;
}

int main()
{
    std::cout << getSqrt<5.0>() << '\n';
    std::cout << getSqrt<-5.0>() << '\n';
    return 0;
}
```

Type deduction for non-type template parameters using auto

```
#include <iostream>

template <auto N> // deduce non-type template parameter from template
argument
void print()
{
    std::cout << N << '\n';
}

int main()
{
    print<5>();   // N deduced as int `5`
    print<'c'>(); // N deduced as char `c`

    return 0;
}
```