TEMPLATE META PROGRAMMING

- **Overview**
  - Recap of Template Functions, Classes, Variables, and Types
  - Template Meta Functions
  - Type Traits
- **References**
  - Marius Bancila: Template Metaprogramming with C++. Packt Publishing (2022)
  - Scott Meyers: Effective Modern C++. O'Reilly (2014)
  - Walter E. Brown: Modern Template Metaprogramming: A Compendium. CppCon (2014). https://github.com/CppCon/CppCon2014.

FUNCTION TEMPLATES

Placeholder for a type.

```
template<typename T1, ..., typename Tn>
T1 AFunction( T2 aParam1, ..., Tn aParamn )
{
    // body of function
};

type1 value = AFunction<type1, ..., typen>( value1, ..., valuen );
```

$T_1, \ldots, T_n$ occur in function signature.

$T_1, \ldots, T_n$ instantiated in function specialization.

Types of $value_1, \ldots, value_n$ must match $type_1, \ldots, type_n$

CLASS TEMPLATES

```
template<typename T1, ..., typename Tn>
class AClassTemplate
{
    // class specification
};

AClassTemplate<type1, ..., typen> obj{};
```

Placeholder for a type.

$T_1, \ldots, T_n$ occur in class specification.

$T_1, \ldots, T_n$ instantiated in class specialization.

new initializer syntax

VARIABLE TEMPLATES

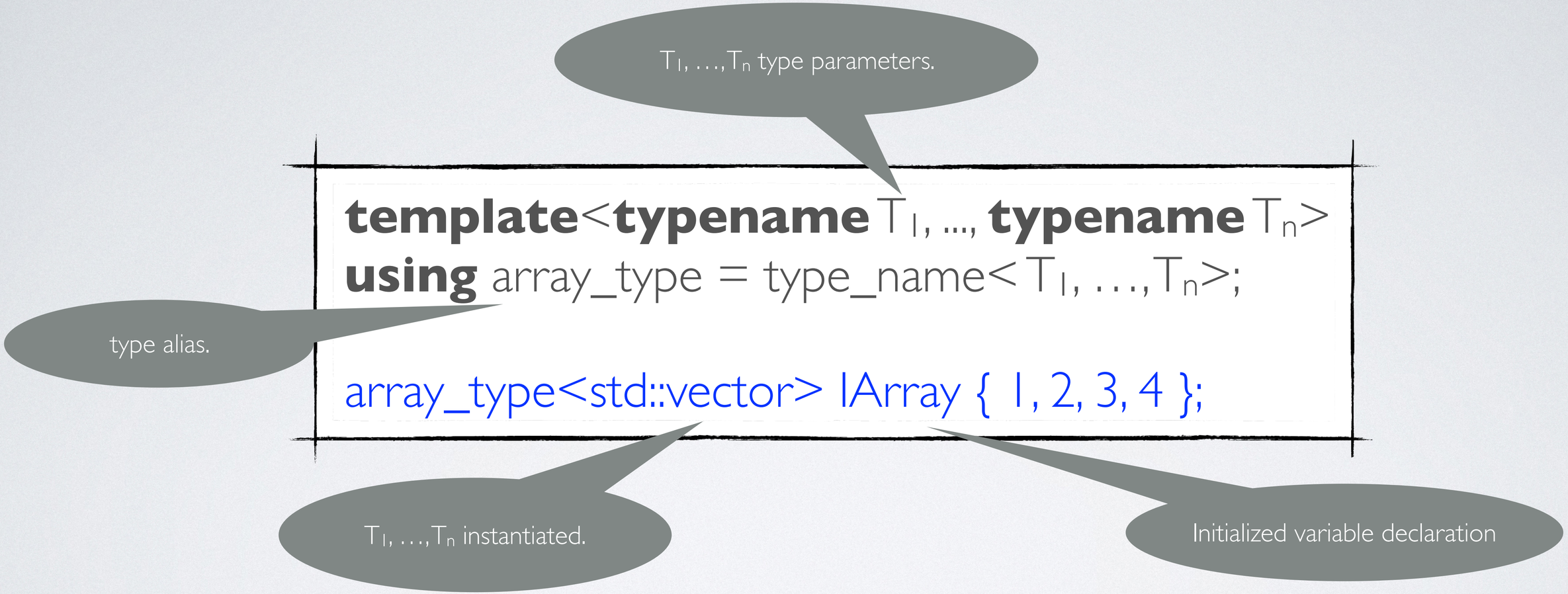
Placeholder for a type.

```
template<typename T>
T pi_v = T(3.14159265358979323846);

float pif = pi_v<float>;
```

Template variable pi_v instantiated to float.

ALIAS TEMPLATES



```
template<typename Key, typename Value>
struct Map
{
    Key key;
   Value value;

    Map( const Key& aKey = Key{}, const Value& aValue = Value{}) noexcept:
        key(aKey), value(aValue)
    {}

    template<typename U>
    operator U() const noexcept { return static_cast<U>(value); }

    template<typename K, typename V>
    friend std::istream& operator>>( std::istream& aIStream, Map<K,V>& aMap );

    template<typename K, typename V>
    friend std::ostream& operator<<( std::ostream& aOStream, const Map<K,V>& aMap );
};

template<typename K, typename V>
std::istream& operator>>( std::istream& aIStream, Map<K,V>& aMap )
{
    return aIStream >> aMap.key >> aMap.value;
}

template<typename K, typename V>
std::ostream& operator<<( std::ostream& aOStream, const Map<K,V>& aMap )
{
    return aOStream << "{" << aMap.key << ", " << aMap.value << "}";
}
```

TEMPLATE MAP

- The type Map is a public template class.
- The type conversion operator U() is a member operator.
- The input and output operators are friends. They are not member operators.
- The non-member operators are template functions over two template type arguments, K and V.

TEMPLATE META FUNCTIONS

DEFINING META FUNCTIONS

- Template meta programming allows one to write code that is evaluated at compile time.
- We use C++ structs to define meta functions. More precisely, we define public class templates that expose a member variable, called value, that denotes the result value of meta functions.
- The meta function is evaluated at compile time. A meta function is invoked by specializing the corresponding class template with suitable types and constant expressions.
- The compiler's overload resolution mechanism allows us to compute the result of a meta function.

GREATEST COMMON DIVISOR

```
template<uint64_t M, uint64_t N>
struct gcd                          // default case
{
    static constexpr uint64_t value = gcd<N, M%N>::value;
};

template<uint64_t M>
struct gcd<M, 0>                    // partial specialization
{
   static constexpr uint64_t value = M;
};
```

- The meta function gcd computes the greatest common divisor.
- The result is available via the constant static member variable value.
- The calculation of the meta function gcd is defined recursively. The recursive case is the most general function overload. The partial specialization is the base case.
- The C++ compiler considers a specialization the better option when two competing solutions are available.

FIBONACCI

```
template<unsigned N, uint64_t Previous = 0UL, uint64_t Current = 1UL>
struct fib                                      // default case
{
    static_assert( N > 1 );

    static constexpr uint64_t value = fib<N-1U, Current, Previous + Current>::value;
};

template<uint64_t Previous, uint64_t Current>
struct fib<1U, Previous, Current>   // partial specialization
{
    static constexpr uint64_t value = Current;
};
```

ACKERMANN

```
template<uint64_t M, uint64_t N>
struct ackermann
{
   static constexpr uint64_t value = ackermann<M-1, ackermann<M, N-1>::value>::value;
};

template<uint64_t N>
struct ackermann<0U, N>
{
    static constexpr uint64_t value = N+1;
};

template<uint64_t M>
struct ackermann<M, 0U>
{
    static constexpr uint64_t value = ackermann<M-1, 1>::value;
};
```

TEST META FUNCTIONS

```
std::cout << "gcd(24, 80) = " << gcd<24, 80>::value << std::endl;

std::cout << "fib(1) = " << fib<1>::value << std::endl;
std::cout << "fib(2) = " << fib<2>::value << std::endl;
std::cout << "fib(3) = " << fib<3>::value << std::endl;
std::cout << "fib(4) = " << fib<4>::value << std::endl;
std::cout << "fib(5) = " << fib<5>::value << std::endl;
std::cout << "fib(93) = " << fib<93>::value << std::endl;

std::cout << "Ackerman(2, 2) = " << ackermann<2, 2>::value << std::endl;
std::cout << "Ackerman(3, 2) = " << ackermann<3, 2>::value << std::endl;
std::cout << "Ackerman(3, 5) = " << ackermann<3, 5>::value << std::endl;
// Next is impossible: template recursion limited to depth of 1024
//    std::cout << "Ackerman(4, 1) = " << ackermann<4, 1>::value << std::endl;
```

```
gcd(24, 80) = 8
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(93) = 12200160415121876738
Ackerman(2, 2) = 7
Ackerman(3, 2) = 29
Ackerman(3, 5) = 253
```

HELPER VARIABLE TEMPLATES

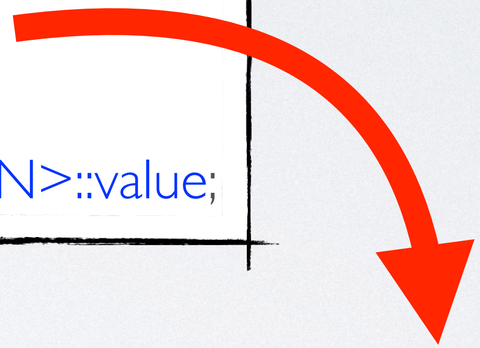
- C++17 introduced the concept of helper variable templates to facilitate the use of meta functions.
- We create a template variable whose name has the postfix _v and bind the value of a meta function to it.

USING HELPER TEMPLATE VARIABLES

```
template<uint64_t M, uint64_t N>
constexpr uint64_t gcd_v = gcd<M, N>::value;

template<unsigned N>
constexpr uint64_t fib_v = fib<N>::value;

template<uint64_t M, uint64_t N>
constexpr uint64_t ackermann_v = ackermann<M, N>::value;
```



```
std::cout << "gcd(24, 80) = " << gcd_v<24, 80> << std::endl;
std::cout << "fib(93) = " << fib_v<93> << std::endl;
std::cout << "Ackerman(3, 5) = " << ackermann_v<3, 5> << std::endl;
```

TYPE TRAITS

A MOTIVATING EXAMPLE

```
std::vector values = { 10, 30, 23, 9, -19, 45, 28, 100, -35, 8 };

auto _30 = std::find( values.cbegin(), values.cend(), 30 );
auto _45 = std::find( values.cbegin(), values.cend(), 45 );

std::cout << iter_distance( _30, _45 ) << std::endl;
```

- We wish to compute the distance between two iterators.

  How do we define the function iter_distance()?

TEMPLATE FUNCTION ITER_DISTANCE

```
template<typename It>
size_t iter_distance(It first, It last)
{
    return last - first;
}
```

- We define a template function over the iterator type and use subtraction to calculate the distance between two iterators. Calculating the distance is O(1).
- This is not a working solution as it only works for random access iterators.

ITER_DISTANCE: GENERAL SOLUTION

```
template<typename It>
size_t iter_distance(It first, It last)
{
    size_t result = 0;

    while ( first != last )
    {
        ++result;
        ++first;
    }

    return result;
}
```

- We calculate the distance between two iterators in a while loop.
- For all instantiations of template type parameter It, this approach always works, but it is not efficient.
- Calculating the distance is O(n).

TYPE-ATTRIBUTE-BASED COMPILE-TIME DECISION

```
template<typename It>
size_t iter_distance(It first, It last)
{
    If type It is a random access iterator
        return last - first;
    otherwise
        size_t result = 0;

        while ( first != last )
        {
            ++result;
            ++first;
            }

        return result;
}
```

- We inspect the attributes of the template type argument at compile time.
- If the template function iter_distance has been specialized with a random access iterator, use the O(1) approach and remove the O(n) option. Otherwise, do the opposite.
- To perform this compile-time analysis, type traits and concepts were invented and added to the C++ language.

THE BASE CLASS FOR C++ TYPE TRAITS

```
template<typename T, T v>
struct integral_constant
{
    static constexpr T value = v;            // v is a constant

    using value_type = T;
    using type = integral_constant<T, v>;    // using injected class name

    // conversion function
    constexpr operator value_type() const noexcept { return value; }

    // compile-time function object
    constexpr value_type operator()() const noexcept { return value; }
};

template<bool B>
using bool_constant = integral_constant<bool, B>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

USING INTEGRAL_CONSTANTS

```
using two = integral_constant<int, 2>;
using four = integral_constant<int, 4>;

static_assert( two::value << 1 == four::value, "2 << 1 != 4" );
static_assert( two() * 2 == four(), "2*2 != 4" );
```

- We can use integral_constant as a meta function and evaluate its value at compile time.
- We obtain the result by accessing the value member or using the compile-time function object.

COMPARING TYPES: IS_SAME

```
template<typename, typename>
struct is_same : false_type {};

template<typename T>
struct is_same<T,T> : true_type {};

template<typename U, typename V>
constexpr bool is_same_v = is_same<U,V>::value;
```

- We use inheritance to create new meta functions.
- The meta function is_same compares two types. We use two corresponding definitions.
- The default is false.
- The specialization uses the type argument for both types. This is the better match and is chosen if the two types are the same.

TESTING IS_SAME

```
using two = integral_constant<int, 2>;
using four = integral_constant<int, 4>;
using myint = int;

std::cout << "Are two and four the same? " << is_same_v<two, four> << std::endl;
std::cout << "Are int and int the same? " << is_same_v<int, int> << std::endl;
std::cout << "Are int and myint the same? " << is_same_v<int, myint> << std::endl;
```

```
Are two and four the same? 0
Are int and int the same? 1
Are int and myint the same? 1
```

- The types two and four are different.
- The types **int** and myint are the same.

WHAT IS THE USE OF THE NESTED TYPE MEMBER?

SUBSTITUTION FAILURE IS NOT AN ERROR

- Substitution Failure Is Not An Error (short SFINAE) is a meta-programming technique that applies during overload resolution of function templates.
- When substituting the explicitly specified or deduced type for the template parameter fails, the specialization is discarded from the overload set rather than causing a compile error.
- An error is reported only if no match remains in the overload set.
- Type traits use SFINAE extensively to control the existence of the nested member type in a meta-class.

ENABLE_IF

```
template<bool, typename = void>
struct enable_if
{
    // SFINAE-based method to force the compiler to pick an overload
};

template<typename T>
struct enable_if<true, T>
{
    using type = T;
};

template<bool B, typename T>
using enable_if_t = enable_if<B, T>::type;
```

- The meta function enable_if defines a public type member if B is **true**. Otherwise, there is no type member.
- It removes functions from the overload candidate set based on type traits.

REMOVE CANDIDATES

```
template<typename T>
enable_if_t<is_same_v<T, float>>      // void
    print( const std::string& aText, const T& aValue )
    {
        std::cout << aText << aValue << std::endl;
    }

template<typename T>
enable_if_t<is_same_v<T, double>> // void
    print( const std::string& aText , const T& aValue )
    {
        std::cout << aText << std::scientific << aValue << std::endl;
    }
```

```
print( "Value is ", 10.2f );
print( "Value is ", 20.4 );
// Error: No matching function call to 'print'.
//print( "Value is “, "Hello, world!" );
```

- The template function print is enabled for **float** and **double** only.
- In all other cases, enabled_if does not have a nested type member.

THE AMAZING WORKHORSE VOID_T

```
template<typename...>
using void_t = void;
```

- The alias template void_t is a variadic template that maps to type **void**.
- The underlying trick is that template type arguments to void_t must be well-formed for void_t to succeed.
- If any of the template type arguments is ill-formed, SFINAE discards the whole part containing void_t. (It is removed from the overload candidates.)
- Walter E. Brown presented void_t at CppCon 2014. It was immediately adopted in C++17.

DECLTYPE AND STD::DECLVAL

- The **decltype** specifier returns a type expression. The **decltype** specifier inspects the declared type of an entity or the type and value category of an expression.
- The std::declval type operator is a helper template for writing expressions that appear in unevaluated contexts, typically the operand to **decltype**.
- For instance, we write

```
decltype( std::declval<const T&>() - std::declval<const T&>() )
```

to denote the type of a subtraction operator for type T, if type T supports it.

META FUNCTION HAS_SUBTRACTION

```
template<typename T>
using subtraction_t = decltype( std::declval<const T&>() - std::declval<const T&>() );

template<typename, typename = void>
struct has_subtraction : false_type {};          // default, no subtraction operation

template<typename T>
struct has_subtraction<T, void_t<subtraction_t<T>>> :
    is_same<subtraction_t<T>, long> {};          // type T supports subtraction operation

template<typename T>
constexpr bool has_subtraction_v = has_subtraction<T>::value;
```

TEST FOR REQUIRED SUBTRACTION OPERATION

```
struct F
{
    long operator-( const F& ) const;
};

std::cout << "Does type F support required subtraction? "
          << has_subtraction_v<F> << std::endl;

struct G
{
};

std::cout << "Does type G support required subtraction? "
          << has_subtraction_v<G> << std::endl;
```

```
Does type F support required subtraction? 1
Does type G support required subtraction? 0
```

- Type F supports the required subtraction operation returning **long**, whereas type G does not.

```
template<typename It>
size_t iter_distance(It first, It last)
{
    if constexpr ( has_subtraction_v<It> )
    {
        return last - first;
    }
    else
    {
        size_t result = 0;

        while ( first != last )
        {
            ++result;
            ++first;
            }

        return result;
    }
}
```

IF CONSTEXPR

- We can use our type trait has_subtraction to control which approach the compiler uses.
- If the template function iter_distance has been specialized with an iterator that supports subtraction (like a random access iterator), the compiler enables the O(1) approach and removes the O(n) option. Otherwise, it does the opposite.
- The notation **if constexpr** enables the compiler to optimize code generation. If the condition yields **true**, the false statement is discarded. Otherwise, the true statement is discarded.
- Please note that has_subtraction demonstrates the principle. Production code uses a different type trait that tests the iterator category.

QUESTIONS