Templates

Motivation

Functionality is often independent of a specific type `T`

- E.g. `swap(T& a, T& b)`
- E.g. `std::vector<T>`
- Many more examples (e.g. in exercises)

Functionality should be available for all suitable types `T`

- How to avoid massive code duplication?
- How to account for user-defined types?

Templates

A *template* defines a family of classes, functions, type aliases, or variables

- Templates are parameterized by one or more template parameters
  - Type template parameters
  - Non-type template parameters
  - Template template parameters
- In order to use a template, *template arguments* need to be provided
  - Template arguments are substituted for the template parameters
  - Results in a *specialization* of the template
- Templates are a *compile-time* construct
  - When used (inaccurate, more details soon), templates are *instantiated*
  - Template instantiation actually compiles the code for the respective specialization

Example

(Simplified) definition of `std::vector`

```
class A;
//-----------------------------------------------------------
template<class T>  // T is a type template parameter
class vector {
    public:
    /* ... */
    void push_back(T&& element);
    /* ... */
};
//-----------------------------------------------------------
int main() {
    vector<int> vectorOfInt; // int is substituted for T
    vector<A> vectorOfA;     // A is substituted for T
}
```

Template Syntax

Several C++ entities can be declared as templates

- Syntax: `template` < *`parameter-list`* > *`declaration`*

`parameter-list` is a comma-separated list of template parameters

- Type template parameters
- Non-type template parameters
- Template template parameters

`declaration` is one of the following declarations

- `class`, `struct` or `union`
- A nested member class or enumeration type
- A function or member function
- A static data member at namespace scope or a data member at class scope
- A type alias

Type Template Parameters

Type template parameters are placeholders for arbitrary types

- Syntax: `typename` *name* or `class` *name*
- *name* may be omitted (e.g. in forward declarations)
- There is no difference between using `typename` or `class`
- In the body of the template declaration, *name* is a type alias for the type supplied during instantiation

```
template <class, class>
struct Baz;
//------------------------------------------------------------
template <class T>
struct Foo {
    T bar(T t) {
        return t + 42;
    }
};
```

Non-Type Template Parameters

Non-type template parameters are placeholders for certain values

- Syntax: *type name*
- *name* may be omitted (e.g. in forward declarations)
- *type* may be an integral type, pointer type, enumeration type or lvalue reference type
- Within the template body, *name* of a non-type parameter can be used in expressions

```
template<class T, unsigned N>
class Array {
    T storage[N];

    public:
    T& operator[](unsigned i) {
        assert(i < N);
        return storage[i];
    }
};
```

Template Template Parameters

Type template parameters can themselves be templated

- Syntax: `template` < *parameter-list* > `typename` *name* or `template` < *parameter-list* > `class` *name*
- *name* may be omitted (e.g. in forward declarations)
- Within the template body, *name* is a template name, i.e. it needs template arguments to be instantiated

```
template<template <class, unsigned> class ArrayType>
class Foo {
    ArrayType<int, 42> someArray;
};
```

Rarely used or required, should be avoided whenever possible

Default Template Arguments

All three types of template parameters can have default values

- Syntax: *template-parameter = default*
- *default* must be a type name for type and template template parameters, and a literal for non-type template parameters
- Template parameters with default values may not be followed by template parameters without default values

```
template<typename T = std::byte, unsigned Capacity = 1024>
class Buffer {
    T storage[Capacity];
};
```

Using Templates

In order to use a templated entity, template arguments need to be provided

- Syntax: *`template-name < parameter-list >`*
- *`template-name`* must be an identifier that names a template
- *`parameter-list`* is a comma-separated list of template arguments
- Results in a *specialization* of the template

Template arguments must match the template parameters

- At most as many arguments as parameters
- One argument for each parameter without a default value

In some cases, template arguments can be *deduced*

- More details soon

Type Template Arguments

Template arguments for type template parameters must name a type (which may be incomplete)

```
class A;
//--------------------------------------------------------------
template<class T1, class T2 = int, class T3 = double>
class Foo { };
//--------------------------------------------------------------
int main() {
    Foo<int> foo1;
    Foo<A> foo2;
    Foo<A*> foo3;
    Foo<int, A> foo4;
    Foo<int, A, A> foo5;
}
```

Non-Type Template Arguments (1)

Template arguments for non-type template parameters must be (converted) constant expressions

- Converted constant expressions can be evaluated at compile-time
- May incur a limited set of implicit conversions
- The (possibly implicitly converted) type of the expression must match the type of the template parameter

Restrictions for non-type template parameters of reference or pointer type

- May not refer to a subobject (non-static class member, base subobject)
- May not refer to a temporary object
- May not refer to a string literal

Non-Type Template Arguments (2)

Example

```
//------------------------------------------------------------
template <unsigned N>
class Foo { };
//------------------------------------------------------------
int main() {
    Foo<42u> foo1; // OK: no conversion
    Foo<42> foo2;  // OK: numeric conversion
}
```

constexpr

Functions or variables cannot be evaluated at compile time by default

- Use the `constexpr` keyword to indicate that the value of a function or variable can be evaluated at compile time
- `constexpr` variables must have literal type and be immediately initialized
- `constexpr` functions must have literal return and parameter types

```
#include <array>
//----------------------------------------------------------------
class Element;
//----------------------------------------------------------------
class Foo {
    constexpr unsigned numElements = 42;
    constexpr unsigned calculateBufferSize(unsigned elements) {
        return elements * sizeof(Element);
    }

    std::array<std::byte, calculateBufferSize(numElements)> array;
};
```

Template Template Arguments

Arguments to template template arguments must name a class template or template alias

```
#include <array>
//--------------------------------------------------------------
template <class T, unsigned long N>
class MyArray { };
//--------------------------------------------------------------
template <template<class, unsigned long> class Array>
class Foo {
    Array<int, 42> bar;
};
//--------------------------------------------------------------
int main() {
    Foo<MyArray> foo1;
    Foo<std::array> foo2;
}
```

Example: Class Templates

```
template <class T, unsigned long N>
class MyArray {
    private:
    T storage[N];

    public:
    /* ... */

    T& operator[](unsigned long index) {
        return storage[index];
    }

    const T& operator[](unsigned long index) const {
        return storage[index];
    }

    /* ... */
};
```

Example: Function Templates

```
class A { };
//-----------------------------------------------------------------
template <class T>
void swap(T& a, T& b) {
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
//-----------------------------------------------------------------
int main() {
    A a1;
    A a2;

    swap<A>(a1, a2);
    swap(a1, a2);    // Also OK: Template arguments are deduced
}
```

Example: Alias Templates

```cpp
namespace something::extremely::nested {
//----------------------------------------------------------------
template <class T, class R>
class Handle { };
//----------------------------------------------------------------
}  // namespace something::extremely::nested
//----------------------------------------------------------------
template <typename T>
using Handle = something::extremely::nested::Handle<T, void*>;
//----------------------------------------------------------------
int main() {
    Handle<int> handle1;
    Handle<double> handle2;
}
```

Example: Variable Templates

```cpp
template <class T>
constexpr T pi = T(3.1415926535897932385L);
//------------------------------------------------------------------
template <class T>
T area(T radius) {
    return pi<T> * radius * radius;
}
//------------------------------------------------------------------
int main() {
    double a = area<double>(1.0);
}
```

Example: Class Member Templates

```cpp
#include <iostream>
#include <array>
//------------------------------------------------------------------
struct Foo {
    template <class T>
    using ArrayType = std::array<T, 42>;

    template <class T>
    void printSize() {
        std::cout << sizeof(T) << std::endl;
    }
};
//------------------------------------------------------------------
int main() {
    Foo::ArrayType<int> intArray;

    Foo foo;
    foo.printSize<Foo::ArrayType<int>>();
}
```

Template Instantiation

A function or class template by itself is not a type, an object, or any other entity

- No assembly is generated from a file that contains only template definitions
- A template specialization must be *instantiated* for any assembly to appear

Template instantiation

- Compiler generates an actual function or class for a template specialization
- Explicit instantiation: Explicitly request instantiation of a specific specialization
- Implicit instantiation: Use a template specialization in a context that requires a complete type

Explicit Template Instantiation (1)

Forces instantiation of a template specialization

- Class template syntax
  - `template class` *template-name* `<` *argument-list* `>;`
  - `template struct` *template-name* `<` *argument-list* `>;`
- Function template syntax
  - `template` *return-type name* `<` *argument-list* `>` `(`*parameter-list*`);`

Explanation

- Explicit instantiations have to follow the one definition rule
- Generates assembly for the function specialization or class specialization and all its member functions
- Template definition must be visible at the point of explicit instantiation

Explicit Template Instantiation (2)

Example

```
template <class T>
struct A {
    T foo(T value) { return value + 42; }

    T bar() { return 42; }
};
//------------------------------------------------------------------
template <class T>
T baz(T a, T b) {
    return a * b;
}
//------------------------------------------------------------------
// Explicit instantiation of A<int>
template struct A<int>;
// Explicit instantiation of baz<float>
template float baz<float>(float, float);
```

Implicit Template Instantiation (1)

Using a template specialization in a context that requires a complete type triggers implicit instantiation

- Only if the specialization has not been explicitly instantiated
- Members of a class template are only implicitly instantiated if they are actually used

The definition of a template must be visible at the point of implicit instantiation

- Definitions must usually be provided in the *header* file if implicit instantiation is desired

Implicit Template Instantiation (2)

Example

```cpp
template <class T>
struct A {
    T foo(T value) {
        return value + 42;
    }

    T bar();
};
//-------------------------------------------------------------
int main() {
    A<int> a;             // Instantiates only A<int>
    int x = a.foo(32);    // Instantiates A<int>::foo

    // No error although A::bar is never defined

    A<float>* aptr;       // Does not instantiate A<float>
}
```

Differences between Explicit and Implicit Instantiation

Implicit instantiation

- Pro: Template can be used with any suitable type
- Pro: No unnecessary assembly is generated
- Con: Definition has to be provided in header
- Con: *User* of our templates has to compile them

Explicit instantiation

- Pro: Explicit instantiations can be compiled into library
- Pro: Definition can be encapsulated in source file
- Con: Limits usability of our templates

Usually, we do not need to explicitly instantiate our templates

Instantiation Caveats

The compiler actually generates code for instantiations

- Code is generated for *each* instantiation with different template arguments
- Conceptually, template parameters are replaced by template arguments
- If one instantiation generates 1 000 lines of assembly, 10 instantiations generate 10 000 lines of assembly
- Can substantially increase compilation time

Instantiations are generated locally for each compilation unit

- Templates are implicitly `inline`
- The same instantiation can exist in different compilation units without violating ODR

Inline vs. Out-of-Line Definition

Out-of-line definitions should be preferred even when defining class templates in headers

- Improves readability of interface
- Requires somewhat “weird” syntax

```
template <class T>
struct A {
    T value;

    A(T value);

    template <class R>
    R convert();
};
//--------------------------------------------------------------
template <class T>
A<T>::A(T value) : value(value) { }
//--------------------------------------------------------------
template <class T>
template <class R>
R A<T>::convert() { return static_cast<R>(value); }
```

Named Requirements (1)

C++ does not (yet) have features to impose restrictions on template parameters

- In theory, a programmer can try to instantiate a template with any arguments
- A template might assume certain things about its parameters (e.g. presence of a member function)
- Behavior of templates resembles duck typing
- May lead to (horrible) compile-time errors if used with incorrect template arguments

```
template <class T>
struct A {
    int bar(T t) { return t.foo(); }
};
//------------------------------------------------------------------
int main() {
    A<int> b;    // OK: A<int>::bar is not instantiated
    b.bar(42);   // ERROR: int does not have foo member
}
```

Named Requirements (2)

The C++ standard library uses *named requirements* to define expectations

- Named requirements specify operations that a type must support
- For template types, the C++ reference documentation lists named requirements that template arguments must satisfy
- The *user* of standard library templates has to ensure that requirements are met

Example: `std::vector<T>::push_back`

- In order to use `void push_back(const T& value);`, `T` must meet the requirements of CopyInsertable
- In order to use `void push_back(T&& value);`, `T` must meet the requirements of MoveInsertable

C++20 concepts will allow to check (some) named requirements at compile-time

Dependent Names (1)

Within a class template, some names may be deduced to refer to the current instantiation

- The class name itself (without template parameters)
- The name of a member of the class template
- The name of a nested class of the class template

```
template <class T>
struct A {
    struct B { };

    B* b;  // B refers to A<T>::B

    A(const A& other);  // A refers to A<T>

    void foo();
    void bar() {
        foo();  // foo refers to A<T>::foo
    }
};
```

Dependent Names (2)

Names that are members of templates are not considered to be types by default

- When using a name that is a member of a template outside of any template declaration or definition
- When using a name that is not a member of the current instantiation within a template declaration or definition
- If such a name should be considered as a type, the `typename` disambiguator has to be used

Dependent Names (3)

Example

```
struct A {
    using MemberTypeAlias = float;
};
//----------------------------------------------------------------
template <class T>
struct B {
    using AnotherMemberTypeAlias = typename T::MemberTypeAlias;
};
//----------------------------------------------------------------
int main() {
    // value has type float
    B<A>::AnotherMemberTypeAlias value = 42.0f;
}
```

Dependent Names (4)

Similar rules apply to template names within template definitions

- Any name that is not a member of the current instantiation is not considered to be a template name
- If such a name should be considered as a template name, the `template` disambiguator has to be used

```
template <class T>
struct A {
    template <class R>
    R convert(T value) { return static_cast<R>(value); }
};
//--------------------------------------------------------------
template <class T>
T foo() {
    A<int> a;

    return a.template convert<T>(42);
}
```

Reference Collapsing

Templates and type aliases may form references to references

```
template <class T>
class Foo {
    using Trref = T&&;
};

int main() {
    Foo<int&&>::Trref x; // what is the type of x?
}
```

Reference collapsing rules apply

- Rvalue reference to rvalue reference collapses to rvalue reference
- Any other combination forms an lvalue reference

Explicit Template Specialization

We may want to modify the behavior of templates for specific template arguments

- For example, a templated `find` method can employ different algorithms on arrays (binary search) vs. linked lists (linear search)

We can explicitly specialize templates to achieve this

- Define specific implementations for certain template arguments
- All template arguments can be specified (full specialization)
- Some template arguments can be specified (partial specialization)

Full Specialization

Defines a specific implementation for a full set of template arguments

- Has to appear after the declaration of the original template
- Syntax: `template <>` *declaration*
- Most types of templates can be fully specialized

```
template <class T>
class MyContainer {
    /* generic implementation */
};
//-------------------------------------------------------------
template <>
class MyContainer<long> {
    /* specific implementation */
};
//-------------------------------------------------------------
int main() {
    MyContainer<float> a;  // uses generic implementation
    MyContainer<long> b;   // uses specific implementation
}
```

Partial Specialization

Defines a specific implementation for a partial set of template arguments

- Has to appear after the declaration of the original template
- `template < parameter-list > class name < argument-list >`
- `template < parameter-list > struct name < argument-list >`
- Only class templates can be partially specialized
- Function overloads can simulate function template specialization

```
template <class C, class T>
class SearchAlgorithm {
    void find (const C& container, const T& value) {
        /* do linear search */
    }
};
//-----------------------------------------------------------------
template<class T>
class SearchAlgorithm<std::vector<T>, T> {
    void find (const std::vector<T>& container, const T& value) {
        /* do binary search */
    }
};
```

Template Argument Deduction

Some template arguments for class and function templates can be *deduced*

- All template arguments have to be known to instantiate a class or function template
- Not all template arguments have to be specified for class and function templates
- Template arguments can be omitted entirely quite frequently
- Makes it possible, for example, to use template operators

```
template <class T>
void swap(T& a, T& b);
//---------------------------------------------------------
int main() {
    int a = 0;
    int b = 42;

    swap(a, b);  // T is deduced to be int
}
```

Function Template Argument Deduction

Deduces template arguments in function calls

- Attempts to deduce the template arguments based on the types of the function arguments
- Argument deduction may fail if ambiguous types are deduced
- Highly complex set of rules (see reference documentation)

```
template <class T>
T max(const T& a, const T& b);
//------------------------------------------------------------
int main() {
    int a = 0;
    long b = 42;

    max(a, b);          // ERROR: Ambiguous deduction of T
    max(a, a);          // OK
    max<int>(a, b);     // OK
    max<long>(a, b);    // OK
}
```

Class Template Argument Deduction

Deduces class template arguments in some cases

- Declarations that also specify initialization of a variable
- new-expressions
- Attempts to deduce the template arguments based on the types of the constructor arguments

```
#include <memory>
//--------------------------------------------------------------
template <class T>
struct Foo {
    Foo(T t);
};
//--------------------------------------------------------------
int main() {
    Foo foo(12);
    std::unique_ptr ptr = make_unique<int>(42);
}
```

The auto Type (1)

The `auto` placeholder can be used to deduce the type of a variable from its initializer

- Deduction follows the same rules as function template argument deduction
- `auto` may be accompanied by the usual modifiers such as `const`, * or &
- Extremely convenient when using complex types (such as standard library iterators)

```
#include <unordered_map>
//------------------------------------------------------------
int main() {
    std::unordered_map<int, const char*> intToStringMap;

    std::unordered_map<int, const char*>::iterator it1 =
        intToStringMap.begin();  // noone wants to read this

    auto it2 = intToStringMap.begin();  // much better
}
```

The auto Type (2)

auto does not require any modifiers to work

- Can make code more error prone and hard to understand
- All known modifiers should always be added to auto

```
const int** foo();
//------------------------------------------------------------
int main() {
    // BAD:
    auto f1 = foo();          // auto is const int**
    const auto f2 = foo();    // auto is int**
    auto** f3 = foo();        // auto is const int

    // GOOD:
    const auto** f4 = foo();  // auto is int
}
```

The auto Type (3)

auto is not deduced to a reference type

- Might incur unwanted copies
- All known modifiers should always be added to auto

```
struct A {
    const A& foo() { return *this; }
};
//-----------------------------------------------------------
int main() {
    A a;
    auto a1 = a.foo();          // BAD:  auto is const A, copy
    const auto& a2 = a.foo()    // GOOD: auto is A, no copy
}
```

Structured Bindings (1)

Binds some names to subobjects or elements of the initializer

- Syntax (1): `auto [ identifier-list ] = expression;`
- Syntax (2): `auto [ identifier-list ]( expression );`
- Syntax (3): `auto [ identifier-list ]{ expression };`
- `auto` may be cv- or reference-qualified

Explanation

- The identifiers in *`identifier-list`* are bound to the subobjects or elements of the initializer
- Can bind to arrays, tuple-like types and accessible data members
- Very useful during iteration, especially over associative containers

Structured Bindings (2)

Example

```
#include <utility>
//-------------------------------------------------------------
struct Foo {
    float y;
    long z;
};
//-------------------------------------------------------------
std::pair<int, long> bar();
//-------------------------------------------------------------
int main() {
    Foo foo;
    int array[4];

    auto [a1, a2, a3, a4] = array;  // copies array, a1 - a4 refer to copy
    auto& [y, z] = foo;   // y refers to foo.y, z refers to foo.z
    auto [l, r] = bar();  // move-constructs pair p, l refers to p.first,
                          // r refers to p.second
}
```

Parameter Packs (1)

Parameter packs are template parameters that accept zero or more arguments

- Non-type: `type ... Args`
- Type: `typename|class ... Args`
- Template: `template < parameter-list > typename|class ... Args`
- Can appear in alias, class and function template parameter lists
- Templates with at least on parameter pack are called *variadic templates*

Function parameter packs

- Appears in the function parameter list of a variadic function template
- Syntax: `Args ... args`

Parameter pack expansion

- Syntax: `pattern ...`
- Expands to a comma-separated list of `pattern`s (`pattern` must contain at least one parameter pack)

Parameter Packs (2)

```
template <typename... T>
struct Tuple { };
//------------------------------------------------------------
template <typename... T>
void printTuple(const Tuple<T...>& tuple);
//------------------------------------------------------------
template <typename... T>
void printElements(const T&... args);
//------------------------------------------------------------
int main() {
    Tuple<int, int, float> tuple;

    printTuple(tuple);
    printElements(1, 2, 3, 4);
}
```

Parameter Packs (3)

Implementation of variadic templates is somewhat involved

- Most straightforward way: Tail recursion (usually optimized away)

```
#include <iostream>
//------------------------------------------------------------
void printElements() { }
//------------------------------------------------------------
template <typename Head, typename... Tail>
void printElements(const Head& head, const Tail&... tail) {
    std::cout << head;

    if constexpr (sizeof...(tail) > 0)
        std::cout << ", ";

    printElements(tail...);
}
//------------------------------------------------------------
int main() {
    printElements(1, 2, 3.0, 3.14, 4);
}
```

Fold Expressions (1)

Reduces a parameter pack over a binary operator *op*

- Syntax (1): `(` *pack op* `...` `)`
- Syntax (2): `(` `...` *op pack* `)`
- Syntax (3): `(` *pack op* `...` *op* *`init`* `)`
- Syntax (4): `(` *`init`* *op* `...` *op pack* `)`
- *pack* must be an expression that contains an unexpanded parameter pack
- *`init`* must be an expression that does not contain a parameter pack

Semantics

- $(E \circ \ldots)$ becomes $E_1 \circ (\ldots(E_{n-1} \circ E_n))$
- $(\ldots \circ E)$ becomes $((E_1 \circ E_2) \circ \ldots) \circ E_n$
- $(E \circ \ldots \circ I)$ becomes $E_1 \circ (\ldots(E_{n-1} \circ (E_n \circ I)))$
- $(I \circ \ldots \circ E)$ becomes $(((I \circ E_1) \circ E_2) \circ \ldots) \circ E_n$

Fold Expressions (2)

Enables more concise implementation of variadic templates in some cases

```
template <typename R, typename... Args>
R reduceSum(const Args&... args) {
    return (args + ...);
}
//------------------------------------------------------------
int main() {
    return reduceSum<int>(1, 2, 3, 4);  // returns 10
}
```

Concise implementations quickly become concise but extremely hard to understand

- Only used in some specialized cases

Template Metaprogramming

Templates are instantiated at *compile-time*

- Allows "programming" at compile-time (*template metaprogramming*)
- Templates are actually a Turing-complete (sub-)language
- Allows for very useful but at times very involved tricks (e.g. type traits)

```
template <unsigned N>
struct Factorial {
    static constexpr unsigned value = N * Factorial<N - 1>::value;
};
//------------------------------------------------------------
template <>
struct Factorial<0> {
    static constexpr unsigned value = 1;
};
//------------------------------------------------------------
int main() {
    return Factorial<6>::value;  // computes 6! at compile time
}
```

static_assert

The `static_assert` declaration checks assertions at compile-time

- Syntax (1): `static_assert (`*`bool-constexpr`*`)`
- Syntax (2): `static_assert (`*`bool-constexpr`*`, `*`message`*`)`
- *bool-constexpr* must be a constant expression that evaluates to `bool`
- *message* may be a string that appears as a compiler error if *bool-constexpr* is `false`

```
template <unsigned N>
class NonEmptyBuffer {
    static_assert(N > 0);
};
```

Type Traits

Type traits compute information about types at compile time

- Simple form of template metaprogramming
- E.g. `std::numeric_limits` is a type trait

```
template <typename T>
struct IsUnsigned {
    static constexpr bool value = false;
};
//------------------------------------------------------------
template <>
struct IsUnsigned <unsigned char> {
    static constexpr bool value = true;
};
/* Further specializations of IsUnsigned for all unsigned types */
//------------------------------------------------------------
template <typename T>
void foo() {
    static_assert(IsUnsigned<T>::value);
}
```

C++ provides many useful type traits (see reference documentation)