Concepts of C++ Programming

Lecture 7: Templates

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

On Complexity

One of the most important ways
to deal with complexity is to

leave it out.

Simple solutions are often better.

Type-Generic Functionality

- Some functionality is independent of specific type
  - E.g., `std::swap`, `std::unique_ptr`, ...
- Copy-paste: massive code duplication, not maintainable
- Macros: only textual replacement $\rightsquigarrow$ very limited

Templates

- Template defines family of classes/functions/type aliases/variables
- *Compile-time* parameterization on types or constants
  - On every instantiation, code compiled with respective specialization

```
template <class T, size_t N>
class array {
  T data[N];
public:
  T& operator[](size_t i) { return data[i]; }
  // ...
};
int main() {
  array<int, 12> a1; // T substituted with int; N with 12
  array<char*, 1> a2; // T substituted with char*; N with 1
}
```

Template Syntax[93]

- `template <`*`parameter-list`*`> declaration`
- Parameter list: comma-separated list of template parameters
  - `class` *`name`* (or `typename` *`name`*): Type parameter
  - *`type name`*: Non-type parameter (int, pointer, enum, lvalue reference)
  - `template <`*`parameter-list`*`> class` *`name`*: template parameter (avoid)
- Can have default parameters (similar to functions)
- Declaration:
  - `class` or `struct` (class template)
  - A function or member function (function template)
  - `using` (alias template)
  - Variable declarartion (variable template)

[93] https://en.cppreference.com/w/cpp/language/templates

Using Templates

- *template-name <parameters>*
- Results in *specialization* of template
- Sometimes, arguments can be *deduced* automatically by compiler

```
class A; // forward declaration => incomplete type
template <class T1, class T2 = unsigned, unsigned N = 4u>
class Foo { /* ... */ };

Foo<A, int, 32u> foo1;
Foo<long, A*, 12> foo2; // 12 converted implicitly to unsigned
Foo<char> foo3;
Foo<Foo<char>**, long&> foo4;
```

More Template Examples

```
template <class T>
using Storage = std::vector<T>;

template <class T>
void swap(T& a, T& b) {
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}
int f() {
  int x = 10, y = 12;
  swap<int>(x, y);
  swap(x, y); // OK, template arguments are deduced
  return x; // returns 10
}
```

More Template Examples

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

    template <class T>
    std::size_t getSize() {
        return sizeof(T);
    }
};

int f() {
    Foo::ArrayType<int> intArray;
    Foo foo;
    return foo.getSize<Foo::ArrayType<int>>();
}
```

Template Instantiation

- Function/class template itself is *not* a type/object/function
  - No machine code generated from template definition!
- Template must be *instantiated*
  - Compiler generates actual function/class for a specialization
  - Specializaton $\approx$ smart code duplication
- Explicit instantiation: explicit request
- Implicit instantiation: specialization used in context that requires complete type

Explicit Instantiation

- Force instantiation of a template specialization
- `template class` *template-name* `<`*arguments*`>;`
- `template` *ret-type name* `<`*arguments*`>(`*func parameters*`);`
- Declaration with preceding `extern`; explicit instantiations follow ODR

```
//--- A.h
template <class T>
void reallyBigA(T* t);
extern template void reallyBigA<int>(int*);

//--- A.cpp
template <class T>
void reallyBigA(T* t) { /* ... */ }
// Causes compiler to instantiate and therefore actually compile the function
template void reallyBigA<int>(int*);
```

Implicit Instantiation

- Occurs when specialization is used where complete type is required
- Members of class template are only instantiated when actually used
- Definitions must be visible for instantiation $\rightsquigarrow$ usually provided in header

```
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>
}
```

Explicit vs. Implicit Instantiation

Explicit Instantiation

- \+ Definition encapsulated in .cpp file
- \+ Can be shipped as library
- – Severely limits usability

Implicit Instantiation

- \+ Flexibly usable with any type
- – Definition must be in header
- – User of templates has to compile them

- Instantiations generated locally in each compilation unit
  - Templates are implicitly `inline`
- Compiler generates code for *each* instantiation
- Substantially increases compile time
- Usually: implicit instantiation due to flexibility

Out-of-Line Definitions

- Slightly weird syntax; sometimes preferable for interface readability

```
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); }
```

Templates: Example (1)

Quiz: What is the output of the program?

```
#include <print>
template <class T> class Foo {
  T value;
public:
  static unsigned count;
  Foo() { count++; }
};
template <class T> unsigned Foo<T>::count = 0;
int main() {
  Foo<int> foo1; Foo<long> foo2; Foo<Foo<int>> foo3;
  std::println("{}/{}", Foo<int>::count, Foo<long>::count);
  return 0;
}
```

A. (compile error) B. 1/1 C. 2/1 D. 3/3 E. 4/4

Quiz: What is problematic about this code?

```
#include <array>
#include <print>
template <class T> struct ContainerContainer {
  T t;
  ContainerContainer() {}
  size_t size() const { return t.size(); }
};
int main() {
  ContainerContainer<std::array<int, 10>> cc1;
  ContainerContainer<long> cc2;
  return cc1.size();
}
```

A. Compile error: `const std::array<...>` has no method `size()`.

B. Compile error: `std::array` is only declared, but not defined.

C. Compile error: `long` has no method `size()`.

D. There is no problem, the program exits with status 10.

Templates: Example (3)

Quiz: What is the output of the program?

```
#include <print>
#include <utility>
struct A { int a; };
int r(A&) { return 1; }
int r(A&&) { return 2; }

template <class T = void> int foo(T&& t) { return r(t); }
int main() {
  A a{12};
  std::println("{}/{}", foo(A{12}), foo(a));
  return 0;
}
```

A. (compile error) B. 1/1 C. 1/2 D. 2/1 E. 2/2

Reference Collapsing[94]

- Template types can form references to references
- But: references of references are not allowed

- `T&& &&` $\Rightarrow$ `T&&`
- `T& &&`, `T& &`, `T&& &` $\Rightarrow$ `T&`

```
template <class T> int foo(T&& t) { return r(t); }
int main() {
  A a{12};
  foo(A{12}); // T is A&& => argument type is A&&
  foo(a); // T is A& => argument type is A&
  return 0;
}
```

[94] https://en.cppreference.com/w/cpp/language/reference#Reference_collapsing

Template Argument Deduction[95]

- All template arguments must be known for instantiation
- But: not all have to be specified for class/function templates

- Based on function/constructor arguments; might fail when ambiguous
- Highly complex set of rules

```
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, T = int
  max<int>(a, b); // OK
  max<long>(a, b); // OK

  std::unique_ptr ptr = make_unique<int>(42); // OK, T = int
}
```

[95] https://en.cppreference.com/w/cpp/language/template_argument_deduction

Templates: Argument Deduction (1)

Quiz: What is the output of the program?

Assume 4-byte integers and 8-byte pointers.

```
#include <print>
template <class T> size_t f(T* x) { return sizeof(*x); }
int main() {
  int* x = nullptr;
  std::println("{}", f(x));
  return 0;
}
```

A. (compile error) B. 4 C. 8 D. (undefined behavior)

Templates: Argument Deduction (2)

Quiz: What is the output of the program?

Assume 4-byte integers and 8-byte pointers.

```
#include <print>
template <class T> size_t f(T* x) { return sizeof(*x); }

int main() {
  std::println("{}", f(nullptr));
  return 0;
}
```

A. (compile error) B. 0 C. (some positive integer) D. (undefined behavior)

auto Type[96]

- auto placeholder: deduce variable type from initializer
- Can (should!) be accompanied by usual modifiers (e.g. const, *, &)
  - auto not deduced to reference type, might cause unwanted copies

```
#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
}
```

[96] https://en.cppreference.com/w/cpp/language/auto

auto Type – Examples

```
const int** foo();
struct A {
  const A& foo() { return *this; }
};

void bar() {
  auto f1 = foo(); // BAD: auto is const int**
  const auto f2 = foo(); // BAD: auto is const int**, f2 is const
  auto** f3 = foo(); // BAD: auto is const int

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

  A a;
  auto a1 = a.foo(); // BAD: auto is const A, copy
  const auto& a2 = a.foo(); // GOOD: auto is A, no copy
}
```

Parameter Packs[97]

- Parameter pack: accept zero or more template arguments
- Type: `class ... Args` / Non-type: `type ... Args`

- Function parameters: `Args ... args`
- Appears in function parameter list of a variadic function template

- Parameter pack expression: `pattern...`
- Expands comma-separated list of `pattern`, which contains a parameter pack
  - E.g., `&args...` expands to `&arg1, &arg2, &arg3`

[97] https://en.cppreference.com/w/cpp/language/parameter_pack

Parameter Packs – Example

- Implementation somewhat difficult to write
- Straightforward way: tail recursion

```
#include <print>
void printElements() { } // recursion end

template <typename Head, typename... Tail>
void printElements(const Head& head, const Tail&... tail) {
  std::print("{}", head);
  if (sizeof...(tail) > 0) // number of elements in Tail
    std::print(",␣");
  printElements(tail...);
}

int main() {
  // Output: "1, 2, 3, 3.14, hello, 4"
  printElements(1, 2, 3.0, 3.14, "hello", 4);
}
```

Fold Expressions[98]

- Reduce parameter pack over binary operator
- `( pack op ... )` becomes $E_1 \circ (... \circ (E_{n-1} \circ E_n))$
- `( ... op pack )` becomes $((E_1 \circ E_2) \circ ...) \circ E_n$
- `( pack op ... op init )` becomes $E_1 \circ (... \circ (E_{n-1} \circ (E_n \circ I)))$
- `( init op ... op pack )` becomes $(((I \circ E_1) \circ E_2) \circ ...) \circ E_n$

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

[98] https://en.cppreference.com/w/cpp/language/fold

Dependent Names (1)[99]

- In class template: class name and members refer to current instantiation

```
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
  }
};
```

[99] https://en.cppreference.com/w/cpp/language/dependent_name

Dependent Names (2)

- Names dependent on template parameter types that are not members of the current instantiation are *not* considered as types by default

$\Rightarrow$ Needs `typename` disambiguator

- Can be omitted in some contexts, see reference

```
struct A {
  using MemberTypeAlias = float;
};
template <class T> struct B {
  using AnotherAlias = T::MemberTypeAlias; // no disambiguator required
  typename T::MemberTypeAlias* ptr; // disambiguator required
};
int main() {
  // outside template declaration => no disambiguator required
  B<A>::AnotherAlias value = 42.0f;
}
```

Dependent Names (3)

- Similar rules apply for template names inside template definitions
- Name that is not member of current instantiation *not* considered as template

$\Rightarrow$ Needs `template` disambiguator

```
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);
}
```

Dependent Names: Motivation

Quiz: What is the output of the following program?

Assume 4-byte integers and 8-byte pointers.

```
#include <print>
template <class T> int fn(bool param, int a) {
  if (param) {
    T::member* a; return sizeof(a);
  }
  return 0;
}
struct S1 { static const int member = 5; };
struct S2 { using member = int; };
int main() {
  std::println("{}/{}", fn<S1>(true, 1), fn<S2>(true, 1));
}
```

A. (compile error) B. 4/1 C. 4/4 D. 4/8 E. 8/8

Explicit Specialization

- Sometimes, we want a different behavior for specific template arguments
- Example: different algorithm for a templated `find` function,
  e.g., binary search for array, linear search for linked
- Example: optimized storage for `bool`

- Explicit specialization: provide specific implementation for certain arguments
- Full specialization: all arguments are specified
- Partial specialization: some arguments are specified

Full Specialization[100]

- `template <>` *declaration*
- Must come after the original template declaration

```
template <class T> class MyContainer {
    /* generic implementation */
};

template <> class MyContainer<bool> {
    /* specific implementation */
};

int main() {
    MyContainer<float> a; // uses generic implementation
    MyContainer<bool> b; // uses specific implementation
}
```

[100] https://en.cppreference.com/w/cpp/language/template_specialization

Partial Specialization[101]

- `template <parameter-list> class` *name* `<`*argument-list*`>`
- Must come after the original template declaration
- Only class templates can be partially specialized

```
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 */
    }
};
```

[101] https://en.cppreference.com/w/cpp/language/partial_specialization

Specializations: Example

Quiz: What is the output of the following program?

```
#include <print>
template <class T> int foo(T) { return 1; }
template <> int foo<int*>(int*) { return 2; }

int bar(const int& l) { return foo(&l); }

int main() {
  std::println("{}", bar(10));
}
```

A. (compile error) B. 1 C. 2 D. (undefined behavior)

Traits

- Trait classes: provide generic way to access information about types

```
template <class> class GraphTraits {};

template <class GraphT>
std::vector<typename GraphTraits<GraphT>::NodePtr>
    traverseGraph(const GraphT& graph) {
  using NodePtr = GraphTraits<GraphT>::NodePtr;
  NodePtr start = GraphTraits<GraphT>::getFirstNode(graph);
  // ...
}

template <> // Specialization of traits class for specific graph
class GraphTraits<MyGraph> {
  using NodePtr = MyNode*;
  NodePtr getFirstNode(const MyGraph& graph) { /* ... */ }
  // ...
}
```

Type Traits (1)[102]

- Specialization useful for querying information about types themselves (traits)

```
// NB: use std::is_same instead
// Base case
template <class T1, class T2>
struct is_same { static constexpr bool value = false; };

// Specialization for case where both types are the same
template <class T>
struct is_same<T, T> { static constexpr bool value = true; };

#include <cstdint>
static_assert(is_same<int, long>::value == false);
static_assert(is_same<int, const int>::value == true);
static_assert(is_same<int, int32_t>::value == true);
```

[102] https://en.cppreference.com/w/cpp/header/type_traits

Type Traits (2)

```
// NB: use std::remove_reference instead
template<class T> struct remove_reference { using type = T; };
template<class T> struct remove_reference<T&> { using type = T; };
template<class T> struct remove_reference<T&&> { using type = T; };

template<class T>
using remove_reference_t = typename remove_reference<T>::type;

static_assert(std::is_same<int, remove_reference_t<int>>::value);
static_assert(std::is_same<int, remove_reference_t<int&>>::value);
static_assert(std::is_same<int, remove_reference_t<int&&>>::value);
static_assert(std::is_same<int*, remove_reference_t<int*>>::value);
```

Implementation of `std::move`

- With type traits, `std::move` becomes easily implementable

```
template <class T>
std::remove_reference_t<T>&& move(T&& t) {
  return static_cast<std::remove_reference_t<T>&&>(t);
}
```

Type Constraints[103]

- Template might assume certain things about parameters
  - E.g., member function, copyable, moveable
  - Effectively "duck typing"
- Might lead to (horrible) compilation errors when used with incorrect arguments

- Constraint: requirements on template arguments
- Concept: named set of requirements

[103] https://en.cppreference.com/w/cpp/language/constraints

Type Constraints – requires Clause[104]

- `requires <`*`constant-expression`*`>` – apply constraint to template

```
#include <concepts>

template <class T> requires true // useless, but valid
void fn() {}

template <class T> requires std::floating_point<T>
T fdiv1(T a, T b) {
  return a / b;
}

// template <Concept X> == template <class X> requires Concept<X>
template <std::floating_point T>
T fdiv2(T a, T b) {
  return a / b;
}
```

[104] https://en.cppreference.com/w/cpp/language/constraints#Requires_clauses

Type Constraints and Concepts

- Can also be combined with `&&` and `||`
- Greatly improve safety: fewer implicit assumptions
- Improve quality of error messages

- We will revisit constraints and concepts later in the lecture

Templates – Summary

- Templates enable compile-time specialization of classes/functions/types
- Instantiation either explicit or implicit on use
  - Implicit instantiation typically used in practice
- Template arguments sometimes can be deduced from (constructor) arguments
- Names dependent on parameter types may need type disambiguator
- Template specialization: different implementation for specific arguments
- Trait classes use specialization to provide generic interface
- Constraints can express requirements on template arguments
- `auto` exposes type deduction to variable declarations

Templates – Questions

- When is a template instantiated?
- What is the benefit of explicit instantiation?
- Where to template definitions go?
- How to provide an out-of-line definition for a method in a template class?
- When is a `typename` disambiguator required?
- Why is using `auto` without modifiers sometime problematic?
- What are use cases of template specialization?
- How to express requirements on type template arguments?