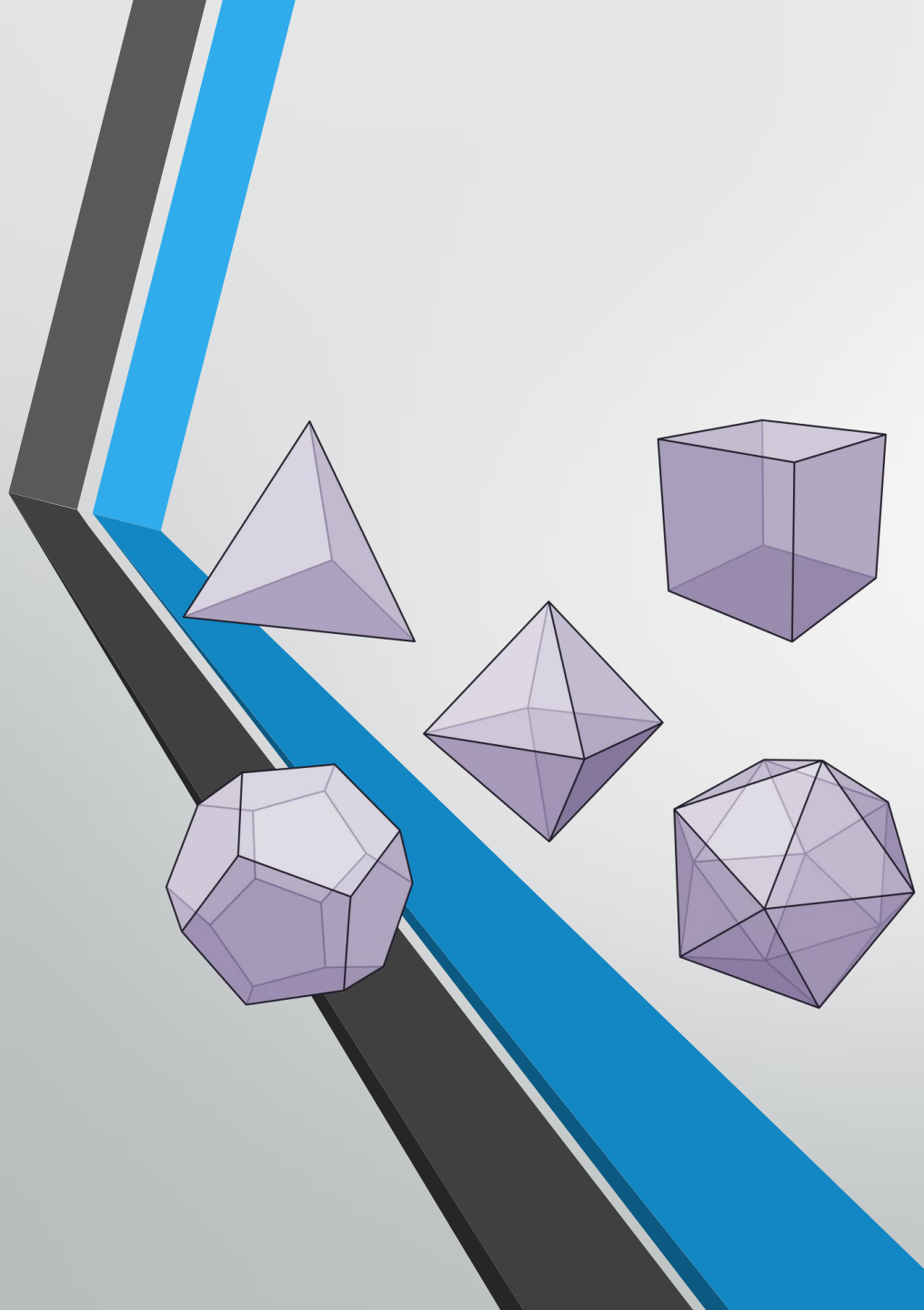

Advanced C++ Programming

Concepts & Constraints

Preliminaries

Overview & Goals

- This chapter introduces **concepts** and **constraints**
- These relate directly to our previous discussion of generic programming and template metaprogramming
- The primary goals of this feature are to
  - Improve static checking of generic code
  - Thereby allow for improved compiler error messages
  - Template overload and specialization selection (without metaprogramming “hacks”)

Some Background

- Concepts are a C++20 feature
- Implementation currently available in the latest versions of GCC, Clang, MSVC (GCC 10.2, Clang 11, MSVC 16.8)
  - Not necessarily 100% complete / bug-free at this point
- Full concept integration for the STL will be added in a future standard

Basic Usage & Results

Basic Example

- Before getting into any syntactic or semantic details,
  let's look at a really basic example: 07_01_basic_sample.cpp

- We can *constrain* our template argument to match a given concept
  - How does this help us?

Compiler Results (gcc 10.2)

No Concepts

```
<source>: In instantiation of 'void fun(T) [with T = meow]':
<source>:27:12:   required from here
<source>:21:16: error: could not convert '<brace-enclosed initializer list>()' from '<brace-
enclosed initializer list>' to 'std::__hash_enum<meow, false>'
   21 |   std::hash<T>{}(arg);
      |   ~~~~~~~~~~~~~~^~~~
      |                 |
      |                 <brace-enclosed initializer list>
<source>:21:16: error: use of deleted function 'std::hash<meow>::~hash()'
In file included from /opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/string_view:4
                 from /opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/bits/basic_st
                 from /opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/string:55,
                 from <source>:3:
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/bits/functional_hash.h:101:12: note
'std::hash<meow>::~hash()' is implicitly deleted because the default definition would be
formed:
  101 |     struct hash : __hash_enum<_Tp>
      |            ^~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/bits/functional_hash.h:101:12: error
'std::__hash_enum<_Tp, <anonymous> >::~__hash_enum() [with _Tp = meow; bool <anonymous>
is private within this context
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/bits/functional_hash.h:83:7: note:
private here
   83 |       ~__hash_enum();
      |       ^
Compiler returned: 1
```

Concepts

```
<source>: In function 'int main()':
<source>:27:12: error: use of function 'void fun(T) [with T = meow]' with unsatisfied constraints
   27 |   fun(meow{}); // Error: meow does not satisfy Hashable
      |            ^
<source>:19:6: note: declared here
   19 | void fun(T arg) {
      |      ^~~
<source>:19:6: note: constraints not satisfied
<source>: In instantiation of 'void fun(T) [with T = meow]':
<source>:27:12:   required from here
<source>:11:9:   required for the satisfaction of 'Hashable<T>' [with T = meow]
<source>:11:20:   in requirements with 'T a' [with _Tp = meow; T = meow]
<source>:12:16: note: the required expression 'std::hash<_Tp>{}(a)' is invalid
   12 |   std::hash<T>{}(a);
      |   ~~~~~~~~~~~~~~^~
cc1plus: note: set '-fconcepts-diagnostics-depth=' to at least 2 for more detail
Compiler returned: 1
```

Compiler Results (Clang 11)

No Concepts

```
<source>:21:15: error: temporary of type '__hash_enum<meow>' has private destructor
       std::hash<T>{}(arg);
                  ^
<source>:27:2: note: in instantiation of function template specialization 'fun<meow>' requested
here
       fun(meow{}); // Error: meow does not satisfy Hashable
       ^
/opt/compiler-explorer/gcc-10.2.0/lib/gcc/x86_64-linux-gnu/10.2.0/../../../../include/c
/bits/functional_hash.h:83:7: note: declared private here
      ~__hash_enum();
      ^
<source>:21:15: error: no matching constructor for initialization of '__hash_enum<meow>
       std::hash<T>{}(arg);
                  ^
/opt/compiler-explorer/gcc-10.2.0/lib/gcc/x86_64-linux-gnu/10.2.0/../../../../include/c
/bits/functional_hash.h:82:7: note: candidate constructor not viable: requires 1 argume
were provided
      __hash_enum(__hash_enum&&);
      ^
/opt/compiler-explorer/gcc-10.2.0/lib/gcc/x86_64-linux-gnu/10.2.0/../../../../include/c
/bits/functional_hash.h:78:12: note: candidate constructor (the implicit copy construct
viable: requires 1 argument, but 0 were provided
    struct __hash_enum
           ^
2 errors generated.
Compiler returned: 1
```

Concepts

```
<source>:27:2: error: no matching function for call to 'fun'
       fun(meow{}); // Error: meow does not satisfy Hashable
       ^~~
<source>:19:6: note: candidate template ignored: constraints not satisfied [with T = meow]
void fun(T arg) {
     ^
<source>:18:11: note: because 'meow' does not satisfy 'Hashable'
       requires Hashable<T>
                ^
<source>:12:15: note: because 'std::hash<T>({})(a)' would be invalid: temporary of type
'__hash_enum<meow>' has private destructor
       std::hash<T>{}(a);
                  ^
1 error generated.
Compiler returned: 1
```

Compiler Results (MSVC 19.28)

No Concepts

```
example.cpp
<source>(21): warning C4834: discarding return value of function with 'nodiscard' attribute
<source>(26): note: see reference to function template instantiation 'void fun<std::string>(T)'
being compiled
        with
        [
            T=std::string
        ]
<source>(21): error C2512: 'std::hash<T>': no appropriate
        with
        [
            T=meow
        ]
<source>(21): note: Invalid aggregate initialization
<source>(27): note: see reference to function template instantiation 'void fun<meow>(T)' being
compiled
        with
        [
            T=meow
        ]
Compiler returned: 2
```

Concepts

```
example.cpp
<source>(27): error C2672: 'fun': no matching overloaded function found
<source>(27): error C7602: 'fun': the associated constraints are not satisfied
<source>(19): note: see declaration of 'fun'
Compiler returned: 2
```

Syntactic Options

- There is a more terse way to specifiy template parameters constrained by a single concept
  - This is easier to read and usually preferable for the basic case
- It’s also possible to specify constraints after the function signature
- Examples here: **07_02_syntax_options.cpp**

Type Constraints on **`auto`**

- You can apply Concepts on auto to (partially) constrain the types it accepts
- This also works in function declarations!
  - Even terser syntax for simple cases
  - Identical to lambda syntax
- Examples here: **07_03_constrained_auto.cpp**

`requires`

Clauses and Expressions

`requires` Clause

```
// can appear as the last element of a function declarator
template <typename T>
void f(T&&) requires Hashable<T>;

// or right after a template parameter list
template <typename T>
requires Hashable<T>
```

- Any primary expression of compile-time evaluated **bool** type is allowed
  - E.g. **`requires true`**
- But the intent is for a **named concept**
  or conjunctions/disjunctions of concepts to be used

`requires` Expression

- The same keyword is also used to start a *requires-expression*
- This is an expression of type **`bool`**,
  which is intended to be used in constraint definitions
- Its value is **`true`** if all constraints are satisfied, **`false`** otherwise

`requires` Expression

- Two syntactic forms:
- **`requires`** `{` *`requirement-seq`* `}`
- **`requires`** `(` *`parameter-list`* `)` `{` *`requirement-seq`* `}`

- Let's look at some examples
  `07_04_requires_expression.cpp`

Requirements

- A requirements sequence can contain 4 kinds of requirements:

Simple Requirements Check that arbitrary (unevaluated) expression is valid.	**Type Requirements** Check that the named type is valid (e.g. check if nested type exists).
Compound Requirements Check the return type and semantic constraints on an expression.	**Nested Requirements** Check additional constraints in a local context.

`07_05_requirements.cpp`

Overload Selection using Concepts

Practical Example

- Remember our dispatch challenge?

  (from the Metaprogramming lecture)

- Let's see what we can do with concepts!

  `07_06_dispatch.cpp`

- Not only is the syntax much clearer,
  it's also more specific and we get better errors!

Underlying Mechanisms

- Just like with templates, there is an underlying mechanism which translates our intuition into language rules
- In this case, we want to define some sets of constraints as *at least as constrained* or *more constrained* than others
- We need a **partial order on constraints**

Partial Ordering of Constraints

- First step is **normalizing** constraints into a sequence of conjunctions and disjunctions of atomic constraints
- To check if P is more constrained than Q (P ***subsumes*** Q):
  1. Convert P to disjunctive normal form and Q to conjunctive normal form
  2. Check that every disjunctive clause in P subsumes every conjunctive clause in Q

     A disjunctive clause subsumes a conjunctive clause iff there is an atomic constraint U in the disjunctive clause and an atomic constraint V in the conjunctive clause such that U subsumes V.

     An atomic constraint U subsumes an atomic constraint V if and only if they are identical. (Types and expressions are not analyzed for equivalence: N > 0 does not subsume N >= 0)

`07_07_partial_order.cpp`

Conclusion

Summary

- Concepts and Constraints allow us to
  - **Specify/constrain categories of types** that our templates should operate on
  - Select our preferred overload and **resolve ambiguity**
  - Do both of those things with much **better error reporting** than previous options
- Language Principles Required
  - Partial Ordering of Constraints