COMP6771
Advanced C++ Programming

9.1 Metaprogramming

In this Lecture

What?

- Template Metaprogramming
- `constexpr` pt. 2
- Perfect forwarding
- Concepts

Why?

- C++'s compile-programming feature set is unique.
- Many production codebases which use templates also use metaprogramming.

Metaprogramming in C++

- Metaprogramming:
  - "a programming technique in which computer programs have the ability to treat other programs as their data".
- Historically, templates were used to do metaprogramming in C++ (**T**emplate **M**eta-**P**rogramming).
  - Templates are *Turing complete*.
  - C++ TMP uses template instantiation to drive compile-time evaluation.
- Modern C++ metaprogramming uses more than just TMP.
- Metaprogramming turns the compiler into a code generator:

Template Metaprogramming (TMP)

- TMP is built off of **SFINAE** and **partial/explicit specialisation**.
- **S**ubstition **F**ailure **Is** **N**ot **A**n **E**rror (SFINAE) is used to manage overload sets.
- Specialisation is used to implement type traits and type transformations.

```cpp
// An example of using specialisation to implement
// a compile-time "if" that can be used where a type
// is expected (such as when making a type alias).


// Primary template: assume B is "true"
template <bool B, typename T, typename F>
struct if_t { using type = T; }

// Partial Specialisation: used when B is "false"
template <typename T, typename F>
struct if_t<false, T, F> { using type = F; };

int main() {
  using int16_t = if_t<sizeof(int) == 2, int, short>::type;

  return int16_t{42};
}
```

SFINAE: Function Templates

Substitution Failure Is Not An Error

- If substitution of a template argument (whether given or deduced) into a template parameter causes a function template to be ill-formed, it is **silently discarded** and is *not* a compile error!
- Can be used to "remove" function templates from being considered during overload resolution.

```
#include <type_traits>

// When "B" is true, maybe::type exists and is T.
// When "B" is false... there is no type!
template <bool B, typename T> struct maybe { using type = T; };
template <typename T> struct maybe<false, T> {};

template <bool B, typename T>
using maybe_t = typename maybe<B, T>::type;

template <typename T>
// For non-integral types, there is no return type.
// That is ill-formed, so this overload is silently discarded
maybe_t<std::is_integral_v<T>, T>
secret_algorithm(T i) { return i * 2; }

template <typename T>
// For non-floating point types, there is no return type.
// That is ill-formed, so this overload is silently discarded
maybe_t<std::is_floating_point_v<T>, T>
secret_algorithm(T fp) { return fp * T{3.14}; }

int main() {
  int secret_int = secret_algorithm(6771);
  float secret_float = secret_algorithm(3.14f);
}
// Here, we don't have two overloads of "secret_algorithm";
// For "int" vs. "float", we explicitly remove the other
// overload from the resolution set using SFINAE!
```

SFINAE: Types

Substitution Failure Is Not An Error

- If a template parameter is used to form a type, or access a member of type, and that use would cause the code to be ill-formed, that code is *silently discarded*.
- This happens when:
  - Attempting to form an array with length 0, or array to void, or any invalid array.
  - Attempting to use a member of a type in an inappropriate context (e.g., using a non-type where a type is required).
  - Etc. Full list of cases here.

```
template <typename T>
class is_class {
  using yes = char[3];
  using no = char[2];

  template <typename C>
  // selected if C is a class type.
  // this is because this member template
  // parameter accepts a pointer to a member function.
  // this is only valid for class-types.
  static yes &test(int C::*);
  template<typename C>

  // if C is not a class type,
  // this overload is selected.
  static no &test(...);

public:
  // the "test" is to see if "T" is a class-type.
  // if it is, then the return type of "test()" will be
  // yes. we can check the size of the return type to
  // determine which overload was selected.
static constexpr auto value =
                sizeof(test<T>(nullptr)) == sizeof(yes);
};
```

SFINAE: Expressions

Substitution Failure Is Not An Error

- NEW in C++11: expression SFINAE.
  - The following expression errors are SFINAE errors.
- Ill-formed expressions used in a template parameter type.
- Ill-formed expressions used in a function type.
- These are not errors but are *silently discarded.*

```cpp
template<int I>
void ex(char(&array1)[I % 2 == 0]) {} /* @ */

template<int I>
void ex(char(&array2)[I % 2 == 1]) {} /* $ */

// When "I" is even, (@) is selected.
// When "I" is odd, ($) is selected.
//
// This is because arrays of length 0 are ill-formed.
// When "I" is even, "I % 2 == 0" is true
//   -> this boolean is converted to int{1}.
//   -> "array1"'s type becomes char(&)[1].
//   whereas "array2"'s type becomes char(&)[0]
// which is ill-formed.
//
// Likewise, when "I" is odd, "I % 2 == 1" is true.
//   -> this boolean is converted to int{1}.
//   -> "array2"'s type becomes char(&)[1].
//   whereas "array1"'s type becomes char(&)[0].
```

Type Traits

- Type traits are a way to ask questions about a type at compile-time.
  - Is T a reference type?
  - What is the rank of T? (non-array is 0, otherwise the number of [ ] in the type)
- Implemented as:
  - A primary struct template that answers the question in general.
  - Specialisations that answer the question specifically.
  - Questions for type use a type member called `type`.
  - Boolean questions use a `static constexpr` boolean called `value`.
- Why?
  - *Extremely* useful for writing optimised generic code.
  - Allow explicit overload set management when writing function templates.

```cpp
// Here are an assortment of some useful type traits.

template <typename T>
// in general, T is not a reference
struct is_reference {
  static constexpr auto value = false;
};

template <typename T>
struct is_reference<T&> { // but T& is!
  static constexpr auto value = true;
};

template <typename T>
struct is_reference<T&&> { // and so is T&&!
  static constexpr auto value = true;
};

template <typename T>
struct rank_of { // in general, T is rank 0.
  static constexpr auto value = 0;
};

template <typename T>
// but arrays have rank 1!
// T could have multiple []: recursively apply the type trait.
struct rank_of<T[]> {
  static constexpr auto value = 1 + rank_of<T>::value;
};
```

Type Transformations

- Type transformations allow modification of an existing type.
  - Implemented similar to type traits.
- Can be used to generically add type modifiers:
  - such as `const`, `volatile`, &, *, etc.
- Can also be used to *remove* modifiers from a type.

```
// Some examples of basic type transformations

template <typename T>
// in general, assume T is not const.
// nothing to remove
struct rm_const { using type = T; };

template <typename T>
// but when T is const...
// Unshell the const from T
// and use the raw type as the result
struct rm_const<const T> { using type = T; };

template <typename T>
// can always add a & to a type (unless void).
struct add_lvalue { using type = T&; };

template <typename T>
// can always add a * to a type.
struct add_pointer { using type = T*; };
```

Standard Type Traits

- The Standard Library has many useful type traits in the <type_traits> header.
- Where possible, you should always try to use or reuse standard type traits.
  - When making your own type traits, it is generally a good idea to simply compose existing standard ones.

```cpp
#include <type_traits>

// We could do this...
template <typename T>
struct is_int_pointer {
  static constexpr auto value = false;
};

template <>
struct is_int_pointer<int *> {
  static constexpr auto value = true;
};

// but why do that when you have the Standard Library?
// Here, we inherit either from true_type or false_type
// depending on if "T" is an "int*" or not.
template <typename T>
struct is_int_pointer : std::conditional_t<
  std::is_same_v<T, int *>,
  std::true_type, // std::true_type::value == true.
  std::false_type // std::true_type::value == false.
>
{};
```

Shortened Type Traits

- NEW! In C++14, you can refer to the Standard Library's type-based traits through a shortened alias.
- NEW! In C++17, you can refer to the Standard Library's boolean-based traits through a shortened alias.

```
#include <type_traits>

// should be int
using return_type =
                std::invoke_result_t<int(char), char>;

// this replaces
// using return_type =
// typename std::invoke_result<int(char), char>::type;

// should be true.
constexpr bool was_it_an_int =
                     std::is_same_v<return_type, int>;

// this replaces
// constexpr bool =
//              std::is_same<return_type, int>::value;
```

TMP: Just a Sample

- What has been shown is just a sample of what can be done with TMP.
- Depending on how clever you want to be, it can become much more difficult.
  - Most compiler bugs are found within its template machinery.
- Some resources to see more examples of TMP:
  - Dr. Walter E. Brown's talk at CppCon 2014 (part 1 & part 2)
  - Jody Hagins talk on type traits
  - An implementation of named tuples through the (ab)use of templates in C++.

UNSW SYDNEY

decltype

- Semantic equivalent of a "typeof" operator.
  - Type deduced from expression.
- **Rule 1**:
  - If expression **e** is any of:
    - variable in local scope
    - variable in namespace scope
    - static member variable
    - function parameters
  - then the result is of type `T`
- **Rule 2**: if **e** is an lvalue, result is `T&`.
- **Rule 3**: if **e** is an xvalue, result is `T&&`.
- **Rule 4**: if **e** is a prvalue, result is `T`.
  - xvalue/prvalue are forms of rvalues.
  - We do not require you to know this.
- Non-simplified set of rules can be found here.

```cpp
// Just a lowly const int
const int i = 0;

// const int - needs to be initialised
decltype(i) x = 0;

// int& - needs to be initialised
// parenthesised expressions always are lvalues
decltype((i)) j = i;

// What is the result of j + i? int!
// b is of type "int" (currently uninitialised)
decltype(j+i) b;

const int *ptr = nullptr;

// Result of *ptr is const int&
// hence c is const int& so must be initalized
decltype(*ptr) c = i;

// int - prvalue
decltype(5) z = 6771;
```

Determining Return Types (Problem)

- Ever wondered what the advantage is of declaring a function using `auto` syntax?
- Consider `foo()` to the right.
  - We want the return type to be the result type of `t + u`.
  - Using `decltype`, this may be achievable!
- Unfortunately, `foo2()` has issues too...
  - `t` and `u` are not in scope yet by the time they are needed!

```
template <typename T, typename U>
??? foo(const T &t, const U &u) {
  return t + u;
}


template <typename T, typename U>
decltype(t + u) foo2(const T &t, const U &u) {
  return t + u;
}

// ERROR! t and u have not been declared yet!
```

Determining Return Types (Solution)

- If the decltype statement was *after* the variable declarations, then it would be fine.
- This is what `auto`-function syntax allows us to do.
  - Once this was realised, the `decltype` return type became optional.
  - Usually, deduce the return type from the function body.
  - But if the deduced type is not what is required (e.g., an implicit conversion is desired), can fall back on `decltype`.

```
#include <iostream>

template <typename T, typename U>
// Here, instead of letting the compiler deduce
// the type of t + u (which should be "int")
// we say the return type should be the result
// of a narrowing conversion from t to double
// and then adding u to that double.
auto foo(T t, U u) -> decltype(double(t) + u) {
  return t + u;
}

int main() {
  std::cout << foo(42, 6771) << std::endl;
}
```

decltype(auto)

- C++ has two sets of deduction rules.
- `auto` deduction rules:
  - Same as template argument deduction rules.
  - Also called object rules, since auto never deduces references or top-level `const`.
- `decltype` rules:
  - Will deduce references and top-level `const`.
  - Parenthesised expressions (e.g. `decltype((x))`) return lvalue references.
- What if you want both?
  - Let the compiler deduce the return object type with `auto`...
  - But then preserve its value category with `decltype`!
  - Mainly used with function return types only.

```cpp
#include <iostream>
#include <type_traits>

int a = 0;

int main() {
  // deduce the type of "a" as an int object
  // but it is being used as an lvalue
  // (note the parentheses)
  // so altogether: i is int&!
  decltype(auto) i = (a);

  // deduce the type of 3.14f as a float object.
  // it is a pure rvalue
  // (note: it is a float literal)
  // so altogether: j is just a float!
  decltype(auto) j = 3.14f;

  std::cout << std::boolalpha;
  std::cout << std::is_same_v<decltype(i), int&>  << ' '
            << std::is_same_v<decltype(j), float> << '\n';
}

// Output: true true
```

Unevaluated Contexts

- C++ has *eager evaluation*.
  - In an expression, the operands are evaluated at runtime for their effects.
  - But the result type of the expression is already known at compile-time.
- An **unevaluated context** is when:
  - Operands are not evaluated -> no runtime effects.
  - Only the statically-known types of expressions are used to further calculate types.
- Some unevaluated contexts:
  - When using `decltype()`
  - When using `noexcept()`
    - `noexcept` is equivalent to `noexcept(true)`
  - When using `sizeof(variable)` rather than `sizeof(type)`
  - Full list

decltype & std::declval<T>

- A common metaprogramming pattern is using `std::declval` with `decltype`.
- `std::declval` is a function template *declaration* that:
  - Returns an rvalue reference to `T`.
  - Let's one use the member functions of `T` with `decltype` without needing to worry about constructors.
- `std::declval<T>` can only be used in unevaluated contexts.
  - Otherwise, a defintion would be required.

```cpp
// How to test the return type of a method is correct
#include <iostream>
#include <type_traits>
#include <utility>

class Integer {
public:
  /* Other implementation */
  const int &i() const;
private:
  int i_;
};

int main() {
  auto is_correct = std::is_same_v<decltype(
    // use std::declval to "get" a const Integer&&
    // then, we can inspect its methods in decltype!
    std::declval<const Integer>().i()
  ), const int&>;
  std::cout << std::boolalpha << is_correct << "\n";
}

// Output: true
```

`constexpr` Revisited

- We have already seen some uses of `constexpr` when defining compile-time calculable variables.
- Since C++11, the use cases of `constexpr` have grown:
  - `constexpr` variables
  - `constexpr` if
  - `constexpr` functions.
- Let's explore `constexpr` more in-depth.

`constexpr` Variables: Revision

- A `constexpr` variable is a variable whose value is calculable at compile-time.
- Unless there is code which requires the existence of it, a `constexpr` variable does not exist at runtime.
  - Its value is hardcoded into the final executable by the compiler.
- Provides benefits over macros for global constants.
  - Is scoped and code will not compile if there is a name conflict.
  - Is type-checked by the compiler.

```
constexpr int N = 4;

int get_int(); // defined elsewhere

int main() {
  const int M = get_int();

  // not OK: M not known until runtime
  int arr1[M] = {0};

  // OK: N is a constexpr variable
  int arr2[N] = {0};
}
```

`if constexpr`

Read "`constexpr if`", written as "`if constexpr`"

- NEW! in C++17: a compile-time if-statement.
  - Can be used anywhere a regular if-statement is usable.
- The condition is evaluated at compile-time and only **one** of the branches is compiled into the final binary.
  - The other branch is discarded.
- Both branches are checked for correct syntax, but only the taken branch is checked for semantics errors.

```cpp
#include <type_traits>

template <typename T>
auto value_or_deref(const T t) {
  // Check if T is a pointer.
  // If it is, dereference t.
  if constexpr (std::is_pointer_v<T>) {
    return *t;
  } else {
    // otherwise, return t directly.
    // For non-pointers, only this branch
    // appears in the final executable!
    return t;
  }
}

template <typename T>
auto compile_error(T) {
  // It is illegal to dereference non-pointers.
  // This kind of error is caught regardless of
  // which branch would be taken at compile-time.
  if constexpr (sizeof(T) == 4) {
    int i = 0;
    return *i;
  } else {
    float f = 0;
    return *f;
  }
}
```

`constexpr` Functions

- A `constexpr` function is a function that *may* be called and computed at compile-time .
  - Happens when the arguments to the function are also `constexpr`.
- A `constexpr` function body can contain:
  - Variable definitions calculable at compile-time.
  - Loops.
  - Calls to other `constexpr` functions.
  - Calculations involving variables calculable at compile-time.
  - Complete list [here](#).
  - If any of the above conditions are not met, the function will be called at runtime.
- The return value of a `constexpr` function can be used to initialise `constexpr` variables.

```
int get_int();

constexpr int factorial(int n) {
  int res = 1;
  while (n > 1) res *= n--;
  return res;
}

int main() {
  // Will be calculated at compile-time because
  // all arguments to factorial are constexpr.
  int arr1[factorial(1)] = {factorial(4)};

  // Error: get_int() is not constexpr
  // so factorial will be called at runtime.
  // cannot initialise a constexpr variable
  // with the return value from a function
  // that is called at runtime.
  constexpr int i = factorial(get_int());
}
```

`consteval` Functions

- NEW! in C++20: `consteval` functions.
- Same rules as a `constexpr` function, except it is guaranteed to be called at compile-time.

```cpp
int get_int();

consteval int factorial(int n) {
  int res = 1;
  while (n > 1) res *= n--;
  return res;
}

int main() {
  // Guaranteed to be called at compile time.
  int arr1[factorial(1)] = {factorial(4)};

  // Error: get_int() cannot be used as an argument
  // to a consteval function because it is not
  // a constant expression.
  constexpr int i = factorial(get_int());
}
```

Literal Types

- A literal type is a type that is initialisable at compile-time.
  - All fundamental types are literal types.
- User-defined types can also be literal types.
  - You must define at least one `constexpr/consteval` constructor.
  - Any base classes must also be literal.
    - Virtual base classes are not allowed.
- Literal types can also be used in `constexpr/consteval` functions.

```cpp
#include <cmath>
#include <iostream>

class point2d {
public:
  constexpr point2d() noexcept = default;
  constexpr point2d(double x, double y) noexcept
          : x_{x}, y_{y} {}

  constexpr double x() const noexcept { return x_; }
  constexpr double y() const noexcept { return y_; }

private:
  double x_;
  double y_;
};

consteval
double distance(const point2d &p, const point2d &q) {
  return std::sqrt(p.x() * q.x() + p.y() * q.y());
}

int main() {
  // Will print 5 -- value calculated at compile-time.
  std::cout << distance({0, 3}, {4, 0}) << std::endl;
}
```

Compile-time Calculation: Templates vs. `constexpr` Functions

- Template Metaprogramming:
  - More powerful / can do more things than `constexpr` functions.
  - Much harder to read and reason about if there's a bug.
  - Can cause code explosion due to one-time use template instantiations.
- `constexpr` functions
  - Can use `if constexpr` and other familiar programming syntax.
  - Much easier to debug and reason about.
  - Newer – can do more and more things with `constexpr` functions with each Standard
    - Since C++20: `try/catch`, *some* dynamic memory allocation at compile-time.
- General guidelines.
  - Prefer `constexpr` functions if they can be used. Readability is important!
  - Fallback onto TMP if ultimate power is desired.
  - As always, try to use the right tool for the job.

Perfect Forwarding

- Often when programming you'll want to wrap a function call to perform some extra logic.
- This presents a problem:
  - How should we accept the parameters to pass onto the underlying function? Does it make sense to use the same types are the function we are wrapping?
  - How to avoid needless copying? Should we `const T&`? What about `T&&`?
- **Perfect Forwarding** is a way to wrap a function and pass it its arguments *perfectly*.
- This relies on C++'s binding rules and Reference Collapsing.

Binding (Non-templates)

Binding table for a concrete `T` (`int`, `char`, etc.)	Arguments			
Parameters		**lvalue**	**const lvalue**	**rvalue**	**const rvalue**
	T	✓	✓	✓	✓
	T&	✓	✗	✗	✗
	const T&	✓	✓	✓	✓
	T&&	✗	✗	✓	✗

`T` and `const T&` bind to everything!
- Unfortunately, `T` creates a copy
- `const T&` is immutable – what if we need to modify the value?

The rules change when templates are involved.

Binding (Templates)

Binding table for `typename T`	Arguments				
Parameters		**lvalue**	**const lvalue**	**rvalue**	**const rvalue**
	T	✓	✓	✓	✓
	T&	✓	✗	✗	✗
	const T&	✓	✓	✓	✓
	T&&	✓	✓	✓	✓

When `T` is a template parameter, `T&&` binds to everything!
- Binds even if `T` is deduced to be `const` or not.
- Binds even if `T` is a value, an lvalue reference, or an rvalue reference.

Reference Collapsing

- **Question:** what is `decltype(t)` in the calls to `accept_everything`?
  - Call (1): `int&&`
  - Call (2): `const int && &`!?
  - Call (3): `int && &&`!?
- C++ has *reference collapsing* rules:
  - `T& & -> T&`
    - an lvalue to an lvalue is still an lvalue.
  - `T&& & -> T&`
    - an lvalue to an rvalue is still an lvalue.
  - `T& && -> T&`
    - an rvalue to an lvalue is still an lvalue.
  - `T&& && -> T&&`
    - an rvalue to an rvalue is still an rvalue.
- Through reference collapsing, the value category of an argument is preserved across function calls.

```cpp
template <typename T>
void accept_everything(T &&t);

int main() {
  int i = 0;
  const int &j = i;
  int &&k = 6771;

  accept_everything(i); // 1
  accept_everything(j); // 2
  accept_everything(k); // 3
}
```

Forwarding References

- Due to T&& binding to everything and reference collapsing, T&& is known as a *forwarding reference.*
  - Called a "universal reference" in older (circa 2011) texts.
- Used in mainly in function templates that act as wrappers to pass arguments along to wrapped functions.
- To actually forward the argument, use `std::forward`

```cpp
// This might be how std::make_unique is implemented
#include <memory>

struct foo { int a; bool b; const char *c; };

template <typename T, typename ...Args>
std::unique_ptr<T> my_make_unique(
  // note the Args&& - this is a forwarding reference!
  Args&& ...args
) {
  // We are going to construct a T from the arguments
  // that we got passed from the caller.
  // We will pass the arguments to T's constructor
  // exactly the way they were passed to us!
  T *ptr = new T{
    // we call std::forward to actually the arguments along.
    // This will call std::forward on each argument
    // inside the parameter pack
    std::forward<Args>(args)...
  };
  return std::unique_ptr{ptr};
}

auto a = 3;
auto msg = "hi!";

// my_make_unique: T = foo, Args = [int&, bool, const char * &&]
//
// foo's constructor will be called as if we called it directly
// with arguments of these types.
auto up = my_make_unique<foo>(a, a == 3, std::move(msg));
```

Uses of `std::forward`

The only real use for std::`forward` is when you want to wrap a function with a parameterised type. This could be because:

- You want to do something else before or after.
  - `std::make_unique` / `std::make_shared` need to wrap a `std::unique` / `std::shared_ptr` variable.
  - A benchmarking library might wrap a function call with timers.
- You want to do something slightly different.
  - `std::vector::emplace()` constructs its element type in uninitialised memory.
- You want to add an extra parameter.
  - E.g. always call a function with the last parameter as 1.
  - This isn't usually very useful, because it can be achieved with `std::bind` or lambda functions more easily.

A Major Problem with Templates

- Recall that templates have two-phase translation.
  - The template definition is checked for syntax errors only.
  - Upon instantiation, the definition is fully type-checked.
- This leads to type errors being detected extremely late during the instantiation phase.
  - C++ is **infamous** for *terrible* template error messages.
- Prior to C++20 there was no way to specify to the compiler requirements of types expected to be used with a template.

Over 100 lines of cryptic error messages

Constrained Templates (Concepts)

- NEW! in C++20: Concepts.
- Define requirements of types checkable at compile-time to be used with templates.
- Constrain the types usable with a template.
  - Better error messages!
- Manage function and class template overload sets *much* easier.
  - This is called "subsumption"

```
#include <vector>

// A custom concept that check if a type satisfies the
// requirements of a forward container (same as the STL)
template <typename Container>
concept forward_container = requires(
  Container c, const Container cc
) {
  // Does a variable "c" of type Container have a
  // begin() method that returns the correct type?
  // Repeat for the other range methods.
  {c.begin()} -> std::same_as<typename Container::iterator>;
  {c.end()} -> std::same_as<typename Container::iterator>;
  {c.cbegin()} -> std::same_as<typename Container::const_iterator>;
  {c.cend()} -> std::same_as<typename Container::const_iterator>;

  // Does a variable cc with type const Container satisfy
  // .begin() and .end() being const-correct?
  {cc.begin()} -> std::same_as<typename Container::const_iterator>;
  {cc.end()} -> std::same_as<typename Container::const_iterator>;
};

// Result of a concept check is a boolean ::
// Can statically assert the result.
static_assert(forward_container<std::vector<int>>);
```

What is a Concept?

- A concept is a compile-time evaluable boolean expression about properties of a type.
  - It is always a namespace-scope template.
- Allows us to write code that encapsulates "named requirements"
  - What does it mean for a type to be Sortable?
    - It must be a sequential container.
    - It must have random access iterators
    - Each element is comparable with operator<.
  - With concepts, this is writable and checkable by the compiler!

```cpp
#include <type_traits>

namespace nonstd {
  template <typename T>
  // parentheses around the sizeof clause is mandatory
  concept bit32 = std::is_integral_v<T> && (sizeof(T) <= 4);

  template <typename T>
  concept random_access_iterator = /* requirements */;

  template <typename T>
  concept sequential_container = /* requirements */;

  template <typename T>
  concept lt_comparable = /* requirements */;

  template <typename T>
  concept sortable =
    sequential_container<T> &&
    random_access_iterator<typename T::iterator> &&
    lt_comparable<typename T::iterator::value_type>;
}
```

Custom Concepts

- It is possible for a concept to be made up only of type traits.
- Custom requirements written within a `requires` expression.
  - `requires` let's you define 0 or more variables with arbitrary types related to the template parameters of the concept.
  - Inside of the `requires` expression, all requirements must either be:
    - a simple requirement (some syntax that is well-formed); or
    - a type requirement (a nested typename exists); or
    - a nested requirement (e.g. `requires other_concept<T>`); or
    - a **compound** requirement.
  - At the end, the logical conjunction of all statements are taken.
    - I.e., all the statements are && together.
- A compound requirement is:
  - An expression wrapped in `{}` that should be well-formed.
  - Can optionally check the return value type as well.

Custom Concept Examples

```cpp
#include <concepts>

// Custom concept that models an arithmetic type.
template <typename T>
concept arithmetic = requires(T t1, T t2) {
  // a nested requirement
  requires std::regular<T>;

  // two simple requirements
  sizeof(T) <= 8;
  t1 % t2;

  // four compound requirements
  {t1 + t2} -> std::same_as<T>;
  {t1 - t2} -> std::same_as<T>;
  {t1 * t2} -> std::same_as<T>;
  {t1 / t2} -> std::same_as<T>;
};
```

```cpp
#include <concepts>

// Custom concept that models an arithmetic type.
template <typename T>
concept forward_iter = requires(T it) {
  // a nested requirement
  requires std::regular<T>;

  // a simple requirement
  // ("it" is explicitly destructible)
  it.~T();

  // four type requirements
  typename T::iterator_category;
  typename T::value_type;
  typename T::reference;
  typename T::pointer;
  typename T::difference_type;

  // four compound requirements
  {++it} -> std::same_as<T&>;
  {it++} -> std::same_as<T>;
  {*it} -> std::same_as<typename T::reference>;

  // operator-> is callable
  // (not checking return type)
  {it.operator->()};
};
```

What is a Good Concept?

- Concepts can check any piece of syntax.
  - May be tempting to define every operation as its own concept and to link them together in a long logical conjunction.
  - **Don't do this.**
- As a general guideline, if a concept's name captures a microscopic idea and ends in "-able" (e.g., addable), it is probably not a good concept in and of itself.
- A concept should capture a whole, practical idea, e.g.:
  - Whether a type is arithmetic, integral, or floating point.
  - Whether a type models the STL's iterator convention.
  - Whether a type can be sorted.
  - Whether a type is contiguously stored.

Constraining Templates

- With concepts, there are two ways to constrain templates.
- Constrain the template parameter directly in the template parameter list.
  - Short and easy.
  - Multi-parameter concepts: the template parameter is automatically passed as the first argument.
- Assert properties about the template parameter after the parameter list.
  - More flexible than constraining in the parameter list.
- All template varieties (function, class, etc.) can be constrained.

```cpp
#include <concepts>

// Guaranteed to only work with
// integers, longs, etc.
template <std::integral T>
T add(const T &t1, const T &t2) {
  return t1 + t2;
}

// THESE TWO ARE EQUIVALENT

template <typename T>
T add (const T &t1, const T & t2)
requires std::integral<T> {
  return t1 + t2;
}
```

```cpp
#include <concepts>

// Guaranteed to only work with
// types convertible to long
template <std::convertible_to<long> T>
long add(const T &t1, const T &t2) {
  return static_cast<long>(t1) +
static_cast<long>(t2);
}

// THESE TWO ARE EQUIVALENT

template <typename T>
long add(const T &t1, const T &t2)
requires std::convertible_to<T, long> {
  return static_cast<long>(t1) +
static_cast<long>(t2);
}
```

`requires` Clause

- When defining a template, it can be constrained with a pre-existing concept.
- But it can also be constrained with an inline requires clause.
- If the constraint is complex, can use `requires requires`.
  - Should be used sparingly.
  - Better to create a custom concept.

```
#include <concepts>

// Integral is just a name!
// Let's implement what it means
// to be integral with requires requires.
template <typename Integral>
Integral add(Integral i1, Integral i2)
requires requires {
  requires std::regular<Integral>;
  std::is_integral_v<Integral>;
} {
return i1 + i2;
}

// This could have been avoided if
// we had made a custom concept OR
// we had used std::integral instead...
```

Constrained auto

- auto is actually a concept!
  - It is the weakest concept: it has no requirements.
- It is possible to constrain auto using concepts.
  - Can be done on function template parameter types.
  - Can be done on regular variables.

```cpp
#include <concepts>
#include <iostream>

std::floating_point auto add(
  std::floating_point auto f1,
  std::floating_point auto f2
) {
  return f1 + f2;
}

int main() {
// OK: two floats and both are floating point.
std::floating_point auto f = add(42.0f, 6771.0f);

// OK: two doubles and both are floating point.
std::floating_point auto d = add(3.14, 2.71);

// OK: float & double both floating point!
// float is convertible to double
std::floating_point auto mixed = add(3.14f, 2.71);

std::cout << std::boolalpha;
std::cout << std::is_same_v<decltype(f), float> << ' '
<< std::is_same_v<decltype(d), double> << ' '
<< std::is_same_v<decltype(mixed), double> << '\n';

// won't compile -- ints are not floating point!!!
// std::floating_point auto error = add(42, 3.0);
} // Output: true true true
```

Overloading Based on Concepts

- It is possible to manage a template's overload set with concepts.
  - Function templates: candidate function set can be controlled
  - Class templates & variable templates: specialisation set can be controlled.
- To accomplish this:
  - The compiler normalises the requirements of the relevant templates into **conjunctive normal form**.
  - It then undergoes a process of requirement *subsumption*:
    - More constrained templates **subsume** lesser-constrained ones.
  - At the end, if more than one template remains, the usage is ambiguous.

```cpp
#include <concepts>
#include <type_traits>

// Pre-C++20: overloading with enable_if
template <typename T> /* 1 */
std::enable_if_t<std::is_integral_v<T>, T>
secret_algorithm(T i) { return i * 2; }

template <typename T> /* 2 */
std::enable_if_t<std::is_floating_point_v<T>, T>
secret_algorithm(T fp) { return fp * T{3.14}; }

// Post-C++20: overloading with concepts
template <std::integral Int> /* 3 */
Int secret_algorithm (Int i) { return i * 2; }

template <std::floating_point Fp> /* 4 */
Fp secret_algorithm(Fp fp) { return fp * Fp{3.14L}; }

int main() {
   // Calls (3)
  int secret_int = secret_algorithm(6771);

  // Calls (4)
  float secret_float = secret_algorithm(3.14f);
}
```

Standard Concepts

- Standard Concepts can be found in a few places:
  - General purpose concepts (`std::same_as`, etc.): `<concepts>`
  - Iterator concepts (`std::bidirectional_iterator`, etc.): `<iterator>`
  - All of the std::ranges library's concepts: `<ranges>`
- As of C++20, there are only a few standard concepts.
  - Many more expected to be added in C++23 and beyond.
- It is encouraged to leverage the Standard Library's concepts when defining your own.
  - Code reuse > reinventing the wheel.

Feedback (stop recording)