8
Compile-Time Programming

C++ has the ability to evaluate expressions at compile time, meaning that values are already calculated when the program executes. Even though metaprogramming has been possible since C++98, it was initially very complicated due to its complex template-based syntax. With the introduction of `constexpr`, `if constexpr`, and recently, C++ *concepts*, metaprogramming has become much more similar to writing regular code.

This chapter will give you a brief introduction to compile-time expression evaluations in C++ and how they can be used for optimization.

We will cover the following topics:

- Metaprogramming using C++ templates and how to write abbreviated function templates in C++20
- Inspecting and manipulating types at compile time using type traits
- Constant expressions that are evaluated by the compiler
- C++20 concepts and how to use them to add constraints to our template parameters
- Some real-world examples of metaprogramming

We will begin with an introduction to template metaprogramming.

Introduction to template metaprogramming

When writing regular C++ code, it is eventually transformed into machine code. **Metaprogramming**, on the other hand, allows us to write code that transforms itself into regular C++ code. In a more general sense, metaprogramming is a technique where we write code that transforms or generates some other code. By using metaprogramming, we can avoid duplicating code that only differs slightly based on the data types we use, or we can minimize runtime costs by precomputing values that can be known before the final program executes. There is nothing that stops us from generating C++ code by using other languages. We could, for example, do metaprogramming by using preprocessor macros extensively or writing a Python script that generates or modifies C++ files for us:

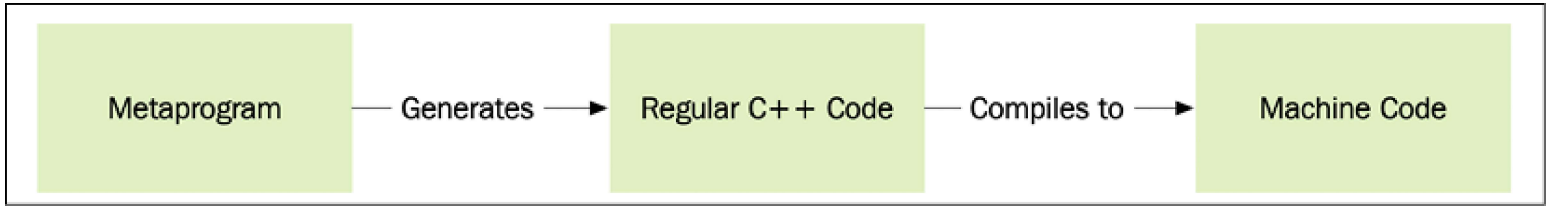

Figure 8.1: A metaprogram generates regular C++ code that will later be compiled into machine code

Even though we could use any language to produce regular code, with C++, we have the privilege of writing metaprograms within the language itself using **templates** and **constant expressions**. The C++ compiler can execute our metaprogram and generate regular C++ code that the compiler will further transform into machine code.

There are many advantages to doing metaprogramming directly within C++ using templates and constant expressions rather than using some other technique:

- We don't have to parse the C++ code (the compiler does that for us).
- There is excellent support for analyzing and manipulating C++ types when using C++ template metaprogramming.
- The code of the metaprogram and the regular non-generic code is mixed in the C++ source. Sometimes, this can make it hard to understand what parts are executed at runtime and compile time, respectively. However, in general, this is a very important aspect of making C++ metaprogramming effective to use.

In its simplest and most common form, template metaprogramming in C++ is used to generate functions, values, and classes that accept different types. A template is said to be **instantiated** when the compiler uses that template to generate a class or a function. Constant expressions are **evaluated** by the compiler to generate constant values:

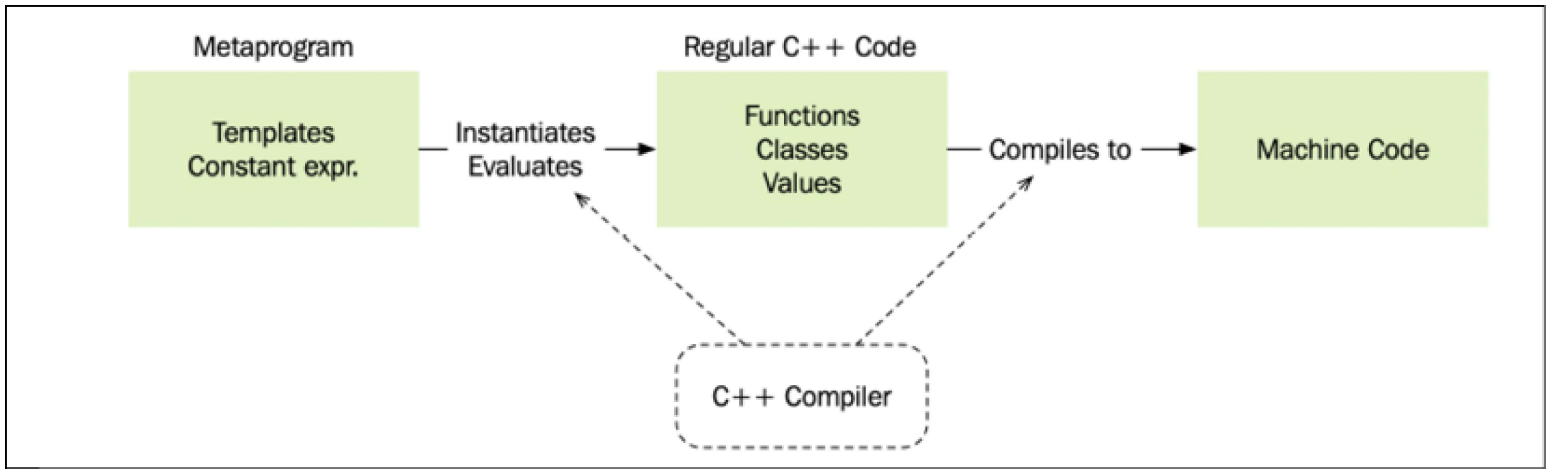

Figure 8.2: Compile-time programming in C++. The metaprogram that will generate regular C++ code is written in C++ itself.

This is a somewhat simplified view; there is nothing that says that the C++ compiler is required to perform the transformations in this way. However, it's useful to think about C++ metaprogramming being carried out in these two distinct phases:

- An initial phase, where templates and constant expressions produce regular C++ code of functions, classes, and constant values. This phase is usually called **constant evaluation**.
- A second phase, where the compiler eventually compiles the regular C++ code into machine code.

Later in this chapter, I will refer to C++ code generated from metaprogramming as *regular C++ code*.

When using metaprogramming, it is important to remember that its main use case is to make great libraries and, thereby, hide complex constructs/optimizations from the user code. Please note that however complex the interior of the code for the metaprogram may be, it's important to hide it behind a good interface so that the user codebase is easy to read and use.

Let's move on and create our first templates for generating function and classes.

Creating templates

Let's take a look at a simple `pow()` function and a `Rectangle` class. By using a **type template parameter**, the `pow()` function and the `Rectangle` class can be used with any integer or floating-point type. Without templates, we would have to create a separate function/class for every base type.

Writing metaprogramming code can be very complex; something that can make it easier is to imagine how the expected regular C++ code is intended to be.

Here is an example of a simple function template:

```
// pow_n accepts any number type
template <typename T>
auto pow_n(const T& v, int n) {
  auto product = T{1};
  for (int i = 0; i < n; ++i) {
    product *= v;
  }
  return product;
}
```

Using this function will generate a function whose return type is dependent on the template parameter type:

```
auto x = pow_n<float>(2.0f, 3); // x is a float
auto y = pow_n<int>(3, 3);      // y is an int
```

The explicit template argument types (`float` and `int` in this case) can (preferably) be omitted, and instead the compiler can figure this out on its own. This mechanism is called **template argument deduction** because the compiler *deduces* the template arguments. The following example will result in the same template instantiation as the one shown previously:

```
auto x = pow_n(2.0f, 3);  // x is a float
auto y = pow_n(3, 3);     // y is an int
```

Correspondingly, a simple class template can be defined as follows:

```
// Rectangle can be of any type
template <typename T>
class Rectangle {
public:
  Rectangle(T x, T y, T w, T h) : x_{x}, y_{y}, w_{w}, h_{h} {}
  auto area() const { return w_ * h_; }
  auto width() const { return w_; }
  auto height() const { return h_; }
private:
  T x_{}, y_{}, w_{}, h_{};
};
```

When a class template is utilized, we can explicitly specify the types that the template should generate the code for, like this:

```
auto r1 = Rectangle<float>{2.0f, 2.0f, 4.0f, 4.0f};
```

But it's also possible to benefit from **class template argument deduction** (**CTAD**), and have the compiler deduce the argument type for us. The following code will instantiate a `Rectangle<int>`:

```
auto r2 = Rectangle{-2, -2, 4, 4};   // Rectangle<int>
```

A function template can then accept a `Rectangle` object where the rectangle dimensions are defined using an arbitrary type `T`, as follows:

```
template <typename T>
auto is_square(const Rectangle<T>& r) {
  return r.width() == r.height();
}
```

Type template parameters are the most common template parameters. Next, you will see how to use numeric parameters instead of type parameters.

Using integers as template parameters

Beyond general types, a template can also be of other types, such as integral types and floating-point types. In the following example, we will use an `int` in the template, which means that the compiler will generate a new function for every unique integer passed as a template argument:

```
template <int N, typename T>
auto const_pow_n(const T& v) {
  auto product = T{1};
  for (int i = 0; i < N; ++i) {
    product *= v;
  }
  return product;
}
```

The following code will oblige the compiler to instantiate two distinct functions: one squares the value and one cubes the value:

```
auto x2 = const_pow_n<2>(4.0f);  // Square
auto x3 = const_pow_n<3>(4.0f);  // Cube
```

Note the difference between the template parameter `N` and the function parameter `v`. For every value of `N`, the compiler generates a new function. However, `v` is passed as a regular parameter and, as such, does not result in a new function.

Providing specializations of a template

By default, the compiler will generate regular C++ code whenever we use a template with new parameters. But it's also possible to provide a custom implementation for certain values of the template parameters. Say, for example, that we want to provide the regular C++ code of our `const_pow_n()` function when it's used with integers and the value of `N` is `2`. We could write a **template specialization** for this case, as follows:

```
template<>
auto const_pow_n<2, int>(const int& v) {
 return v * v;
}
```

For function templates, we need to fix *all* template parameters when writing a specialization. For example, it's not possible to only specify the value of `N` and let the type argument `T` be unspecified.

However, for class templates, it is possible to specify only a subset of the template parameters. This is called **partial template specialization**. The compiler will choose the most specific template first.

The reason we cannot apply partial template specialization to functions is that functions can be overloaded (and classes cannot). If we were allowed to mix overloads and partial specialization, it would be very hard to comprehend.

How the compiler handles a template function

When the compiler handles a template function, it constructs a regular function with the template parameters expanded. The following code will make the compiler generate regular functions since it utilizes templates:

```
auto a = pow_n(42, 3);        // 1. Generate new function
auto b = pow_n(42.f, 2);      // 2. Generate new function
auto c = pow_n(17.f, 5);      // 3.
auto d = const_pow_n<2>(42.f); // 4. Generate new function
auto e = const_pow_n<2>(99.f); // 5.
auto f = const_pow_n<3>(42.f); // 6. Generate new function
```

Thus, when compiled, as distinguished from regular functions, the compiler will generate new functions for every unique set of *template parameters*. This means that it is the equivalent of manually creating four different functions that look something like this:

```
auto pow_n_float(float v, int n) {/*...*/}   // Used by: 1
auto pow_n_int(int v, int n) {/*...*/}      // Used by: 2 and 3
```

```
auto const_pow_n__2_float (float v) {/*...*/} // Used by: 4 and 5
auto const_pow_n__3_float(float v) {/*...*/}  // Used by: 6
```

This is important for understanding how metaprogramming works. The template code generates non-templated C++ code, which is then executed as regular code. If the generated C++ code does not compile, the error will be caught at compile time.

Abbreviated function templates

C++20 introduced a new abbreviated syntax for writing function templates by adopting the same style used by generic lambdas. By using `auto` for function parameter types, we are actually creating a function template rather than a regular function. Recall our initial `pow_n()` template, which was declared like this:

```
template <typename T>
auto pow_n(const T& v, int n) {
 // ...
```

Using the abbreviated function template syntax, we can instead declare it by using `auto`:

```
auto pow_n(const auto& v, int n) { // Declares a function template
 // ...
```

The difference between these two versions is that the abbreviated version doesn't have an explicit placeholder for the type of the variable `v`. And since we were using the placeholder `T` in our implementation, this code will unfortunately fail to compile:

```
auto pow_n(const auto& v, int n) {
  auto product = T{1}; // Error: What is T?
  for (int i = 0; i < n; ++i) {
    product *= v;
  }
  return product;
}
```

To fix this, we can use the `decltype` specifier.

Receiving the type of a variable with decltype

The `decltype` specifier is used to retrieve the type of a variable and is used when an explicit type name is not available.

Sometimes, we need an explicit placeholder for a type but none are available, only the variable name is. This happened to us in our implementation of the `pow_n()` function previously, when using the abbreviated function template syntax.

Let's look at an example of using `decltype` by fixing our implementation of `pow_n()`:

```
auto pow_n(const auto& v, int n) {
  auto product = decltype(v){1};   // Instead of T{1}
  for (int i = 0; i < n; ++i) { product *= v; }
  return product;
}
```

Although this code compiles and works, we are a bit lucky since the type of `v` is actually a `const` reference and not the type we want for the variable `product`. We can get around this by using the left-to-right declaration style. But trying to rewrite the line where the product is defined to something that would appear to be identical reveals a problem:

```
auto pow_n(const auto& v, int n) {
  decltype(v) product{1};
  for (int i = 0; i < n; ++i) { product *= v; } // Error!
  return product;
}
```

Now, we are getting a compilation error since `product` is a `const` reference and may not be assigned to a new value.

What we really want is to get rid of the `const` reference from the type of `v` when defining the variable `product`. We could use a handy template called `std::remove_cvref` for this purpose. Our definition of `product` would then look like this instead:

```
typename std::remove_cvref<decltype(v)>::type product{1};
```

Phew! In this particular case, it would probably have been easier to stick with our initial `template <typename T>` syntax. But now, you have learned how to use `std::remove_cvref` together with `decltype`, which is a common pattern when writing generic C++ code.

Before C++20, it was common to see `decltype` in the body of generic lambdas. However, it is now possible to avoid the rather inconvenient `decltype` by adding explicit template parameters to generic lambdas:

```
auto pow_n = []<class T>(const T& v, int n) {
  auto product = T{1};
  for (int i = 0; i < n; ++i) { product *= v; }
  return product;
};
```

In the definition of the lambda, we are writing `<class T>` in order to get an identifier for the type of the argument that can be used inside the body of the function.

It might take some time to get accustomed to using `decltype` and utilities for manipulating types. Maybe `std::remove_cvref` looks a bit mysterious at first. It's a template from the `<type_traits>` header, which we will look further into in the next section.

Type traits

When doing template metaprogramming, you may often find yourself in situations where you need information about the types you are dealing with at compile time. When writing regular (non-generic) C++ code, we work with concrete types that we have complete knowledge about, but this is not the case when writing a template; the concrete types are not determined until a template is being instantiated by the compiler. Type traits let us extract information about the types our templates are dealing with in order to generate efficient and correct C++ code.

In order to extract information about template types, the standard library provides a type traits library, which is available in the `<type_traits>` header. All type traits are evaluated at compile time.

Type trait categories

There are two categories of type traits:

- Type traits that return information about a type as a boolean or an integer value.
- Type traits that return a new type. These type traits are also called metafunctions.

The first category returns `true` or `false`, depending on the input, and ends with `_v` (short for value).

The `_v` postfix was added in C++17. If your library implementation does not provide `_v` postfixes for type traits, then you can use the older version, `std::is_floating_point<float>::value`. In other words, remove the `_v` extension and add `::value` at the end.

Here are some examples of compile-time type checking using type traits for fundamental types:

```
auto same_type = std::is_same_v<uint8_t, unsigned char>;
auto is_float_or_double = std::is_floating_point_v<decltype(3.f)>;
```

Type traits can also be used on user-defined types:

```
class Planet {};
class Mars : public Planet {};
class Sun {};
```

```
static_assert(std::is_base_of_v<Planet, Mars>);
static_assert(!std::is_base_of_v<Planet, Sun>);
```

The second category of type traits returns a new type and ends with `_t` (short for type). These type trait transformations (or metafunctions) come in handy when dealing with pointers and references:

```
// Examples of type traits which transforms types
using value_type = std::remove_pointer_t<int*>;  // -> int
using ptr_type = std::add_pointer_t<float>;      // -> float*
```

The type trait `std::remove_cvref` that we used earlier is also part of this category. It removes the reference part (if any) and the `const` and `volatile` qualifiers from a type. `std::remove_cvref` was introduced in C++20. Before that, it was conventional to use `std::decay` for this task.

Using type traits

As already mentioned, all type traits are evaluated at compile time. For example, this function, which returns `1` if the value is greater than or equal to zero and `-1` otherwise, can immediately return `1` for unsigned integers, as follows:

```
template<typename T>
auto sign_func(T v) -> int {
 if (std::is_unsigned_v<T>) {
  return 1;
 }
```

```
  return v < 0 ? -1 : 1;
}
```

Since type traits are evaluated at compile time, the compiler will generate the code shown in the following table when invoked with an unsigned and signed integer, respectively:

Used with an unsigned integer...	...generated function:

`auto unsigned_v = uint32_t{42};`
`auto sign = sign_func(unsigned_v);`
	`int sign_func(uint32_t v) {`
	`  if (true) {`
	`  return 1;`
	`  }`
	`  return v < 0 ? -1 : 1;`
	`}`

Used with a signed integer...	...generated function:

`auto signed_v = int32_t{-42};`
`auto sign = sign_func(signed_v);`
	`int sign_func(int32_t v) {`
	`  if (false) {`
	`  return 1;`
	`  }`

```
    return v < 0 ? -1 : 1;
}
```

Table 8.1: Based on the type we pass to sign_func() (in the left column), different functions is generated by the compiler (in the right column).

Next, let's talk about constant expressions.

Programming with constant expressions

An expression prefixed with the `constexpr` keyword tells the compiler that the expression should be evaluated at compile time:

```
constexpr auto v = 43 + 12; // Constant expression
```

The `constexpr` keyword can also be used with functions. In that case, it tells the compiler that a certain function is intended to be evaluated at compile time if all the conditions allowing for compile-time evaluation are fulfilled. Otherwise, it will execute at runtime, like a regular function.

A `constexpr` function has a few restrictions; it is not allowed to do the following:

- Handle local static variables
- Handle `thread_local` variables

- Call any function, which, in itself, is not a `constexpr` function

With the `constexpr` keyword, writing a compile-time evaluated function is as easy as writing a regular function since its parameters are regular parameters instead of template parameters.

Consider the following `constexpr` function:

```
constexpr auto sum(int x, int y, int z) { return x + y + z; }
```

Let's call the function like this:

```
constexpr auto value = sum(3, 4, 5);
```

Since the result of `sum()` is used in a constant expression and all its parameters can be determined at compile time, the compiler will generate the following regular C++ code:

```
const auto value = 12;
```

This is then compiled into machine code, as usual. In other words, the compiler evaluates a `constexpr` function and generates regular C++ code where the result is calculated.

If we called `sum()` instead and stored the result in a variable that is *not* marked with `constexpr`, the compiler *might* (most likely) evaluate `sum()` at compile time:

```
auto value = sum(3, 4, 5); // value is not constexpr
```

In summary, if a `constexpr` function is invoked from a constant expression and all its arguments are constant expressions, it is guaranteed to be evaluated at compile time.

Constexpr functions in a runtime context

In the previous example, the summed values (`3` , `4` , `5`) were known to the compiler at compile time, but how do `constexpr` functions handle variables whose values are not known until runtime? As mentioned in the previous section, `constexpr` is an indicator to the compiler that a function, under certain conditions, can be evaluated at compile time. If variables with values are unknown until the runtime is invoked, they will be evaluated just like regular functions.

In the following example, the values of `x` , `y` , and `z` are provided from the user at runtime, and therefore, it would be impossible for the compiler to calculate the sum at compile time:

```
int x, y, z;
std::cin >> x >> y >> z;     // Get user input
auto value = sum(x, y, z);
```

If we didn't intend to use `sum()` during runtime at all, we could prohibit such usage by making it an immediate function.

Declaring immediate functions using consteval

A `constexpr` function can be called at runtime or compile time. If we want to limit the uses of a function so that it's only invoked at compile time, we can do that by using the keyword `consteval` instead of `constexpr`. Let's assume that we want to prohibit all uses of `sum()` at runtime. With C++20, we can do that with the following code:

```
consteval auto sum(int x, int y, int z) { return x + y + z; }
```

A function that is declared using `consteval` is called an **immediate function** and can only produce constants. If we want to call `sum()`, we need to call it from within a constant expression, or the compilation will fail:

```
constexpr auto s = sum(1, 2, 3); // OK
auto x = 10;
auto s = sum(x, 2, 3);         // Error, expression is not const
```

The compiler will also complain if we try to use `sum()` with parameters that are not known at compile time:

```
int x, y, z;
std::cin >> x >> y >> z;
constexpr auto s = sum(x, y, z); // Error
```

Let's discuss the `if constexpr` statement next.

The if constexpr statement

The `if constexpr` statement allows template functions to evaluate different scopes in the same function at compile time (also called compile-time polymorphism). Take a look at the following example, where a function template called `speak()` tries to differentiate member functions, depending on the type:

```
struct Bear { auto roar() const { std::cout << "roar\n"; } };
struct Duck { auto quack() const { std::cout << "quack\n"; } };
template <typename Animal>
auto speak(const Animal& a) {
  if (std::is_same_v<Animal, Bear>) { a.roar(); }
  else if (std::is_same_v<Animal, Duck>) { a.quack(); }
}
```

Let's say we compile the following lines:

```
auto bear = Bear{};
speak(bear);
```

The compiler will then generate a `speak()` function, similar to this:

```
auto speak(const Bear& a) {
  if (true) { a.roar(); }
  else if (false) { a.quack(); } // This line will not compile
}
```

As you can see, the compiler will keep the call to the member function, `quack()`, which will then fail to compile since `Bear` does not contain a `quack()` member function. This happens even though the `quack()` member function will never be executed due to the `else if (false)` statement.

In order to make the `speak()` function compile, regardless of the type, we need to inform the compiler that we'd like to completely ignore the scope if the `if` statement is `false`. Conveniently, this is exactly what `if constexpr` does.

Here is how we can write the `speak()` function with the ability to handle both `Bear` and `Duck`, even though they do not share a common interface:

```
template <typename Animal>
auto speak(const Animal& a) {
  if constexpr (std::is_same_v<Animal, Bear>) { a.roar(); }
  else if constexpr (std::is_same_v<Animal, Duck>) { a.quack(); }
}
```

When `speak()` is invoked with `Animal == Bear`, as follows:

```
auto bear = Bear{};
speak(bear);
```

the compiler generates the following function:

```
auto speak(const Bear& animal) { animal.roar(); }
```

When `speak()` is invoked with `Animal == Duck`, as follows:

```
auto duck = Duck{};
speak(duck);
```

the compiler generates the following function:

```
auto speak(const Duck& animal) { animal.quack(); }
```

If `speak()` is invoked with any other primitive type, such as `Animal == int`, as follows:

```
speak(42);
```

the compiler generates an empty function:

```
auto speak(const int& animal) {}
```

Unlike a regular `if` statement, the compiler is now able to generate multiple different functions: one using `Bear`, another one using `Duck`, and a last one if the type is neither `Bear` nor `Duck`. If we want to make this third case a compilation error, we can do that by adding an `else` case with a `static_assert`:

```
template <typename Animal>
auto speak(const Animal& a) {
```

```
  if constexpr (std::is_same_v<Animal, Bear>) { a.roar(); }
  else if constexpr (std::is_same_v<Animal, Duck>) { a.quack(); }
  else { static_assert(false); } // Trig compilation error
}
```

We will talk more about the usefulness of `static_assert` later.

As mentioned earlier, the way `constexpr` is being used here can be referred to as compile-time polymorphism. So, how does it relate to runtime polymorphism?

Comparison with runtime polymorphism

As a side note, if we were to implement the previous example with traditional runtime polymorphism, using inheritance and virtual functions to achieve the same functionality, the implementation would look as follows:

```
struct AnimalBase {
  virtual ~AnimalBase() {}
  virtual auto speak() const -> void {}
};
struct Bear : public AnimalBase {
  auto roar() const { std::cout << "roar\n"; }
  auto speak() const -> void override { roar(); }
};
struct Duck : public AnimalBase {
  auto quack() const { std::cout << "quack\n"; }
  auto speak() const -> void override { quack(); }
};
```

```
auto speak(const AnimalBase& a) {
  a.speak();
}
```

The objects have to be accessed using pointers or references, and the type is inferred at *runtime*, which results in a performance loss compared with the compile-time version, where everything is available when the application executes. The following image shows the difference between the two types of polymorphism in C++:

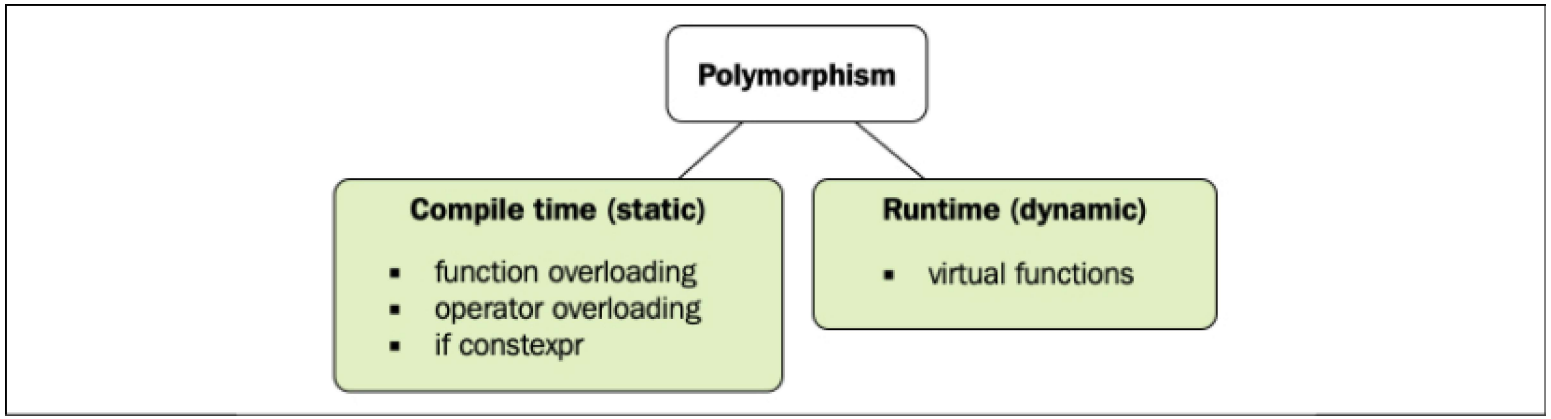

Figure 8.3: Runtime polymorphism is supported by virtual functions whereas compile time polymorphism is supported by function/operator overloading and if constexpr.

Now, we will continue to see how we can use `if constexpr` for something more useful.

Example of the generic modulus function using if constexpr

This example will show you how to use `if constexpr` to distinguish between operators and global functions. In C++, the `%` operator is used to get the modulus of integers, while `std::fmod()` is used for floating-point types. Say we'd like to generalize our codebase and create a generic modulus function called `generic_mod()`.

If we were to implement `generic_mod()` with a regular `if` statement, as follows:

```
template <typename T>
auto generic_mod(const T& v, const T& n) -> T {
  assert(n != 0);
  if (std::is_floating_point_v<T>) { return std::fmod(v, n); }
  else { return v % n; }
}
```

it would fail if invoked with `T == float` as the compiler would generate the following function, which would fail to compile:

```
auto generic_mod(const float& v, const float& n) -> float {
  assert(n != 0);
  if (true) { return std::fmod(v, n); }
  else { return v % n; } // Will not compile
}
```

Even though the application cannot reach it, the compiler will generate the line `return v % n;`, which isn't compliant with `float`. The compiler doesn't care that the application cannot reach it—since it cannot generate an assembly for it, it will fail to compile.

As in the previous example, we will change the `if` statement to an `if constexpr` statement:

```
template <typename T>
auto generic_mod(const T& v, const T& n) -> T {
  assert(n != 0);
  if constexpr (std::is_floating_point_v<T>) {
    return std::fmod(v, n);
  } else {              // If T is a floating point,
    return v % n;       // this code is eradicated
  }
}
```

Now, when the function is invoked with a floating-point type, it will generate the following function, where the `v % n` operation is eradicated:

```
auto generic_mod(const float& v, const float& n) -> float {
  assert(n != 0);
  return std::fmod(v, n);
}
```

The runtime `assert()` tells us that we cannot call this function if the second argument is 0.

Checking programming errors at compile time

Assert statements are a simple but very powerful tool for validating invariants and contracts between callers and callees in a codebase, (see *Chapter 2*, *Essential C++ Techniques*.) It's possible to check pro-

gramming errors while executing the program using `assert()`. But we should always strive to detect errors as early as possible, and if we have a constant expression, we can catch programming errors when compiling the program using `static_assert()`.

Using assert to trigger errors at runtime

Review the templated version of `pow_n()`. Let's say we want to prevent it from being called with negative exponents (the `n` value). To prevent this in the runtime version, where `n` is a regular argument, we can add a runtime assertion:

```
template <typename T>
auto pow_n(const T& v, int n) {
  assert(n >= 0); // Only works for positive numbers
  auto product = T{1};
  for (int i = 0; i < n; ++i) {
    product *= v;
  }
  return product;
}
```

If the function is called with a negative value for `n`, the program will break and inform us where we should start looking for the bug. This is good, but it would be even better if we could track this error at compile time rather than runtime.

Using static_assert to trigger errors at compile time

If we do the same to the template version, we can utilize `static_assert()`. The `static_assert()` declaration, unlike a regular assert, will refuse to compile if the condition isn't fulfilled. So, it's better to break the build than have a program break at runtime. In the following example, if the template parameter `N` is a negative number, `static_assert()` will prevent the function from compiling:

```
template <int N, typename T>
auto const_pow_n(const T& v) {
  static_assert(N >= 0, "N must be positive");
  auto product = T{1};
  for (int i = 0; i < N; ++i) {
    product *= v;
  }
  return product;
}
auto x = const_pow_n<5>(2);  // Compiles, N is positive
auto y = const_pow_n<-1>(2); // Does not compile, N is negative
```

In other words, with regular variables, the compiler is only aware of the type and has no idea what it contains. With compile-time values, the compiler knows both the type and the value. This allows the compiler to calculate other compile-time values.

Instead of using an `int` and assert that it's non-negative, we could (should) have used an `unsigned int` instead. We are only using a signed `int` in this example to demonstrate the use of `assert()` and `static_assert()`.

Using compile-time asserts is one way to check constraints at compile time. It is a simple but very useful tool. The support for compile-time programming has seen some very exciting progress over the last few years in C++. Now, we will move on to one of the biggest features from C++20 that takes constraints checking to a new level.

Constraints and concepts

So far, we have covered quite a few important techniques for writing C++ metaprograms. You have seen how templates can generate concrete classes and functions for us with excellent support from the type traits library. Furthermore, you have seen how the use of `constexpr`, `consteval`, and `if constexpr` can help us move computations from runtime to compile time. In that way, we can detect programming errors at compile time and write programs with lower runtime costs. This is great, but there is still plenty of room for improvement when it comes to writing and consuming generic code in C++. Some of the issues that we haven't addressed yet include:

1. Interfaces are too generic. When using a template with some arbitrary type, it's hard to know what the requirements of that type are. This makes the templates hard to use if we only inspect the template interface. Instead, we have to rely on documentation or dig deep into the implementation of a template.
2. Type errors are caught late by the compiler. The compiler will eventually check the types when compiling the regular C++ code, but the error messages are usually hard to interpret. Instead, we would like type errors to be caught in the instantiation phase.
3. Unconstrained template parameters make metaprogramming hard. The code we have written so far in this chapter has used unconstrained template parameters, with the exception of a few static asserts. This is manageable for small examples, but it would be much easier to write and reason about our

metaprograms if we could have access to more meaningful types, in the same way the type system helps us write correct non-generic C++ code.
4. Conditional code generation (compile-time polymorphism) can be performed using `if constexpr`, but it quickly becomes hard to read and write at a larger scale.

As you will see in this section, C++ concepts address these issues in an elegant and effective way by introducing two new keywords: `concept` and `requires`. Before exploring constraints and concepts, we will spend some time considering the shortcomings of template metaprogramming without concepts. Then, we will use constraints and concepts to strengthen our code.

An unconstrained version of a Point2D template

Suppose we are writing a program that deals with a two-dimensional coordinate system. We have a class template that represents a point with `x` and `y` coordinates, as follows:

```
template <typename T>
class Point2D {
public:
  Point2D(T x, T y) : x_{x}, y_{y} {}
  auto x() { return x_; }
  auto y() { return y_; }
  // ...
private:
  T x_{};
  T y_{};
};
```

Let's assume that we need to find the Euclidean distance between two points, **p1** and **p2**, as illustrated here:

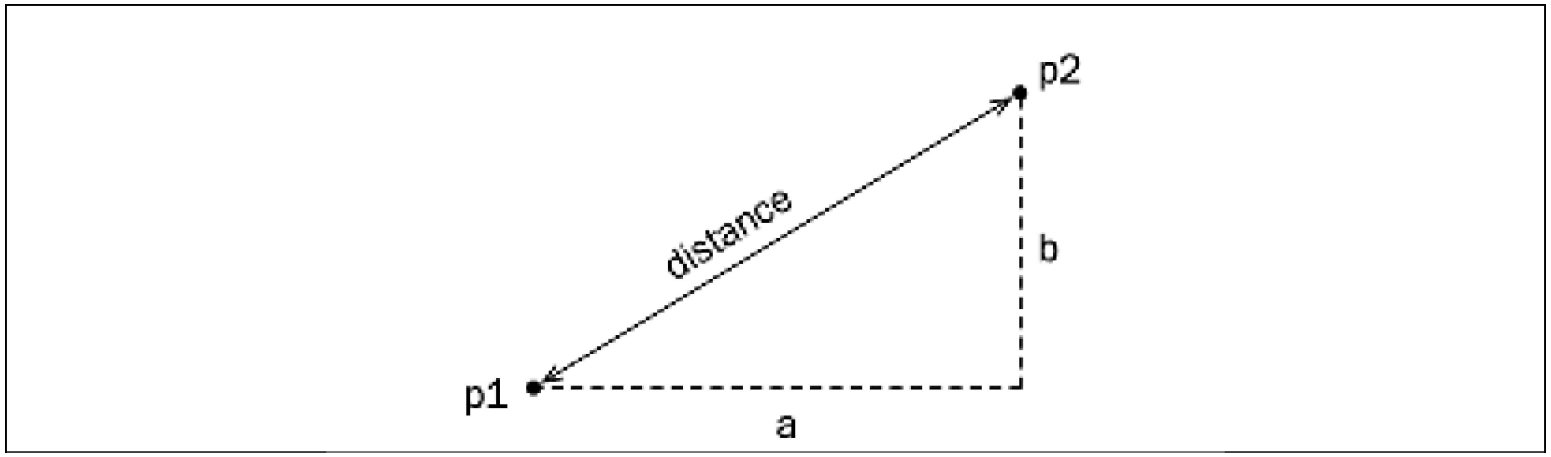

Figure 8.4: Finding the Euclidean between p1 and p2

To compute the distance, we implement a free function that takes two points and uses the Pythagorean theorem (the actual math is of less importance here):

```
auto dist(auto p1, auto p2) {
  auto a = p1.x() - p2.x();
  auto b = p1.y() - p2.y();
  return std::sqrt(a*a + b*b);
}
```

A small test program verifies that we can instantiate the `Point2D` template with integers and compute the distance between the two points:

```
int main() {
  auto p1 = Point2D{2, 2};
  auto p2 = Point2D{6, 5};
  auto d = dist(p1, p2);
  std::cout << d;
}
```

This code compiles and runs fine and outputs `5` to the console.

Generic interfaces and bad error messages

Before moving on, let's take a short detour and reflect for a while on the function template `dist()`. Let's imagine that we didn't have easy access to the implementation of `dist()` and only could read the interface:

```
auto dist(auto p1, auto p2) // Interface part
```

What can we say about the return type and the types of `p1` and `p2`? Practically nothing—because `p1` and `p2` are completely *unconstrained*, the interface of `dist()` does not reveal anything for us. This doesn't mean that we can pass anything to `dist()`, though, because in the end, the generated regular C++ code has to compile.

For example, if we try to instantiate our `dist()` template with two integers instead of `Point2D` objects like this:

```
auto d = dist(3, 4);
```

the compiler will gladly generate a regular C++ function, similar to this:

```
auto dist(int p1, int p2) {
  auto a = p1.x() - p2.x();  // Will generate an error:
  auto b = p1.y() - p2.y();  // int does not have x() and y()
  return std::sqrt(a*a + b*b);
}
```

The error will be caught later on when the regular C++ code is checked by the compiler. Clang generates the following error message when trying to instantiate `dist()` with two integers:

```
error: member reference base type 'int' is not a structure or union
auto a = p1.x() - p2.y();
```

This error message refers to the *implementation* of `dist()`, something that the caller of the function `dist()` shouldn't need to know about. This is a trivial example, but trying to interpret error messages caused by providing wrong types to templates from sophisticated template libraries can be a real challenge.

Even worse, if we are really unlucky, we get through the entire compilation by providing types that don't make sense at all. In this case, we are instantiating a `Point2D` with `const char*`:

```
int main() {
  auto from = Point2D{"2.0", "2.0"}; // Ouch!
  auto to = Point2D{"6.0", "5.0"};   // Point2D<const char*>
  auto d = dist(from, to);
  std::cout << d;
}
```

It compiles and runs, but the output is probably not what we would expect. We want to catch these sorts of errors earlier on in the process, something we can achieve by using constraints and concepts as shown in the image below:

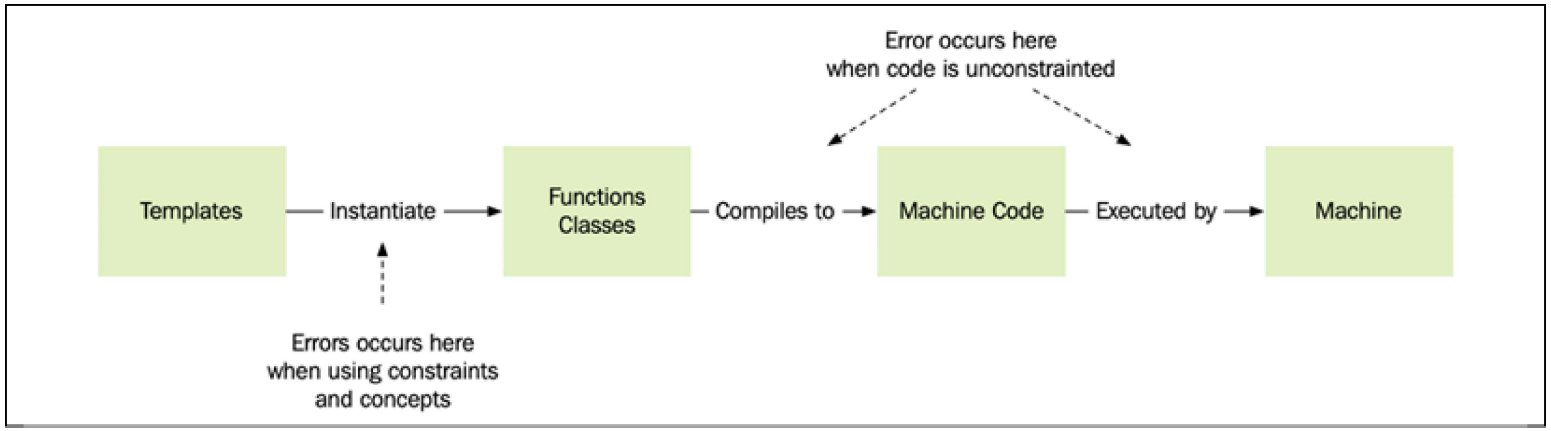

Figure 8.5: Type errors can be caught at instantiation phase using constraints and concepts

Later on, you will see how this code can be made more expressive so that it is easier to use correctly and harder to misuse. We will do this by adding concepts and constraints to our code. But first, I will provide a quick overview of how to define and use concepts.

A syntactic overview of constraints and concepts

This section is a short introduction to constraints and concepts. We will not cover them completely in this book but I will provide you with enough material to be productive.

Defining new concepts

Defining new concepts is straightforward with some help from the type traits that you are already familiar with. The following example defines the concept `FloatingPoint` using the keyword `concept`:

```
template <typename T>
concept FloatingPoint = std::is_floating_point_v<T>;
```

The right-hand side of the assignment expression is where we can specify the constraints of the type `T`. It's also possible to combine multiple constraints using `||` (logical OR) and `&&` (logical AND). The following example uses `||` to combine floats and integrals into a `Number` concept:

```
template <typename T>
concept Number = FloatingPoint<T> || std::is_integral_v<T>;
```

You will note that, it's possible to build concepts using already defined concepts on the right-hand side as well. The standard library contains a `<concepts>` header, which defines many useful concepts, such as `std::floating_point` (which we should use rather than defining our own).

Furthermore, we can use the `requires` keyword to add a set of statements that should be added to our concept definition. For example, this is the definition of the concept `std::range` from the Ranges library:

```
template<typename T>
concept range = requires(T& t) {
  ranges::begin(t);
  ranges::end(t);
};
```

In short, this concept states that a range is something that we can pass to `std::ranges::begin()` and `std::ranges::end()`.

It's possible to write more sophisticated `requires` clauses than this, and you will see more on that later on.

Constraining types with concepts

We can add constraints to template parameter types by using the `requires` keyword. The following template can only be instantiated with the parameters of integral types by using the concept `std::integral`:

```
template <typename T>
requires std::integral<T>
auto mod(T v, T n) {
  return v % n;
}
```

We can use the same technique when defining class templates:

```
template <typename T>
requires std::integral<T>
struct Foo {
  T value;
};
```

An alternative syntax allows us to write this in a more compact way by replacing the `typename` directly with the concept:

```
template <std::integral T>
auto mod(T v, T n) {
  return v % n;
}
```

This form can also be used with class templates:

```
template <std::integral T>
struct Foo {
  T value;
};
```

If we want to use the abbreviated function template form when defining a function template, we can add the concept in front of the `auto` keyword:

```
auto mod(std::integral auto v, std::integral auto n) {
  return v % n;
}
```

The return type can also be constrained by using concepts:

```
std::integral auto mod(std::integral auto v, std::integral auto n) {
  return v % n;
}
```

As you can see, there are many ways to specify the same thing. The abbreviated form combined with concepts has really made it easy to both read and write constrained function templates. Another powerful feature of C++ concepts is the ability to overload functions in a clear and expressive way.

Function overloading

Recall the `generic_mod()` function we implemented earlier using `if constexpr`. It looked something like this:

```
template <typename T>
auto generic_mod(T v, T n) -> T {
  if constexpr (std::is_floating_point_v<T>) {
    return std::fmod(v, n);
  } else {
    return v % n;
```

```
 }
}
```

By using concepts, we can overload a function template similar to how we would have done if we'd written a regular C++ function:

```
template <std::integral T>
auto generic_mod(T v, T n) -> T {          // Integral version
 return v % n;
}
template <std::floating_point T>
auto generic_mod(T v, T n) -> T {          // Floating point version
 return std::fmod(v, n);
}
```

With your new knowledge of constraints and concepts, it's time to go back to our example with the `Point2D` template and see how it can be improved.

A constrained version of the Point2D template

Now that you know how to define and use concepts, let's put them to use by writing a better version of our templates, `Point2D` and `dist()`. Remember that what we're aiming for is a more expressive interface and to have errors caused by irrelevant parameter types appear at template instantiation.

We will begin by creating a concept for arithmetic types:

```
template <typename T>
concept Arithmetic = std::is_arithmetic_v<T>;
```

Next, we will create a concept called `Point` that defines that a point should have the member functions `x()` and `y()` return the same type, and that this type should support arithmetic operations:

```
template <typename T>
concept Point = requires(T p) {
  requires std::is_same_v<decltype(p.x()), decltype(p.y())>;
  requires Arithmetic<decltype(p.x())>;
};
```

This concept can now make the interface of `dist()` much better with explicit constraints:

```
auto dist(Point auto p1, Point auto p2) {
  // Same as before ...
```

This is starting to look really promising, so let's just add a constraint to our return type as well. Although `Point2D` might be instantiated with an integral type, we know that the distance can be a floating-point number. The concept `std::floating_point` from the standard library is well suited for this. Here is the final version of `dist()`:

```
std::floating_point auto dist(Point auto p1, Point auto p2) {
  auto a = p1.x() - p2.x();
  auto b = p1.y() - p2.y();
```

```
  return std::sqrt(a*a + b*b);
}
```

Our interface is now more descriptive, and when we try to instantiate it with wrong parameter types, we will get errors during the instantiation phase rather than the final compilation phase.

We should now do the same to our `Point2D` template to avoid someone accidentally instantiating it with types that it wasn't intended to handle. For example, we would like to prevent someone from instantiating a `Point2D` class with `const char*`, like this:

```
auto p1 = Point2D{"2.0", "2.0"}; // How can we prevent this?
```

We have already created the `Arithmetic` concept, which we can use here to put constraints in the template parameter of `Point2D`. Here is how we do this:

```
template <Arithmetic T> // T is now constrained!
class Point2D {
public:
  Point2D(T x, T y) : x_{x}, y_{y} {}
  auto x() { return x_; }
  auto y() { return y_; }
  // ...
private:
  T x_{};
  T y_{};
};
```

The only thing we needed to change was to specify that the type `T` should support the operations specified by the concept `Arithmetic`. Trying to instantiate a template using `const char*` will now generate a direct error message while the compiler tries to instantiate a `Point2D<const char*>` class.

Adding constraints to your code

The usefulness of concepts reaches far beyond template metaprogramming. It's a fundamental feature of C++20 that changes how we write and reason about code using concepts other than concrete types or completely unconstrained variables declared with `auto`.

A concept is very similar to a type (such as `int`, `float`, or `Plot2D<int>`). Both types and concepts specify a set of supported operations on an object. By inspecting a type or a concept, we can determine how certain objects can be constructed, moved, compared, and accessed by member functions and so on. A big difference, though, is that a concept does not say anything about how an object is stored in memory, whereas a type provides this information in addition to its set of supported operations. For example, we can use the `sizeof` operator on a type but not on a concept.

With concepts and `auto`, we can declare variables without the need for spelling out the exact type, but still express the intent with our code very clearly. Have a look at the following code snippet:

```
const auto& v = get_by_id(42); // What can I do with v?
```

Most of the time, when we stumble upon code like this, we are interested in knowing what operations we can perform on `v` rather than knowing the exact type. Adding a concept in front of `auto` makes the difference:

```
const Person auto& v = get_by_id(42);
v.get_name();
```

It's possible to use concepts in almost all contexts where we can use the keyword `auto`: local variables, return values, function arguments, and so forth. Using concepts in our code makes it easier to read. At the time of writing this book (mid 2020), there is currently no additional support for concepts in the established C++ IDEs. However, it is just a matter of time before code completion, as well as other useful editor features based on concepts, will be available and make C++ coding both more fun and safer.

Concepts in the standard library

C++20 also included a new `<concepts>` header with predefined concepts. You have already seen some of them in action. Many concepts are based on the traits from the type traits library. However, there are a few fundamental concepts that have not been expressed with traits previously. Among the most important ones are the comparison concepts such as `std::equality_comparable` and `std::totally_ordered`, as well as the object concepts such as `std::movable`, `std::copyable`, `std::regular`, and `std::semiregular`. We will not spend any more time on the concepts from the standard library but remember to keep them in mind before starting to define your own. Defining concepts on the correct level of generality is not trivial and it's usually wise to define new concepts based on the already existing ones.

Let's end this chapter by having a look at some real-world examples of metaprogramming in C++.

Real-world examples of metaprogramming

Advanced metaprogramming can appear to be very academic, so in order to demonstrate its usefulness, let's look at some examples that not only demonstrate the syntax of metaprogramming, but how it can be used in practice.

Example 1: creating a generic safe cast function

When casting between data types in C++, there is a multitude of different ways things can go wrong:

- You might lose a value if casting to an integer type of a lower bit length.
- You might lose a value if casting a negative value to an unsigned integer.
- If casting from a pointer to any other integer than `uintptr_t`, the correct address might become incorrect. This is because C++ only guarantees that `uintptr_t` is the only integer type that can withhold an address.
- If casting from `double` to `float`, the result might be `int` if the `double` value is too large for `float` to withhold.
- If casting between pointers with a `static_cast()`, we might get undefined behavior if the types aren't sharing a common base class.

In order to make our code more robust, we can create a generic checked cast function that verifies our casts in debug mode and performs our casts as fast as possible if in release mode.

Depending on the types that are being cast, different checks are performed. If we try to cast between types that are not verified, it won't compile.

These are the cases `safe_cast()` is intended to handle:

- **Same type**: Obviously, if we're casting the same type, we just return the input value.

- **Pointer to pointer**: If casting between pointers, `safe_cast()` performs a dynamic cast in debug mode to verify it is castable.
- **Double to floating point**: `safe_cast()` accepts precision loss when casting from `double` to `float` with one exception – if casting from a `double` to a `float`, there is a chance the double is too large for the float to handle the result.
- **Arithmetic to arithmetic**: If casting between arithmetic types, the value is cast back to its original type to verify no precision has been lost.
- **Pointer to non-pointer**: If casting from a pointer to a non-pointer type, `safe_cast()` verifies that the destination type is an `uintptr_t` or `intptr_t`, the only integer types that are guaranteed to hold an address.

In any other case, the `safe_cast()` function fails to compile.

Let's see how we can implement this. We start by fetching information about our cast operation in `const-expr` booleans. The reason they are `constexpr` booleans and not `const` booleans is that we will utilize them later in `if constexpr` expressions, which require `constexpr` conditions:

```
template <typename T> constexpr auto make_false() { return false; }
template <typename Dst, typename Src>
auto safe_cast(const Src& v) -> Dst{
 using namespace std;
 constexpr auto is_same_type = is_same_v<Src, Dst>;
 constexpr auto is_pointer_to_pointer =
  is_pointer_v<Src> && is_pointer_v<Dst>;
 constexpr auto is_float_to_float =
  is_floating_point_v<Src> && is_floating_point_v<Dst>;
 constexpr auto is_number_to_number =
  is_arithmetic_v<Src> && is_arithmetic_v<Dst>;
```

```
constexpr auto is_intptr_to_ptr =
  (is_same_v<uintptr_t,Src> || is_same_v<intptr_t,Src>)
  && is_pointer_v<Dst>;
constexpr auto is_ptr_to_intptr =
  is_pointer_v<Src> &&
  (is_same_v<uintptr_t,Dst> || is_same_v<intptr_t,Dst>);
```

So, now that we have all the necessary information about the cast as `constexpr` booleans, we assert at compile time that we can perform the cast. As mentioned previously, a `static_assert()` will fail to compile if the condition is not satisfied (unlike a regular assert, which verifies conditions at runtime).

Note the usage of `static_assert()` and `make_false<T>` at the end of the `if` / `else` chain. We cannot just type `static_assert(false)` as that would prevent `safe_cast()` from compiling at all; instead, we utilize the template function `make_false<T>()` to delay the generation until required.

When the actual `static_cast()` is performed, we cast back to the original type and verify that the result is equal to the uncasted argument using a regular runtime `assert()` This way, we can make sure the `static_cast()` has not lost any data:

```
if constexpr(is_same_type) {
  return v;
}
else if constexpr(is_intptr_to_ptr || is_ptr_to_intptr){
  return reinterpret_cast<Dst>(v);
}
else if constexpr(is_pointer_to_pointer) {
  assert(dynamic_cast<Dst>(v) != nullptr);
  return static_cast<Dst>(v);
```

```
  }
  else if constexpr (is_float_to_float) {
    auto casted = static_cast<Dst>(v);
    auto casted_back = static_cast<Src>(v);
    assert(!isnan(casted_back) && !isinf(casted_back));
    return casted;
  }
  else if constexpr (is_number_to_number) {
    auto casted = static_cast<Dst>(v);
    auto casted_back = static_cast<Src>(casted);
    assert(casted == casted_back);
    return casted;
  }
  else {
    static_assert(make_false<Src>(),"CastError");
    return Dst{}; // This can never happen,
    // the static_assert should have failed
  }
}
```

Note how we use the `if constexpr` in order for the function to conditionally compile. If we use a regular `if` statement, the function will fail to compile:

```
auto x = safe_cast<int>(42.0f);
```

This is because the compiler will try to compile the following line and `dynamic_cast` only accepts pointers:

```
// type To is an integer
assert(dynamic_cast<int>(v) != nullptr); // Does not compile
```

However, thanks to the `if constexpr` and `safe_cast<int>(42.0f)` constructs, the following function compiles properly:

```
auto safe_cast(const float& v) -> int {
  constexpr auto is_same_type = false;
  constexpr auto is_pointer_to_pointer = false;
  constexpr auto is_float_to_float = false;
  constexpr auto is_number_to_number = true;
  constexpr auto is_intptr_to_ptr = false;
  constexpr auto is_ptr_to_intptr = false
  if constexpr(is_same_type) { /* Eradicated */ }
  else if constexpr(is_intptr_to_ptr||is_ptr_to_intptr){/* Eradicated */}
  else if constexpr(is_pointer_to_pointer) {/* Eradicated */}
  else if constexpr(is_float_to_float) {/* Eradicated */}
  else if constexpr(is_number_to_number) {
    auto casted = static_cast<int>(v);
    auto casted_back = static_cast<float>(casted);
    assert(casted == casted_back);
    return casted;
  }
  else { /* Eradicated */ }
}
```

As you can see, except for the `is_number_to_number` clause, everything in-between the `if constexpr` statements has been completely eradicated, allowing the function to compile.

Example 2: hash strings at compile time

Let's say we have a resource system consisting of an unordered map of strings that identifies bitmaps. If a bitmap is already loaded, the system returns the loaded bitmap; otherwise, it loads the bitmap and returns it:

```
// External function which loads a bitmap from the filesystem
auto load_bitmap_from_filesystem(const char* path) -> Bitmap {/* ... */}
// Bitmap cache
auto get_bitmap_resource(const std::string& path) -> const Bitmap& {
  // Static storage of all loaded bitmaps
  static auto loaded = std::unordered_map<std::string, Bitmap>{};
  // If the bitmap is already in loaded_bitmaps, return it
  if (loaded.count(path) > 0) {
    return loaded.at(path);
  }
  // The bitmap isn't already loaded, load and return it
  auto bitmap = load_bitmap_from_filesystem(path.c_str());
  loaded.emplace(path, std::move(bitmap));
  return loaded.at(path);
}
```

The bitmap cache is then utilized wherever a bitmap resource is needed:

- If it's not loaded yet, the `get_bitmap_resource()` function will load and return it
- If it's already been loaded somewhere else, the `get_bitmap_resource()` will simply return the loaded function

So, independent of which of these draw functions is executed first, the second one will not have to load the bitmap from disk:

```
auto draw_something() {
  const auto& bm = get_bitmap_resource("my_bitmap.png");
  draw_bitmap(bm);
}
auto draw_something_again() {
  const auto& bm = get_bitmap_resource("my_bitmap.png");
  draw_bitmap(bm);
}
```

Since we are using an unordered map, we need to compute a hash value whenever we check for a bitmap resource. You will now see how we can optimize the runtime code by moving computations to compile time.

The advantages of the compile-time hash sum calculation

The problem that we will try to solve is that every time the line `get_bitmap_resource("my_bitmap.png")` is executed, the application will compute the hash sum of the string `"my_bitmap.png"` at runtime. What we would like to do is perform this calculation at compile time so that when the application executes, the hash sum has already been calculated. In other words, just as you have learned to use metaprogramming

to generate functions and classes at compile time, we will now have it generate the hash sum at compile time.

You might have already come to the conclusion that this is a so-called *micro-optimization*: calculating the hash sum of a small string won't affect the application's performance at all as it is such a tiny operation. That is probably completely true; this is just an example of how to move a calculation from runtime to compile time, and there might be other instances where this can make a significant performance impact.

As a side note, when writing software for weak hardware, string hashing is a pure luxury, but hashing strings at compile time gives us this luxury on any platform since everything is computed at compile time.

Implementing and verifying a compile-time hash function

In order to enable the compiler to calculate the hash sum at compile time, we rewrite `hash_function()` so that it takes a raw null-terminated `char` string as a parameter of an advanced class like `std::string`, which cannot be evaluated at compile time. Now, we can mark `hash_function()` as `constexpr`:

```
constexpr auto hash_function(const char* str) -> size_t {
  auto sum = size_t{0};
  for (auto ptr = str; *ptr != '\0'; ++ptr)
    sum += *ptr;
  return sum;
}
```

Now, let's invoke this with a raw literal string known at compile time:

```
auto hash = hash_function("abc");
```

The compiler will generate the following piece of code, which is the sum of the ASCII values corresponding to `a`, `b`, and `c` (`97`, `98`, and `99`):

```
auto hash = size_t{294};
```

Just accumulating the individual values is a very bad hash function; do not do this in a real-world application. It's only here because it's easy to grasp. A better hash function would be to combine all the individual characters with `boost::hash_combine()`, as explained in *Chapter 4*, *Data Structures*.

`hash_function()` will only evaluate at compile time if the compiler knows the string at compile time; if not, the compiler will execute `constexpr` at runtime, just like any other expression.

Now that we have the hash function in place, it's time to create a string class that uses it.

Constructing a PrehashedString class

We are now ready to implement a class for pre-hashed strings that will use the hash function we created. This class consists of the following:

- A constructor that takes a raw string as a parameter and calculates the hash at construction.
- Comparison operators.

- A `get_hash()` member function, which returns the hash.
- An overload of `std::hash()`, which simply returns the hash value. This overload is used by `std::unordered_map`, `std::unordered_set`, or any other class from the standard library that uses hash values. To put it simply, this makes the container aware that a hash function exists for the `PrehashedString`.

Here is a basic implementation of a `PrehashedString` class:

```
class PrehashedString {
public:
 template <size_t N>
 constexpr PrehashedString(const char(&str)[N])
    : hash_{hash_function(&str[0])}, size_{N - 1},
    // The subtraction is to avoid null at end
     strptr_{&str[0]} {}
 auto operator==(const PrehashedString& s) const {
  return
    size_ == s.size_ &&
    std::equal(c_str(), c_str() + size_, s.c_str());
 }
 auto operator!=(const PrehashedString& s) const {
  return !(*this == s); }
 constexpr auto size()const{ return size_; }
 constexpr auto get_hash()const{ return hash_; }
 constexpr auto c_str()const->const char*{ return strptr_; }
private:
 size_t hash_{};
 size_t size_{};
 const char* strptr_{nullptr};
```

```
};
namespace std {
template <>
struct hash<PrehashedString> {
  constexpr auto operator()(const PrehashedString& s) const {
    return s.get_hash();
  }
};
} // namespace std
```

Note the template trick in the constructor. This forces the `PrehashedString` to only accept compile-time string literals. The reason for this is that the `PrehashedString` class does not own the `const char* ptr` and therefore we may only use it with string literals created at compile time:

```
// This compiles
auto prehashed_string = PrehashedString{"my_string"};
// This does not compile
// The prehashed_string object would be broken if the str is modified
auto str = std::string{"my_string"};
auto prehashed_string = PrehashedString{str.c_str()};
// This does not compile.
// The prehashed_string object would be broken if the strptr is deleted
auto* strptr = new char[5];
auto prehashed_string = PrehashedString{strptr};
```

So, now that we have everything in place, let's see how the compiler handles `PrehashedString`.

Evaluating PrehashedString

Here is a simple test function that returns the hash value for the string `"abc"` (used for simplicity):

```
auto test_prehashed_string() {
  const auto& hash_fn = std::hash<PrehashedString>{};
  const auto& str = PrehashedString("abc");
  return hash_fn(str);
}
```

Since our hash function simply sums the values, and the letters in `"abc"` have ASCII values of *a* = 97, *b* = 98, and *c* = 99, the assembler (generated by Clang) should output the sum 97 + 98 + 99 = 294 somewhere. Inspecting the assembler, we can see that the `test_prehashed_string()` function compiles to exactly one `return` statement, which returns `294`:

```
mov eax, 294
ret
```

This means that the whole `test_prehashed_string()` function has been executed at compile time; when the application executes, the hash sum has already been calculated!

Evaluating get_bitmap_resource() with PrehashedString

Let's return to our original `get_bitmap_resource()` function, the `std::string`, which was originally used and exchanged for a `PrehashedString`:

```
// Bitmap cache
auto get_bitmap_resource(const PrehashedString& path) -> const Bitmap&
{
  // Static storage of all loaded bitmaps
  static auto loaded_bitmaps =
    std::unordered_map<PrehashedString, Bitmap>{};
  // If the bitmap is already in loaded_bitmaps, return it
  if (loaded_bitmaps.count(path) > 0) {
    return loaded_bitmaps.at(path);
  }
  // The bitmap isn't already loaded, load and return it
  auto bitmap = load_bitmap_from_filesystem(path.c_str());
  loaded_bitmaps.emplace(path, std::move(bitmap));
  return loaded_bitmaps.at(path);
}
```

We also need a function to test with:

```
auto test_get_bitmap_resource() { return get_bitmap_resource("abc"); }
```

What we would like to know is whether this function precalculated the hash sum. Since `get_bitmap_resource()` does quite a lot (constructing a static `std::unordered_map`, inspecting the map, and so on), the resulting assembly is about 500 lines. Nevertheless, if our magic hash sum is found in the assembler, this means that we have succeeded.

When inspecting the assembler generated by Clang, we will find a line that corresponds to our hash sum, `294`:

```
.quad   294             # 0x126
```

To confirm this, we will change the string from `"abc"` to `"aaa"`, which should change this line in the assembler to 97 * 3 = 291, but everything else should be exactly the same.

We're doing this to make sure this wasn't just some other magic number that popped up, totally unrelated to the hash sum.

Inspecting the resulting assembler, we will find the desired result:

```
.quad   291             # 0x123
```

Everything, except this line, is the same, so we can safely assume that the hash is calculated at compile time.

The examples we have looked at demonstrate that we can use compile-time programming for very different things. Adding safety checks that can be verified at compile time allows us to find bugs without running the program and searching for errors with coverage tests. And moving expensive runtime operations to compile time makes our final program faster.

Summary

In this chapter, you have learned how to use metaprogramming to generate functions and values at compile time instead of runtime. You also discovered how to do this in a modern C++ way by using templates, the `constexpr`, `static_assert()`, and `if constexpr`, type traits, and concepts. Moreover, with constant string hashing, you saw how to use compile-time evaluation in a practical context.

In the next chapter, you will learn how to further expand your C++ toolbox so that you can create libraries by constructing hidden proxy objects.