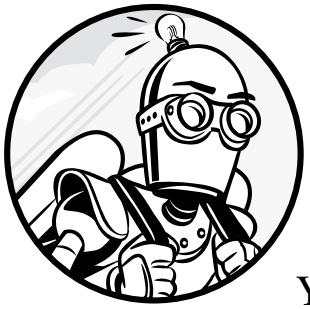
6

COMPILE-TIME POLYMORPHISM

The more you adapt, the more interesting you are.
—Martha Stewart



In this chapter, you'll learn how to achieve compile-time polymorphism with templates. You'll learn how to declare and use templates, enforce type safety, and survey some of the templates' more advanced usages. This chapter concludes with a comparison of runtime and compile-time polymorphism in C++.

Templates

C++ achieves compile-time polymorphism through *templates*. A template is a class or function with template parameters. These parameters can stand in for any type, including fundamental and user-defined types. When the compiler sees a template used with a type, it stamps out a bespoke template instantiation.

Template instantiation is the process of creating a class or a function from a template. Somewhat confusingly, you can also refer to "a template instantiation" as the result of the template instantiation process. Template instantiations are sometimes called concrete classes and concrete types.

The big idea is that, rather than copying and pasting common code all over the place, you write a single template; the compiler generates new template instances when it encounters a new combination of types in the template parameters.

Declaring Templates

You declare templates with a *template prefix*, which consists of the keyword `template` followed by angle brackets `< >`. Within the angle brackets, you place the declarations of one or more template parameters. You can declare template parameters using either the `typename` or `class` keywords followed by an identifier. For example, the template prefix `template<typename T>` declares that the template takes a template parameter `T`.

NOTE *The coexistence of the* `typename` *and* `class` *keywords is unfortunate and confusing. They mean the same thing. (They're both supported for historical reasons.) This chapter always uses* `typename`.

Template Class Definitions

Consider `MyTemplateClass` in Listing 6-1, which takes three template parameters: `X`, `Y`, and `Z`.

```
template❶<typename X, typename Y, typename Z> ❷
struct MyTemplateClass❸ {
  X foo(Y&); ❹
private:
  Z* member; ❺
};
```

Listing 6-1: A template class with three template parameters

The `template` keyword ❶ begins the template prefix, which contains the template parameters ❷. This `template` preamble leads to something special about the remaining declaration of `MyTemplateClass` ❸. Within `MyTemplateClass`, you use `X`, `Y`, and `Z` as if they were any fully specified type, like an `int` or a user-defined class.

The `foo` method takes a `Y` reference and returns an `X` ❹. You can declare members with types that include template parameters, like a pointer to `Z` ❺. Besides the special prefix beginning ❶, this template class is essentially identical to a non-template class.

Template Function Definitions

You can also specify template functions, like the `my_template_function` in Listing 6-2 that also takes three template parameters: `X`, `Y`, and `Z`.

```
template<typename X, typename Y, typename Z>
X my_template_function(Y& arg1, const Z* arg2) {
  --snip--
}
```

Listing 6-2: A template function with three template parameters

Within the body of `my_template_function`, you can use `arg1` and `arg2` however you'd like, as long as you return an object of type `X`.

Instantiating Templates

To instantiate a template class, use the following syntax:

```
tc_name❶<t_param1❷, t_param2, ...> my_concrete_class{ ... }❸;
```

The *`tc_name`* ❶ is where you place the template class's name. Next, you fill in your template parameters ❷. Finally, you treat this combination of template name and parameters as if it were a normal type: you use whatever initialization syntax you like ❸.

Instantiating a template function is similar:

```
auto result = tf_name❶<t_param1❷, t_param2, ...>(f_param1❸, f_param2, ...);
```

The *`tf_name`* ❶ is where you put the template function's name. You fill in the parameters just as you do for template classes ❷. You use the combination of template name and parameters as if it were a normal type. You invoke this template function instantiation with parentheses and function parameters ❸.

All this new notation might be daunting to a newcomer, but it's not so bad once you get used to it. In fact, it's used in a set of language features called named conversion functions.

Named Conversion Functions

Named conversions are language features that explicitly convert one type into another type. You use named conversions sparingly in situations where you cannot use implicit conversions or constructors to get the types you need.

All named conversions accept a single object parameter, which is the object you want to cast *`object-to-cast`*, and a single type parameter, which is the type you want to cast to *`desired-type`*:

```
named-conversion<desired-type>(object-to-cast)
```

For example, if you need to modify a `const` object, you would first need to cast away the `const` qualifier. The named conversion function `const_cast` allows you to perform this operation. Other named conversions help you to reverse implicit casts (`static_cast`) or reinterpret memory with a different type (`reinterpret_cast`).

NOTE *Although named conversion functions aren't technically template functions, they are conceptually very close to templates—a relationship reflected in their syntactic similarity.*

const_cast

The `const_cast` function shucks away the `const` modifier, allowing the modification of `const` values. The *`object-to-cast`* is of some `const` type, and the *`desired-type`* is that type minus the `const` qualifier.

Consider the `carbon_thaw` function in Listing 6-3, which takes a `const` reference to an `encased_solo` argument.

```
void carbon_thaw(const❶ int& encased_solo) {
  //encased_solo++; ❷ // Compiler error; modifying const
  auto& hibernation_sick_solo = const_cast❸<int&❹>(encased_solo❺);
  hibernation_sick_solo++; ❻
}
```

Listing 6-3: A function using `const_cast`. Uncommenting yields a compiler error.

The `encased_solo` parameter is `const` ❶, so any attempt to modify it ❷ would result in a compiler error. You use `const_cast` ❸ to obtain the non-`const` reference `hibernation_sick_solo`. The `const_cast` takes a single template parameter, the type you want to cast into ❹. It also takes a function parameter, the object you want to remove `const` from ❺. You're then free to modify the `int` pointed to by `encased_solo` via the new, non-`const` reference ❻.

Only use `const_cast` to obtain write access to `const` objects. Any other type conversion will result in a compiler error.

NOTE *Trivially, you can use `const_cast` to add `const` to an object's type, but you shouldn't because it's verbose and unnecessary. Use an implicit cast instead. In Chapter 7, you'll learn what the `volatile` modifier is. You can also use `const_cast` to remove the `volatile` modifier from an object.*

static_cast

The `static_cast` reverses a well-defined implicit conversion, such as an integer type to another integer type. The *`object-to-cast`* is of some type that the *`desired-type`* implicitly converts to. The reason you might need `static_cast` is that, generally, implicit casts aren't reversible.

The program in Listing 6-4 defines an `increment_as_short` function that takes a `void` pointer argument. It employs a `static_cast` to create a `short` pointer from this argument, increment the pointed-to `short`, and return the result. In some low-level applications, such as network programming

or handling binary file formats, you might need to interpret raw bytes as an integer type.

```
#include <cstdio>

short increment_as_short(void*❶ target) {
  auto as_short = static_cast❷<short*❸>(target❹);
  *as_short = *as_short + 1;
  return *as_short;
}

int main() {
  short beast{ 665 };
  auto mark_of_the_beast = increment_as_short(&beast);
  printf("%d is the mark_of_the_beast.", mark_of_the_beast);
}
```

```
666 is the mark_of_the_beast.
```

Listing 6-4: A program using `static_cast`

The `target` parameter is a `void` pointer ❶. You employ `static_cast` to cast `target` into a `short*` ❷. The template parameter is the desired type ❸, and the function parameter is the object you want to cast into ❹.

Notice that the implicit conversion of `short*` to `void*` is well defined. Attempting ill-defined conversions with `static_cast`, such as converting a `char*` to a `float*`, will result in a compiler error:

```
float on = 3.5166666666;
auto not_alright = static_cast<char*>(&on); // Bang!
```

To perform such chainsaw juggling, you need to use `reinterpret_cast`.

reinterpret_cast

Sometimes in low-level programming, you must perform type conversions that are not well defined. In system programming and especially in embedded environments, you often need complete control over how to interpret memory. The `reinterpret_cast` gives you such control, but ensuring the correctness of these conversions is entirely your responsibility.

Suppose your embedded device keeps an `unsigned long` timer at memory address 0x1000. You could use `reinterpret_cast` to read from the timer, as demonstrated in Listing 6-5.

```
#include <cstdio>

int main() {
  auto timer = reinterpret_cast❶<const unsigned long*❷>(0x1000❸);
  printf("Timer is %lu.", *timer);
}
```

Listing 6-5: A program using `reinterpret_cast`. This program will compile, but you should expect a runtime crash unless 0x1000 is readable.

The `reinterpret_cast` ❶ takes a type parameter corresponding to the desired pointer type ❷ and the memory address the result should point to ❸.

Of course, the compiler has no idea whether the memory at address 0x1000 contains an `unsigned long`. It's entirely up to you to ensure correctness. Because you're taking full responsibility for this very dangerous construction, the compiler forces you to employ `reinterpret_cast`. You couldn't, for example, replace the initialization of `timer` with the following line:

```
const unsigned long* timer{ 0x1000 };
```

The compiler will grumble about converting an `int` to a pointer.

narrow_cast

Listing 6-6 illustrates a custom `static_cast` that performs a runtime check for *narrowing*. Narrowing is a loss in information. Think about converting from an `int` to a `short`. As long as the value of `int` fits into a `short`, the conversion is reversible and no narrowing occurs. If the value of `int` is too big for the `short`, the conversion isn't reversible and results in narrowing.

Let's implement a named conversion called `narrow_cast` that checks for narrowing and throws a `runtime_error` if it's detected.

```
#include <stdexcept>

template <typename To❶, typename From❷>
To❸ narrow_cast(From❹ value) {
  const auto converted = static_cast<To>(value); ❺
  const auto backwards = static_cast<From>(converted); ❻
  if (value != backwards) throw std::runtime_error{ "Narrowed!" }; ❼
  return converted; ❽
}
```

Listing 6-6: A `narrow_cast` definition

The `narrow_cast` function template takes two template parameters: the type you're casting `To` ❶ and the type you're casting `From` ❷. You can see these template parameters in action as the return type of the function ❸ and the type of the parameter value ❹. First, you perform the requested conversion using `static_cast` to yield `converted` ❺. Next, you perform the conversion in the opposite direction (from `converted` to type `From`) to yield `backwards` ❻. If `value` doesn't equal `backwards`, you've narrowed, so you throw an exception ❼. Otherwise, you return `converted` ❽.

You can see `narrow_cast` in action in Listing 6-7.

```
#include <cstdio>
#include <stdexcept>

template <typename To, typename From>
To narrow_cast(From value) {
  --snip--
}
```

```
int main() {
  int perfect{ 496 }; ❶
  const auto perfect_short = narrow_cast<short>(perfect); ❷
  printf("perfect_short: %d\n", perfect_short); ❸
  try {
    int cyclic{ 142857 }; ❹
    const auto cyclic_short = narrow_cast<short>(cyclic); ❺
    printf("cyclic_short: %d\n", cyclic_short);
  } catch (const std::runtime_error& e) {
    printf("Exception: %s\n", e.what()); ❻
  }
}
```

```
perfect_short: 496 ❸
Exception: Narrowed! ❻
```

Listing 6-7: A program using `narrow_cast`. (The output comes from an execution on Windows 10 x64.)

You first initialize `perfect` to 496 ❶ and then `narrow_cast` it to the short `perfect_short` ❷. This proceeds without exception because the value 496 fits easily into a 2-byte short on Windows 10 x64 (maximum value 32767). You see the output as expected ❸. Next, you initialize `cyclic` to 142857 ❹ and attempt to `narrow_cast` to the short `cyclic_short` ❺. This throws a `runtime_error` because 142857 is greater than the short's maximum value of 32767. The check within `narrow_cast` will fail. You see the exception printed in the `output` ❻.

Notice that you need to provide only a single template parameter, the return type, upon instantiation ❶❹. The compiler can deduce the `From` parameter based on usage.

mean: A Template Function Example

Consider the function in Listing 6-8 that computes the mean of a `double` array using the sum-and-divide approach.

```
#include <cstddef>

double mean(const double* values, size_t length) {
  double result{}; ❶
  for(size_t i{}; i<length; i++) {
    result += values[i]; ❷
  }
  return result / length; ❸
}
```

Listing 6-8: A function for computing the mean of an array

You initialize a `result` variable to zero ❶. Next, you sum over `values` by iterating over each index `i`, adding the corresponding element to `result` ❷. Then you divide `result` by `length` and `return` ❸.

Genericizing mean

Suppose you want to support `mean` calculations for other numeric types, such as `float` or `long`. You might be thinking, "That's what function overloads are for!" Essentially, you would be correct.

Listing 6-9 overloads `mean` to accept a `long` array. The straightforward approach is to copy and paste the original, then replace instances of `double` with `long`.

```
#include <cstddef>

long❶ mean(const long*❷ values, size_t length) {
  long result{}; ❸
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
  return result / length;
}
```

Listing 6-9: An overload of Listing 6-8 accepting a `long` array

That sure is a lot of copying and pasting, and you've changed very little: the return type ❶, the function argument ❷, and `result` ❸.

This approach doesn't scale as you add more types. What if you want to support other integral types, such as `short` types or `uint_64` types? What about `float` types? What if, later on, you want to refactor some logic in `mean`? You're in for a lot of tedious and error-prone maintenance.

There are three changes to `mean` in Listing 6-9, and all of them involve finding and replacing `double` types with `long` types. Ideally, you could have the compiler automatically generate versions of the function for you whenever it encounters usage with a different type. The key is that none of the logic changes—only the types.

What you need to solve this copy-and-paste problem is *generic programming*, a programming style where you program with yet-to-be-specified types. You achieve generic programming using the support C++ has for templates. Templates allow the compiler to instantiate a custom class or function based on the types in use.

Now that you know how to declare templates, consider the `mean` function again. You still want `mean` to accept a wide range of types—not just `double` types—but you don't want to have to copy and paste the same code over and over again.

Consider how you can refactor Listing 6-8 into a template function, as demonstrated in Listing 6-10.

```
#include <cstddef>

template<typename T> ❶
T❷ mean(constT*❸ values, size_t length) {
  T❹ result{};
  for(size_t i{}; i<length; i++) {
    result += values[i];
```

```
  }
  return result / length;
}
```

Listing 6-10: Refactoring Listing 6-8 into a template function

Listing 6-10 kicks off with a template prefix ❶. This prefix communicates a single template parameter `T`. Next, you update `mean` to use `T` instead of `double` ❷❸❹.

Now you can use `mean` with many different types. Each time the compiler encounters a usage of `mean` with a new type, it performs template instantiation. It's *as if* you had done the copy-paste-and-replace-types procedure, but the compiler is much better at doing detail-oriented, monotonous tasks than you are. Consider the example in Listing 6-11, which computes means for `double`, `float`, and `size_t` types.

```
#include <cstddef>
#include <cstdio>

template<typename T>
T mean(const T* values, size_t length) {
  --snip--
}

int main() {
  const double nums_d[] { 1.0, 2.0, 3.0, 4.0 };
  const auto result1 = mean<double>(nums_d, 4); ❶
  printf("double: %f\n", result1);

  const float nums_f[] { 1.0f, 2.0f, 3.0f, 4.0f };
  const auto result2 = mean<float>(nums_f, 4); ❷
  printf("float: %f\n", result2);

  const size_t nums_c[] { 1, 2, 3, 4 };
  const auto result3 = mean<size_t>(nums_c, 4); ❸
  printf("size_t: %zu\n", result3);
}
```

```
double: 2.500000
float: 2.500000
size_t: 2
```

Listing 6-11: A program using the template function `mean`

Three templates are instantiated ❶❷❸; it's as if you generated the overloads isolated in Listing 6-12 by hand. (Each template instantiation contains types, shown in bold, where the compiler substituted a type for a template parameter.)

```
double mean(const double* values, size_t length) {
  double result{};
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
```

```
  return result / length;
}

float mean(const float* values, size_t length) {
  float result{};
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
  return result / length;
}

char mean(const char* values, size_t length) {
  char result{};
  for(size_t i{}; i<length; i++) {
    result += values[i];
  }
  return result / length;
}
```

Listing 6-12: The template instantiations for Listing 6-11

The compiler has done a lot of work for you, but you might have noticed that you had to type the pointed-to array type twice: once to declare an array and again to specify a template parameter. This gets tedious and can cause errors. If the template parameter doesn't match, you'll likely get a compiler error or cause unintended casting.

Fortunately, you can generally omit the template parameters when invoking a template function. The process that the compiler uses to determine the correct template parameters is called *template type deduction*.

Template Type Deduction

Generally, you don't have to provide template function parameters. The compiler can deduce them from usage, so a rewrite of Listing 6-11 without them is shown in Listing 6-13.

```
#include <cstddef>
#include <cstdio>

template<typename T>
T mean(const T* values, size_t length) {
  --snip--
}

int main() {
  const double nums_d[] { 1.0, 2.0, 3.0, 4.0 };
  const auto result1 = mean(nums_d, 4); ❶
  printf("double: %f\n", result1);

  const float nums_f[] { 1.0f, 2.0f, 3.0f, 4.0f };
  const auto result2 = mean(nums_f, 4); ❷
  printf("float: %f\n", result2);

  const size_t nums_c[] { 1, 2, 3, 4 };
```

```
  const auto result3 = mean(nums_c, 4); ❸
  printf("size_t: %zu\n", result3);
}
```

```
double: 2.500000
float: 2.500000
size_t: 2
```

Listing 6-13: A refactor of Listing 6-11 without explicit template parameters

It's clear from usage that the template parameters are `double` ❶, `float` ❷, and `size_t` ❸.

NOTE *Template type deduction mostly works the way you might expect, but there is some nuance you'll want to become familiar with if you're writing a lot of generic code. For more information, see the ISO standard [temp]. Also, refer to Item 1 of* Effective Modern C++ *by Scott Meyers and Section 23.5.1 of* The C++ Programming Language*, 4th Edition, by Bjarne Stroustrup.*

Sometimes, template arguments cannot be deduced. For example, if a template function's return type is a template argument, you must specify template arguments explicitly.

SimpleUniquePointer: A Template Class Example

A *unique pointer* is an RAII wrapper around a free-store-allocated object. As its name suggests, the unique pointer has a single owner at a time, so when a unique pointer's lifetime ends, the pointed-to object gets destructed.

The underlying object's type in unique pointers doesn't matter, making them a prime candidate for a template class. Consider the implementation in Listing 6-14.

```
template <typename T> ❶
struct SimpleUniquePointer {
  SimpleUniquePointer() = default; ❷
  SimpleUniquePointer(T* pointer)
    : pointer{ pointer } { ❸
  }
  ~SimpleUniquePointer() { ❹
    if(pointer) delete pointer;
  }
  SimpleUniquePointer(const SimpleUniquePointer&) = delete;
  SimpleUniquePointer& operator=(const SimpleUniquePointer&) = delete; ❺
  SimpleUniquePointer(SimpleUniquePointer&& other) noexcept ❻
    : pointer{ other.pointer } {
    other.pointer = nullptr;
  }
  SimpleUniquePointer& operator=(SimpleUniquePointer&& other) noexcept { ❼
    if(pointer) delete pointer;
    pointer = other.pointer;
    other.pointer = nullptr;
    return *this;
```

```
  }
  T* get() { ❽
    return pointer;
  }
private:
  T* pointer;
};
```

Listing 6-14: A simple unique pointer implementation

You announce the template class with a template prefix ❶, which establishes `T` as the wrapped object's type. Next, you specify a default constructor using the `default` keyword ❷. (Recall from Chapter 4 that you need `default` when you want both a default constructor *and* a non-default constructor.) The generated default constructor will set the private member `T*` pointer to `nullptr` thanks to default initialization rules. You have a non-default constructor that takes a `T*` and sets the private member pointer ❸. Because the pointer is possibly `nullptr`, the destructor checks before deleting ❹.

Because you want to allow only a single owner of the pointed-to object, you `delete` the copy constructor and the copy-assignment operator ❺. This prevents double-free issues, which were discussed in Chapter 4. However, you can make your unique pointer moveable by adding a move constructor ❻. This steals the value of `pointer` from `other` and then sets the pointer of `other` to `nullptr`, handing responsibility of the pointed-to object to `this`. Once the move constructor returns, the moved-from object is destroyed. Because the moved-from object's pointer is set to `nullptr`, the destructor will not delete the pointed-to object.

The possibility that `this` already owns an object complicates the move assignment ❼. You must check explicitly for prior ownership, because failure to delete a pointer leaks a resource. After this check, you perform the same operations as in the copy constructor: you set `pointer` to the value of `other.pointer` and then set `other.pointer` to `nullptr`. This ensures that the moved-from object doesn't delete the pointed-to object.

You can obtain direct access to the underlying pointer by calling the `get` method ❽.

Let's enlist our old friend `Tracer` from Listing 4-5 to investigate `SimpleUniquePointer`. Consider the program in Listing 6-15.

```
#include <cstdio>
#include <utility>

template <typename T>
struct SimpleUniquePointer {
  --snip--
};

struct Tracer {
  Tracer(const char* name) : name{ name } {
    printf("%s constructed.\n", name); ❶
  }
  ~Tracer() {
```

```
    printf("%s destructed.\n", name); ❷
  }
private:
  const char* const name;
};

void consumer(SimpleUniquePointer<Tracer> consumer_ptr) {
  printf("(cons) consumer_ptr: 0x%p\n", consumer_ptr.get()); ❸
}

int main() {
  auto ptr_a = SimpleUniquePointer(new Tracer{ "ptr_a" });
  printf("(main) ptr_a: 0x%p\n", ptr_a.get()); ❹
  consumer(std::move(ptr_a));
  printf("(main) ptr_a: 0x%p\n", ptr_a.get()); ❺
}
--------------------------------------------------------------------------------
ptr_a constructed. ❶
(main) ptr_a: 0x000001936B5A2970 ❹
(cons) consumer_ptr: 0x000001936B5A2970 ❸
ptr_a destructed. ❷
(main) ptr_a: 0x0000000000000000 ❺
```

Listing 6-15: A program investigating `SimpleUniquePointers` with the `Tracer` class

First, you dynamically allocate a `Tracer` with the message `ptr_a`. This prints the first message ❶. You use the resulting `Tracer` pointer to construct a `SimpleUniquePointer` called `ptr_a`. Next, you use the `get()` method of `ptr_a` to retrieve the address of its `Tracer`, which you print ❹. Then you use `std::move` to relinquish the `Tracer` of `ptr_a` to the `consumer` function, which moves `ptr_a` into the `consumer_ptr` argument.

Now, `consumer_ptr` owns the `Tracer`. You use the `get()` method of `consumer_ptr` to retrieve the address of `Tracer`, then print ❸. Notice this address matches the one printed at ❹. When `consumer` returns, `consumer_ptr` dies because its storage duration is the scope of `consumer`. As a result, `ptr_a` gets destructed ❷.

Recall that `ptr_a` is in a moved-from state—you moved its `Tracer` into `consumer`. You use the `get()` method of `ptr_a` to illustrate that it now holds a `nullptr` ❺.

Thanks to `SimpleUniquePointer`, you won't leak a dynamically allocated object; also, because the `SimpleUniquePointer` is just carrying around a pointer under the hood, move semantics are efficient.

NOTE *The `SimpleUniquePointer` is a pedagogical implementation of the stdlib's `std::unique_ptr`, which is a member of the family of RAII templates called smart pointers. You'll learn about these in Part II.*

Type Checking in Templates

Templates are type safe. During template instantiation, the compiler pastes in the template parameters. If the resulting code is incorrect, the compiler will not generate the instantiation.