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(T*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: %zd\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: %zd\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.

Consider the template function in Listing 6-16, which squares an element and returns the result.

```
template<typename T>
T square(T value) {
  return value * value; ❶
}
```

Listing 6-16: A template function that squares a value

The T has a silent requirement: it must support multiplication ❶.

If you try to use square with, say, a char*, the compilation will fail, as shown in Listing 6-17.

```
template<typename T>
T square(T value) {
  return value * value;
}

int main() {
  char my_char{ 'Q' };
  auto result = square(&my_char); ❶ // Bang!
}
```

Listing 6-17: A program with a failed template instantiation. (This program fails to compile.)

Pointers don't support multiplication, so template initialization fails ❶.

The square function is trivially small, but the failed template initialization's error message isn't. On MSVC v141, you get this:

```
main.cpp(3): error C2296: '*': illegal, left operand has type 'char *'
main.cpp(8): note: see reference to function template instantiation 'T
*square<char*>(T)' being compiled
        with
        [
            T=char *
        ]
main.cpp(3): error C2297: '*': illegal, right operand has type 'char *'
```

And on GCC 7.3, you get this:

```
main.cpp: In instantiation of 'T square(T) [with T = char*]':
main.cpp:8:32:   required from here
main.cpp:3:16: error: invalid operands of types 'char*' and 'char*' to binary
'operator*'
   return value * value;
          ~~~~~~^~~~~~~
```

These error messages exemplify the notoriously cryptic error messages emitted by template initialization failures.

Although template instantiation ensures type safety, the checking happens very late in the compilation process. When the compiler instantiates a template, it pastes the template parameter types into the template. After type insertion, the compiler attempts to compile the result. If instantiation fails, the compiler emits the dying words inside the template instantiation.

C++ template programming shares similarities with *duck-typed languages*. Duck-typed languages (like Python) defer type checking until runtime. The underlying philosophy is that if an object looks like a duck and quacks like a duck, then it must be type duck. Unfortunately, this means you can't generally know whether an object supports a particular operation until you execute the program.

With templates, you cannot know whether an instantiation will succeed until you try to compile it. Although duck-typed languages might blow up at runtime, templates might blow up at compile time.

This situation is widely regarded as unacceptable by right-thinking people in the C++ community, so there is a splendid solution called concepts.

Concepts

Concepts constrain template parameters, allowing for parameter checking at the point of instantiation rather than the point of first use. By catching usage issues at the point of instantiation, the compiler can give you a friendly, informative error code—for example, "You tried to instantiate this template with a `char*`, but this template requires a type that supports multiplication."

Concepts allow you to express requirements on template parameters directly in the language.

Unfortunately, concepts aren't yet officially part of the C++ standard, although they've been voted into C++ 20. At press time, GCC 6.0 and later support the Concepts Technical Specification, and Microsoft is actively working toward implementing concepts in its C++ compiler, MSVC. Regardless of its unofficial status, it's worth exploring concepts in some detail for a few reasons:

- They'll fundamentally change the way you achieve compile-time polymorphism. Familiarity with concepts will pay major dividends.
- They provide a conceptual framework for understanding some of the makeshift solutions that you can put in place to get better compiler errors when templates are misused.
- They provide an excellent conceptual bridge from compile-time templates to interfaces, the primary mechanism for runtime polymorphism (covered in Chapter 5).
- If you can use GCC 6.0 or later, concepts *are* available by turning on the `-fconcepts` compiler flag.

WARNING *C++ 20's final concept specification will almost certainly deviate from the Concepts Technical Specification. This section presents concepts as specified in the Concepts Technical Specification so you can follow along.*

Defining a Concept

A concept is a template. It's a constant expression involving template arguments, evaluated at compile time. Think of a concept as one big *predicate*: a function that evaluates to `true` or `false`.

If a set of template parameters meets the criteria for a given concept, that concept evaluates to `true` when instantiated with those parameters; otherwise, it will evaluate to `false`. When a concept evaluates to `false`, template instantiation fails.

You declare concepts using the keyword `concept` on an otherwise familiar template function definition:

```
template<typename T1, typename T2, ...>
concept bool ConceptName() {
  --snip--
}
```

Type Traits

Concepts validate type parameters. Within concepts, you manipulate types to inspect their properties. You can hand roll these manipulations, or you can use the type support library built into the stdlib. The library contains utilities for inspecting type properties. These utilities are collectively called *type traits*. They're available in the `<type_traits>` header and are part of the `std` namespace. Table 6-1 lists some commonly used type traits.

NOTE *See Chapter 5.4 of* The C++ Standard Library, *2nd Edition, by Nicolai M. Josuttis for an exhaustive listing of type traits available in the stdlib.*

Table 6-1: Selected Type Traits from the `<type_traits>` Header

Type trait	Checks if template argument is . . .
`is_void`	`void`
`is_null_pointer`	`nullptr`
`is_integral`	`bool`, a `char` type, an `int` type, a `short` type, a `long` type, or a `long long` type
`is_floating_point`	`float`, `double`, or `long double`
`is_fundamental`	Any of `is_void`, `is_null_pointer`, `is_integral`, or `is_floating_point`
`is_array`	An array; that is, a type containing square brackets `[]`
`is_enum`	An enumeration type (`enum`)
`is_class`	A class type (but not a union type)
`is_function`	A function
`is_pointer`	A pointer; function pointers count, but pointers to class members and `nullptr` do not
`is_reference`	A reference (either lvalue or rvalue)
`is_arithmetic`	`is_floating_point` or `is_integral`

Type trait	Checks if template argument is . . .
is_pod	A plain-old-data type; that is, a type that can be represented as a data type in plain C
is_default_constructible	Default constructible; that is, it can be constructed without arguments or initialization values
is_constructible	Constructible with the given template parameters: this type trait allows the user to provide additional template parameters beyond the type under consideration
is_copy_constructible	Copy constructible
is_move_constructible	Move constructible
is_destructible	Destructible
is_same	The same type as the additional template parameter type (including const and volatile modifiers)
is_invocable	Invocable with the given template parameters: this type trait allows the user to provide additional template parameters beyond the type under consideration

Each type trait is a template class that takes a single template parameter, the type you want to inspect. You extract the results using the template's static member `value`. This member equals `true` if the type parameter meets the criteria; otherwise, it's `false`.

Consider the type trait classes `is_integral` and `is_floating_point`. These are useful for checking if a type is (you guessed it) integral or floating point. Both of these templates take a single template parameter. The example in Listing 6-18 investigates type traits with several types.

```
#include <type_traits>
#include <cstdio>
#include <cstdint>

constexpr const char* as_str(bool x) { return x ? "True" : "False"; } ❶

int main() {
  printf("%s\n", as_str(std::is_integral<int>::value)); ❷
  printf("%s\n", as_str(std::is_integral<const int>::value)); ❸
  printf("%s\n", as_str(std::is_integral<char>::value)); ❹
  printf("%s\n", as_str(std::is_integral<uint64_t>::value)); ❺
  printf("%s\n", as_str(std::is_integral<int&>::value)); ❻
  printf("%s\n", as_str(std::is_integral<int*>::value)); ❼
  printf("%s\n", as_str(std::is_integral<float>::value)); ❽
}
--------------------------------------------------------------------------------
True ❷
True ❸
True ❹
True ❺
False ❻
False ❼
False ❽
```

Listing 6-18: A program using type traits

Listing 6-18 defines the convenience function `as_str` ❶ to print Boolean values with the string `True` or `False`. Within `main`, you print the result of various type trait instantiations. The template parameters `int` ❷, `const int` ❸, `char` ❹, and `uint64_t` ❺ all return `true` when passed to `is_integral`. Reference types ❻❼ and floating-point types ❽ return `false`.

NOTE *Recall that `printf` doesn't have a format specifier for `bool`. Rather than using the integer format specifier `%d` as a stand-in, Listing 6-18 employs the `as_str` function, which returns the string literal `True` or `False` depending on the value of the `bool`. Because these values are string literals, you can capitalize them however you like.*

Type traits are often the building blocks for a concept, but sometimes you need more flexibility. Type traits tell you *what* types are, but sometimes you must also specify *how* the template will use them. For this, you use requirements.

Requirements

Requirements are ad hoc constraints on template parameters. Each concept can specify any number of requirements on its template parameters. Requirements are encoded into requires expressions denoted by the `requires` keyword followed by function arguments and a body.

A sequence of syntactic requirements comprises the requirements expression's body. Each syntactic requirement puts a constraint on the template parameters. Together, requires expressions have the following form:

```
requires (arg-1, arg-2, ...❶) {
  { expression1❷ } -> return-type1❸;
  { expression2 } -> return-type2;
  --snip--
}
```

Requires expressions take arguments that you place after the `requires` keyword ❶. These arguments have types derived from template parameters. The syntactic requirements follow, each denoted with `{ } ->`. You put an arbitrary expression within each of the braces ❷. This expression can involve any number of the arguments to the argument expression.

If an instantiation causes a syntactic expression not to compile, that syntactic requirement fails. Supposing the expression evaluates without error, the next check is whether the return type of that expression matches the type given after the arrow `->` ❸. If the expression result's evaluated type can't implicitly convert to the return type ❸, the syntactic requirement fails.

If any of the syntactic requirements fail, the requires expression evaluates to `false`. If all of the syntactic requirements pass, the requires expression evaluates to `true`.

Suppose you have two types, `T` and `U`, and you want to know whether you can compare objects of these types using the equality `==` and inequality `!=` operators. One way to encode this requirement is to use the following expression.

```
// T, U are types
requires (T t, U u) {
  { t == u } -> bool; // syntactic requirement 1
  { u == t } -> bool; // syntactic requirement 2
  { t != u } -> bool; // syntactic requirement 3
  { u != t } -> bool; // syntactic requirement 4
}
```

The requires expression takes two arguments, one each of types `T` and `U`. Each of the syntactic requirements contained in the requires expression is an expression using `t` and `u` with either `==` or `!=`. All four syntactic requirements enforce a `bool` result. Any two types that satisfy this requires expression are guaranteed to support comparison with `==` and `!=`.

Building Concepts from Requires Expressions

Because requires expressions are evaluated at compile time, concepts can contain any number of them. Try to construct a concept that guards against the misuse of `mean`. Listing 6-19 annotates some of the implicit requirements used earlier in Listing 6-10.

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

Listing 6-19: A relisting of 6-10 with annotations for some implicit requirements on `T`

You can see three requirements implied by this code:

- `T` must be default constructible ❶.
- `T` supports `operator+=` ❷.
- Dividing a `T` by a `size_t` yields a `T` ❸.

From these requirements, you could create a concept called `Averageable`, as demonstrated in Listing 6-20.

```
template<typename T>
concept bool Averageable() {
  return std::is_default_constructible<T>::value ❶
    && requires (T a, T b) {
      { a += b } -> T; ❷
      { a / size_t{ 1 } } -> T; ❸
    };
}
```

Listing 6-20: An `Averageable` concept. Annotations are consistent with the requirements and the body of `mean`.

You use the type trait is_default_constructible to ensure that T is default constructible ❶, that you can add two T types ❷, and that you can divide a T by a size_t ❸ and get a result of type T.

Recall that concepts are just predicates; you're building a Boolean expression that evaluates to true when the template parameters are supported and false when they're not. The concept is composed of three Boolean expressions AND-ed (&&) together: two type traits ❶❸ and a requires expression. If any of the three returns false, the concept's constraints are not met.

Using Concepts

Declaring concepts is a lot more work than using them. To use a concept, just use the concept's name in place of the typename keyword.

For example, you can refactor Listing 6-13 with the Averageable concept, as shown in Listing 6-21.

```
#include <cstddef>
#include <type_traits>

template<typename T>
concept bool Averageable() { ❶
  --snip--
}

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

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

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

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

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

Listing 6-21: A refactor of Listing 6-13 using Averageable

After defining Averageable ❶, you just use it in place of typename ❷. No further modification is necessary. The code generated from compiling Listing 6-13 is identical to the code generated from compiling Listing 6-21.

The payoff is when you get to try to use mean with a type that is not Averageable: you get a compiler error at the point of instantiation. This produces much better compiler error messages than you would obtain from a raw template.

Look at the instantiation of mean in Listing 6-22 where you "accidentally" try to average an array of double pointers.

```
--snip–
int main() {
  auto value1 = 0.0;
  auto value2 = 1.0;
  const double* values[] { &value1, &value2 };
  mean(values❶, 2);
}
```

Listing 6-22: A bad template instantiation using a non-Averageable argument

There are several problems with using values ❶. What can the compiler tell you about those problems?

Without concepts, GCC 6.3 produces the error message shown in Listing 6-23.

```
<source>: In instantiation of 'T mean(const T*, size_t) [with T = const
double*; size_t = long unsigned int]':
<source>:17:17:   required from here
<source>:8:12: error: invalid operands of types 'const double*' and 'const
double*' to binary 'operator+'
     result += values[i]; ❶
     ~~~~~~~^~~~~~~~~~
<source>:8:12: error:   in evaluation of 'operator+=(const double*, const
double*)'
<source>:10:17: error: invalid operands of types 'const double*' and 'size_t'
{aka 'long unsigned int'} to binary 'operator/'
   return result / length; ❷
          ~~~~~~~^~~~~~~~
```

Listing 6-23: Error message from GCC 6.3 when compiling Listing 6-22

You might expect a casual user of mean to be extremely confused by this error message. What is i ❶? Why is a const double* involved in division ❷?

Concepts provide a far more illuminating error message, as Listing 6-24 demonstrates.

```
<source>: In function 'int main()':
<source>:28:17: error: cannot call function 'T mean(const T*, size_t) [with T
= const double*; size_t = long unsigned int]'
   mean(values, 2); ❶
                 ^
<source>:16:3: note:   constraints not satisfied
 T mean(const T* values, size_t length) {
   ^~~~
<source>:6:14: note: within 'template<class T> concept bool Averageable()
[with T = const double*]'
```

```
 concept bool Averageable() {
              ^~~~~~~~~~~
<source>:6:14: note:     with 'const double* a'
<source>:6:14: note:     with 'const double* b'
<source>:6:14: note: the required expression '(a + b)' would be ill-formed ❷
<source>:6:14: note: the required expression '(a / b)' would be ill-formed ❸
```

Listing 6-24: Error message from GCC 7.2 when compiling Listing 6-22 with concepts enabled

This error message is fantastic. The compiler tells you which argument (`values`) didn't meet a constraint ❶. Then it tells you that `values` is not `Averageable` because it doesn't satisfy two required expressions ❷❸. You know immediately how to modify your arguments to make this template instantiation successful.

When concepts incorporate into the C++ standard, it's likely that the stdlib will include many concepts. The design goal of concepts is that a programmer shouldn't have to define very many concepts on their own; rather, they should be able to combine concepts and ad hoc requirements within a template prefix. Table 6-2 provides a partial listing of some concepts you might expect to be included; these are borrowed from Andrew Sutton's implementation of concepts in the Origins Library.

NOTE *See* https://github.com/asutton/origin/ *for more information on the Origins Library. To compile the examples that follow, you can install Origins and use GCC version 6.0 or later with the* `-fconcepts` *flag.*

Table 6-2: The Concepts Contained in the Origins Library

Concept	A type that . . .
`Conditional`	Can be explicitly converted to `bool`
`Boolean`	Is `Conditional` and supports `!`, `&&`, and `||` Boolean operations
`Equality_comparable`	Supports `==` and `!=` operations returning a `Boolean`
`Destructible`	Can be destroyed (compare `is_destructible`)
`Default_constructible`	Is default constructible (compare `is_default_constructible`)
`Movable`	Supports move semantics: it must be move assignable and move constructible (compare `is_move_assignable`, `is_move_constructible`)
`Copyable`	Supports copy semantics: it must be copy assignable and copy constructible (compare `is_copy_assignable`, `is_copy_constructible`)
`Regular`	Is default constructible, copyable, and `Equality_comparable`
`Ordered`	Is `Regular` and is totally ordered (essentially, it can be sorted)
`Number`	Is `Ordered` and supports math operations like `+`, `-`, `/`, and `*`
`Function`	Supports invocation; that is, you can call it (compare `is_invocable`)
`Predicate`	Is a `Function` and returns `bool`
`Range`	Can be iterated over in a range-based `for` loop

There are several ways to build constraints into a template prefix. If a template parameter is only used to declare the type of a function parameter, you can omit the template prefix entirely:

```
return-type function-name(Concept1❶ arg-1, ...) {
  --snip--
}
```

Because you use a concept rather than a `typename` to define an argument's type ❶, the compiler knows that the associated function is a template. You are even free to mix concepts and concrete types in the argument list. In other words, whenever you use a concept as part of a function definition, that function becomes a template.

The template function in Listing 6-25 takes an array of `Ordered` elements and finds the minimum.

```
#include <origin/core/concepts.hpp>
size_t index_of_minimum(Ordered❶* x, size_t length) {
  size_t min_index{};
  for(size_t i{ 1 }; i<length; i++) {
    if(x[i] < x[min_index]) min_index = i;
  }
  return min_index;
}
```

Listing 6-25: A template function using the `Ordered` concept

Even though there's no template prefix, `index_of_minimum` is a template because `Ordered` ❶ is a concept. This template can be instantiated in the same way as any other template function, as demonstrated in Listing 6-26.

```
#include <cstdio>
#include <cstdint>
#include <origin/core/concepts.hpp>

struct Goblin{};

size_t index_of_minimum(Ordered* x, size_t length) {
  --snip--
}

int main() {
  int x1[] { -20, 0, 100, 400, -21, 5123 };
  printf("%zd\n", index_of_minimum(x1, 6)); ❶

  unsigned short x2[] { 42, 51, 900, 400 };
  printf("%zd\n", index_of_minimum(x2, 4)); ❷

  Goblin x3[] { Goblin{}, Goblin{} };
  //index_of_minimum(x3, 2); ❸ // Bang! Goblin is not Ordered.
}
```

```
4 ❶
0 ❷
```

Listing 6-26: A listing employing `index_of_minimum` from Listing 6-25. Uncommenting ❸ causes compilation to fail.

The instantiations for `int` ❶ and `unsigned short` ❷ arrays succeed because types are `Ordered` (see Table 6-2).

However, the `Goblin` class is not `Ordered`, and template instantiation would fail if you tried to compile ❸. Crucially, the error message would be informative:

```
error: cannot call function 'size_t index_
of_minimum(auto:1*, size_t) [with auto:1 = Goblin; size_t = long unsigned int]'
   index_of_minimum(x3, 2); // Bang! Goblin is not Ordered.
                         ^
note:   constraints not satisfied
 size_t index_of_minimum(Ordered* x, size_t length) {
        ^~~~~~~~~~~~~~~~
note: within 'template<class T> concept bool origin::Ordered() [with T =
Goblin]'
 Ordered()
```

You know that the `index_of_minimum` instantiation failed and that the issue is with the `Ordered` concept.

Ad Hoc Requires Expressions

Concepts are fairly heavyweight mechanisms for enforcing type safety. Sometimes, you just want to enforce some requirement directly in the template prefix. You can embed requires expressions directly into the template definition to accomplish this. Consider the `get_copy` function in Listing 6-27 that takes a pointer and safely returns a copy of the pointed-to object.

```
#include <stdexcept>

template<typename T>
  requires❶ is_copy_constructible<T>::value ❷
T get_copy(T* pointer) {
  if (!pointer) throw std::runtime_error{ "Null-pointer dereference" };
  return *pointer;
}
```

Listing 6-27: A template function with an ad hoc requires expression

The template prefix contains the `requires` keyword ❶, which begins the requires expression. In this case, the type trait `is_copy_constructible` ensures that `T` is copyable ❷. This way, if a user accidentally tries to `get_copy` with a pointer that points to an uncopyable object, they'll be presented with a clear explanation of why template instantiation failed. Consider the example in Listing 6-28.

```
#include <stdexcept>
#include <type_traits>

template<typename T>
  requires std::is_copy_constructible<T>::value
T get_copy(T* pointer) { ❶
  --snip--
}

struct Highlander {
  Highlander() = default; ❷
  Highlander(const Highlander&) = delete; ❸
};

int main() {
  Highlander connor; ❹
  auto connor_ptr = &connor; ❺
  auto connor_copy = get_copy(connor_ptr); ❻
}
--------------------------------------------------------------------------------
In function 'int main()':
error: cannot call function 'T get_copy(T*) [with T = Highlander]'
   auto connor_copy = get_copy(connor_ptr);
                                         ^
note:   constraints not satisfied
 T get_copy(T* pointer) {
   ^~~~~~~~
note: 'std::is_copy_constructible::value' evaluated to false
```

Listing 6-28: Program using the get_copy template in Listing 6-27. This code doesn't compile.

The definition of `get_copy` ❶ is followed by a `Highlander` class definition, which contains a default constructor ❷ and a deleted copy constructor ❸. Within `main`, you've initialized a `Highlander` ❹, taken its reference ❺, and attempted to instantiate `get_copy` with the result ❻. Because there can be only one `Highlander` (it's not copyable), Listing 6-28 produces an exquisitely clear error message.

static_assert: The Preconcepts Stopgap

As of C++17, concepts aren't part of the standard, so they're not guaranteed to be available across compilers. There is a stopgap you can apply in the interim: the `static_assert` expression. These assertions evaluate at compile time. If an assertion fails, the compiler will issue an error and optionally provide a diagnostic message. A `static_assert` has the following form:

```
static_assert(boolean-expression, optional-message);
```

In the absence of concepts, you can include one or more `static_assert` expressions in the bodies of templates to assist users in diagnosing usage errors.

Suppose you want to improve the error messages of mean without leaning on concepts. You can use type traits in combination with static_assert to achieve a similar result, as demonstrated in Listing 6-29.

```
#include <type_traits>

template <typename T>
T mean(T* values, size_t length) {
  static_assert(std::is_default_constructible<T>(),
    "Type must be default constructible."); ❶
  static_assert(std::is_copy_constructible<T>(),
    "Type must be copy constructible."); ❷
  static_assert(std::is_arithmetic<T>(),
    "Type must support addition and division."); ❸
  static_assert(std::is_constructible<T, size_t>(),
    "Type must be constructible from size_t."); ❹
  --snip--
}
```

Listing 6-29: Using static_assert expressions to improve compile time errors in mean in Listing 6-10.

You see the familiar type traits for checking that T is default ❶ and copy constructible ❷, and you provide error methods to help users diagnose issues with template instantiation. You use is_arithmetic ❸, which evaluates to true if the type parameter supports arithmetic operations (+, -, /, and *), and is_constructible ❹, which determines whether you can construct a T from a size_t.

Using static_assert as a proxy for concepts is a hack, but it's widely used. Using type traits, you can limp along until concepts are included in the standard. You'll often see static_assert if you use modern third-party libraries; if you're writing code for others (including future you), consider using static_assert and type traits.

Compilers, and often programmers, don't read documentation. By baking requirements directly into the code, you can avoid stale documentation. In the absence of concepts, static_assert is a fine stopgap.

Non-Type Template Parameters

A template parameter declared with the typename (or class) keyword is called a *type template parameter*, which is a stand-in for some yet-to-be-specified type. Alternatively, you can use *non-type template parameters*, which are stand-ins for some yet-to-be-specified value. Non-type template parameters can be any of the following:

- An integral type
- An lvalue reference type
- A pointer (or pointer-to-member) type

- A std::nullptr_t (the type of nullptr)
- An enum class

Using a non-type template parameter allows you to inject a value into the generic code at compile time. For example, you can construct a template function called get that checks for out-of-bounds array access at compile time by taking the index you want to access as a non-type template parameter.

Recall from Chapter 3 that if you pass an array to a function, it decays into a pointer. You can instead pass an array reference with a particularly off-putting syntax:

```
element-type(param-name&)[array-length]
```

For example, Listing 6-30 contains a get function that makes a first attempt at performing bounds-checked array access.

```
#include <stdexcept>

int& get(int (&arr)[10]❶, size_t index❷) {
  if (index >= 10) throw std::out_of_range{ "Out of bounds" }; ❸
  return arr[index]; ❹
}
```

Listing 6-30: A function for accessing array elements with bounds checking

The get function accepts a reference to an int array of length 10 ❶ and an index to extract ❷. If index is out of bounds, it throws an out_of_bounds exception ❸; otherwise, it returns a reference to the corresponding element ❹.

You can improve Listing 6-30 in three ways, which are all enabled by non-type template parameters genericizing the values out of get.

First, you can relax the requirement that arr refer to an int array by making get a template function, as in Listing 6-31.

```
#include <stdexcept>

template <typename T❶>
T&❷ get(T❸ (&arr)[10], size_t index) {
  if (index >= 10) throw std::out_of_range{ "Out of bounds" };
  return arr[index];
}
```

Listing 6-31: A refactor of Listing 6-30 to accept an array of a generic type

As you've done throughout this chapter, you've genericized the function by replacing a concrete type (here, int) with a template parameter ❶❷❸.

Second, you can relax the requirement that arr refer to an array of length 10 by introducing a non-type template parameter Length. Listing 6-32 shows how: simply declare a size_t Length template parameter and use it in place of 10.

```
#include <stdexcept>

template <typename T, size_t Length❶>
T& get (T(&arr)[Length❷], size_t index) {
  if (index >= Length❸) throw std::out_of_range{ "Out of bounds" };
  return arr[index];
}
```

Listing 6-32: A refactor of Listing 6-31 to accept an array of a generic length

The idea is the same: rather than replacing a specific type (`int`), you've replaced a specific integral value (`10`) ❶❷❸. Now you can use the function with arrays of any size.

Third, you can perform compile time bounds checking by taking `size_t index` as another non-type template parameter. This allows you to replace the `std::out_of_range` with a `static_assert`, as in Listing 6-33.

```
#include <cstdio>

template <size_t Index❶, typename T, size_t Length>
T& get(T (&arr)[Length]) {
  static_assert(Index < Length, "Out-of-bounds access"); ❷
  return arr[Index❸];
}

int main() {
  int fib[]{ 1, 1, 2, 0 }; ❹
  printf("%d %d %d ", get<0>(fib), get<1>(fib), get<2>(fib)); ❺
  get<3>(fib) = get<1>(fib) + get<2>(fib); ❻
  printf("%d", get<3>(fib)); ❼
  //printf("%d", get<4>(fib)); ❽
}
--------------------------------------------------------------------------
1 1 2 ❺3 ❼
```

Listing 6-33: A program using compile time bounds-checked array accesses

You've moved the `size_t` index parameter into a non-type template parameter ❶ and updated the array access with the correct name `Index` ❸. Because `Index` is now a compile time constant, you also replace the `logic_error` with a `static_assert`, which prints the friendly message `Out-of-bounds access` whenever you accidentally try to access an out-of-bounds element ❷.

Listing 6-33 also contains example usage of get in `main`. You've first declared an `int` array `fib` of length 4 ❹. You then print the first three elements of the array using `get` ❺, set the fourth element ❻, and print it ❼. If you uncomment the out-of-bounds access ❽, the compiler will generate an error thanks to the `static_assert`.

Variadic Templates

Sometimes, templates must take in an unknown number of arguments. The compiler knows these arguments at template instantiation, but you want to avoid having to write many different templates each for different numbers of arguments. This is the raison d'être of variadic templates. *Variadic templates* take a variable number of arguments.

You denote variadic templates using a final template parameter that has a special syntax, namely `typename... arguments`. The ellipsis indicates that `arguments` is a *parameter pack type*, meaning you can declare parameter packs within your template. A parameter pack is a template argument that accepts zero or more function arguments. These definitions can seem a bit abstruse, so consider the following sample variadic template that builds upon `SimpleUniquePointer`.

Recall from Listing 6-14 that you pass a raw pointer into the constructor of `SimpleUniquePointer`. Listing 6-34 implements a `make_simple_unique` function that handles construction of the underlying type.

```
template <typename T, typename... Arguments❶>
SimpleUniquePointer<T> make_simple_unique(Arguments... arguments❷) {
  return SimpleUniquePointer<T>{ new T{ arguments...❸ } };
}
```

Listing 6-34: Implementing a `make_simple_unique` function to ease `SimpleUniquePointer` usage

You define the parameter pack type `Arguments` ❶, which declares `make_simple_unique` as a variadic template. This function passes arguments ❷ to the constructor of template parameter `T` ❸.

The upshot is you can now create `SimpleUniquePointers` very easily, even when the pointed-to object has a non-default constructor.

NOTE *There is a slightly more efficient implementation of Listing 6-34. If `arguments` is an rvalue, you can move it directly into the constructor of `T`. The stdlib contains a function called `std::forward` in the `<utility>` header that will detect whether `arguments` is an lvalue or rvalue and perform a copy or move, respectively. See Item 23 in* Effective Modern C++ *by Scott Meyers.*

Advanced Template Topics

For everyday polymorphic programming, templates are your go-to tool. It turns out that templates are also used in a wide range of advanced settings, especially in implementing libraries, high-performance programs, and embedded system firmware. This section outlines some of the major terrain features of this vast space.

Template Specialization

To understand advanced template usage, you must first understand template specialization. Templates can actually take more than just `concept` and `typename` parameters (type parameters). They can also accept fundamental types, like `char` (value parameters), as well as other templates. Given the tremendous flexibility of template parameters, you can make a lot of compile-time decisions about their features. You could have different versions of templates depending on the characteristics of these parameters. For example, if a type parameter is `Ordered` instead of `Regular`, you might be able to make a generic program more efficient. Programming this way is called *template specialization*. Refer to the ISO standard [temp.spec] for more information about template specialization.

Name Binding

Another critical component of how templates get instantiated is name binding. Name binding helps determine the rules for when the compiler matches a named element within a template to a concrete implementation. The named element could, for example, be part of the template definition, a local name, a global name, or from some named namespace. If you want to write heavily templated code, you need to understand how this binding occurs. If you're in such a situation, refer to Chapter 9, "Names in Templates," in *C++ Templates: The Complete Guide* by David Vandevoorde et al. and to [temp.res].

Type Function

A *type function* takes types as arguments and returns a type. The type traits with which you build up concepts are closely related to type functions. You can combine type functions with compile time control structures to do general computation, such as programming control flow, at compile time. Generally, programming using these techniques is called *template metaprogramming*.

Template Metaprogramming

Template metaprogramming has a deserved reputation for resulting in code that is exceedingly clever and absolutely inscrutable to all but the mightiest of wizards. Fortunately, once concepts are part of the C++ standard, template metaprogramming should become more approachable to us mere mortals. Until then, tread carefully. For those interested in further detail on this topic, refer to *Modern C++ Design: Generic Programming and Design Patterns Applied* by Andrei Alexandrescu and *C++ Templates: The Complete Guide* by David Vandevoorde et al.

Template Source Code Organization

Each time a template is instantiated, the compiler must be able to generate all the code necessary to use the template. This means all the information about how to instantiate a custom class or function must be available within the same translation unit as the template instantiation. By far, the most popular way to achieve this is to implement templates entirely within header files.

There are some modest inconveniences associated with this approach. Compile times can increase, because templates with the same parameters might get instantiated multiple times. It also decreases your ability to hide implementation details. Fortunately, the benefits of generic programming far outweigh these inconveniences. (Major compilers will probably minimize the problems of compile times and code duplication anyway.)

There are even a few advantages to having header-only templates:

- It's very easy for others to use your code: it's a matter of applying `#include` to some headers (rather than compiling the library, ensuring the resulting object files are visible to the linker, and so on).
- It's trivially easy for compilers to inline header-only templates, which can lead to faster code at runtime.
- Compilers can generally do a better job of optimizing code when all of the source is available.

Polymorphism at Runtime vs. Compile Time

When you want polymorphism, you should use templates. But sometimes you can't use templates because you won't know the types used with your code until runtime. Remember that template instantiation only occurs when you pair a template's parameters with types. At this point, the compiler can instantiate a custom class for you. In some situations, you might not be able to perform such pairings until your program is executing (or, at least, performing such pairing at compile time would be tedious).

In such cases, you can use runtime polymorphism. Whereas the template is the mechanism for achieving compile-time polymorphism, the runtime mechanism is the interface.

Summary

In this chapter, you explored polymorphism in C++. The chapter started with a discussion of what polymorphism is and why it's so tremendously useful. You explored how to achieve polymorphism at compile time with templates. You learned about type checking with concepts and then explored some advanced topics, such as variadic templates and template metaprogramming.

EXERCISES

6-1. The mode of a series of values is the value that appears most commonly. Implement a mode function using the following signature: `int mode(const int* values, size_t length)`. If you encounter an error condition, such as input having multiple modes and no values, return zero.

6-2. Implement `mode` as a template function.

6-3. Modify mode to accept an `Integer` concept. Verify that mode fails to instantiate with floating types like `double`.

6-4. Refactor `mean` in Listing 6-13 to accept an array rather than pointer and length arguments. Use Listing 6-33 as a guide.

6-5. Using the example from Chapter 5, make `Bank` a template class that accepts a template parameter. Use this type parameter as the type of an account rather than `long`. Verify that your code still works using a `Bank<long>` class.

6-6. Implement an `Account` class and instantiate a `Bank<Account>`. Implement functions in `Account` to keep track of balances.

6-7. Make `Account` an interface. Implement a `CheckingAccount` and `SavingsAccount`. Create a program with several checking and savings accounts. Use a `Bank<Account>` to make several transactions between the accounts.

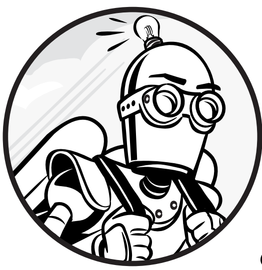
FURTHER READING

- *C++ Templates: The Complete Guide*, 2nd Edition, by David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor (Addison-Wesley, 2017)
- *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* by Scott Meyers (O'Reilly Media, 2015)
- *The C++ Programming Language*, 4th Edition, by Bjarne Stroustrup (Pearson Education, 2013)
- *Modern C++ Design: Generic Programming and Design Patterns Applied* by Andrei Alexandrescu (Addison-Wesley, 2001)

7

EXPRESSIONS

Here is the essence of mankind's creative genius: not the edifices of civilization nor the bang-flash weapons which can end it, but the words which fertilize new concepts like spermatozoa attacking an ovum.
—Dan Simmons, Hyperion



Expressions are computations that produce results and side effects. Generally, expressions contain operands and operators that do work on them. A number of operators are baked into the core language, and you'll see a majority of them in this chapter. The chapter begins with a discussion of built-in operators before moving on to discuss the overloading operator `new` and user-defined literals and then diving into an exploration of type conversions. When you create your own user-defined types, you'll often need to describe how these types convert into other types. You'll explore these user-defined conversions before learning about `constexpr` constant expressions and the widely misunderstood `volatile` keyword.

Operators

Operators, such as the addition (+) and address-of (&) operators, do work on arguments called operands, such as numerical values or objects. In this section, we'll look at logical, arithmetic, assignment, increment/decrement, comparison, member access, ternary conditional, and comma operators.

Logical Operators

The C++ expression suite includes a full complement of logical operators. Within this category are the (regular) operators AND (`&&`), OR (`||`), and NOT (`!`), which take `bool`-convertible operands and return an object of type `bool`. Also, *bitwise logical operators* work on integral types like `bool`, `int`, and `unsigned long`. These operators include AND (`&`), OR (`|`), XOR (`^`), complement (`~`), left shift (`<<`), and right shift (`>>`). Each performs a Boolean operation at the bit level and returns an integral type matching its operands.

Table 7-1 lists all of these logical operators alongside some examples.

Table 7-1: Logical Operators

<table>
<tr><th>Operator</th><th>Name</th><th>Example expression</th><th>Result</th></tr>
<tr><td>x & y</td><td>Bitwise AND</td><td>0b1100 & 0b1010</td><td>0b1000</td></tr>
<tr><td>x | y</td><td>Bitwise OR</td><td>0b1100 | 0b1010</td><td>0b1110</td></tr>
<tr><td>x ^ y</td><td>Bitwise XOR</td><td>0b1100 ^ 0b1010</td><td>0b0110</td></tr>
<tr><td>~x</td><td>Bitwise complement</td><td>~0b1010</td><td>0b0101</td></tr>
<tr><td>x << y</td><td>Bitwise left shift</td><td>0b1010 << 2
0b0011 << 4</td><td>0b101000
0b110000</td></tr>
<tr><td>x >> y</td><td>Bitwise right shift</td><td>0b1010 >> 2
0b10110011 >> 4</td><td>0b10
0b1011</td></tr>
<tr><td>x && y</td><td>AND</td><td>true && false
true && true</td><td>false
true</td></tr>
<tr><td>x || y</td><td>OR</td><td>true || false
false || false</td><td>true
false</td></tr>
<tr><td>!x</td><td>NOT</td><td>!true
!false</td><td>false
true</td></tr>
</table>

Arithmetic Operators

Additional unary and binary *arithmetic operators* work with both integral and floating-point types (also called the *arithmetic types*). You'll use built-in arithmetic operators wherever you need to perform mathematical computations. They perform some of the most basic elements of work, whether you're incrementing an index variable or performing computationally intensive statistical simulations.

Unary Arithmetic Operators

The *unary plus* + and *unary minus* - operators take a single arithmetic operand. Both operators *promote* their operands to int. So, if the operand is of type bool, char, or short int, the result of the expression is an int.

Unary plus doesn't do much besides promotion; unary minus, on the other hand, will flip the sign of the operand. For example, given char x = 10, +x results in an int with a value of 10 and -x results in an int with a value of –10.

Binary Arithmetic Operators

Aside from the two unary arithmetic operators, there are five *binary* arithmetic operators: *addition* +, *subtraction* -, *multiplication* *, *division* /, and *modulo* %. These operators take two operands and perform the indicated mathematical operation. Like their unary counterparts, these binary operators cause integer promotion on their operands. For example, adding two char operands will result in an int. There are floating-point promotion rules, too:

- If an operand is long double, the other operand is promoted to long double.
- If an operand is double, the other operand is promoted to double.
- If an operand is float, the other operand is promoted to float.

If none of the floating-point promotion rules apply, you then check whether either argument is signed. If so, both operands become signed. Finally, as with the promotion rules for floating-point types, the size of the largest operand is used to promote the other operand:

- If an operand is long long, the other operand is promoted to long long.
- If an operand is long, the other operand is promoted to long.
- If an operand is int, the other operand is promoted to int.

Although these rules are not too complicated to memorize, I recommend checking your work by leaning on auto type deduction. Just assign the result of an expression to an auto-declared variable and check the deduced type.

Don't confuse casting and promotion. Casting is when you have an object of one type and need to convert it to another type. Promotion is the set of rules for interpreting literals. For example, if you have a platform with a 2-byte short and you performed signed conversion on an unsigned short with a value of 40000, the result is an integer overflow and undefined behavior. This is entirely different from processing promotion rules on the literal 40000. If it needs to be signed, the literal's type is signed int, because a signed short is not large enough to hold such a value.

NOTE *You can use your IDE or even RTTI's* typeid *to print the type to console.*