But now to return to an earlier point—the lack of high-quality educational opportunities and materials for learning C++. High-quality C++ education is now being provided within the C++ committee itself—there's a study group dedicated just to teaching C++!—and the latter issue is in my opinion completely resolved by the very book you're holding.

Unlike all other C++ books I've read, this book teaches you the basics and the principles. It teaches you how to reason, and then lets you reason through the things that the Standard Template Library offers you. The payoff may take a bit longer, but you will be so much more satisfied to see your first results compile and run when you fully understand how C++ works. This book even includes topics that most C++ books shy away from: setting up your environment and testing your code before running the full program.

Enjoy reading this book and trying out all its exercises, and good luck on your C++ journey!

Peter Bindels
Principal Software Engineer, TomTom

ACKNOWLEDGMENTS

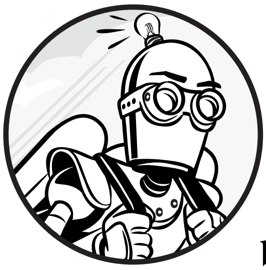
Above all, I thank my family for giving me creative space. It took twice as long to write half of what I proposed, and for your patience I owe you immeasurably.

I'm indebted to Kyle Willmon and Aaron Bray, who taught me C++; to Tyler Ortman, who shepherded this book from a proposal; to Bill Pollock, who rehabilitated my expositive style; to Chris Cleveland, Patrick De Justo, Anne Marie Walker, Annie Choi, Meg Sneeringer, and Riley Hoffman, whose top-notch editing benefited this book enormously; and to the many early readers who transmuted raw chapters into inestimable feedback.

And finally I thank Jeff Lospinoso, who bequeathed to his wide-eyed, ten-year-old nephew the well-thumbed, coffee-stained Camel Book that ignited the spark.

INTRODUCTION

Grab the ol' brush and paint along with us.
—Bob Ross



The demand for system programming is enormous. With the ubiquity of web browsers, mobile devices, and the Internet of Things, there has perhaps never been a better time to be a system programmer. Efficient, maintainable, and correct code is desired in all cases, and it's my firm belief that C++ is the right language for the job *in general*.

In the hands of a knowledgeable programmer, C++ can produce smaller, more efficient, and more readable code than any other system programming language on the planet. It's a language committed to the ideal of zero-overhead abstraction mechanisms—so your programs are fast and quick to program—as well as simple, direct mapping to hardware—so you have low-level control when you need it. When you program in C++, you stand on the shoulders of giants who have spent decades crafting an incredibly powerful and flexible language.

A huge benefit of learning C++ is that you gain access to the C++ Standard Library, the *stdlib*, free of charge. The stdlib is composed of three interlocking parts: containers, iterators, and algorithms. If you've ever written your own quicksort algorithm by hand or if you've programmed system code and been bitten by buffer overflows, dangling pointers, use-after frees, and double frees, you'll enjoy getting acquainted with the stdlib. It provides you with an unrivaled combination of type safety, correctness, and efficiency. In addition, you'll like how compact and expressive your code can be.

At the core of the C++ programming model is the *object life cycle*, which gives you strong guarantees that resources your program uses, such as files, memory, and network sockets, release correctly, even when error conditions occur. When used effectively, exceptions can clean out large amounts of error-condition-checking clutter from your code. Also, move/copy semantics provide safety, efficiency, and flexibility to manage resource ownership in a way that earlier system programming languages, like C, simply don't provide.

C++ is a living, breathing language; after more than 30 years, the International Organization for Standardization (ISO) committee for C++ regularly makes improvements in the language. Several updates to the standard have been released in the past decade: C++11, C++14, and C++17, which were released in 2011, 2014, and 2017, respectively. You can expect a new C++20 in 2020.

When I use the term *modern C++*, I mean the latest C++ version that embraces the features and paradigms presented in these additions. These updates have made serious refinements to the language that improve its expressiveness, efficiency, safety, and overall usability. By some measures, the language has never been more popular, and it's not going away any time soon. If you decide to invest in learning C++, it will pay dividends for years to come.

About This Book

Although a number of very high-quality books are available to modern C++ programmers, such as Scott Meyer's *Effective Modern C++* and Bjarne Stroustrup's *The C++ Programming Language*, 4th Edition, they're generally quite advanced. Some introductory C++ texts are available, but they often skip over crucial details because they're geared to those totally new to programming. For the experienced programmer, it's not clear where to dive into the C++ language.

I prefer to learn about complicated topics deliberately, building concepts from their fundamental elements. C++ has a daunting reputation because its fundamental elements nest so tightly together, making it difficult to construct a complete picture of the language. When I learned C++, I struggled to get my mind around the language, bouncing among books, videos, and exhausted colleagues. So I wrote the book I wish I'd had five years ago.

Who Should Read This Book?

This book is intended for intermediate to advanced programmers already familiar with basic programming concepts. If you don't specifically have *system* programming experience, that's okay. Experienced application programmers are welcome.

NOTE *If you're a seasoned C programmer or an aspiring system programmer wondering whether you should invest in learning C++, be sure to read* An Overture to C Programmers *on page xxxvii for a detailed examination.*

What's in This Book?

The book is divided into two parts. Part I covers the core C++ language. Rather than presenting the C++ language chronologically (from old-style C++ 98 to modern C++11/14/17), you'll learn idiomatic, modern C++ directly. Part II introduces you to the world of the C++ Standard Library (stdlib) where you'll learn the most important and essential concepts.

Part I: The C++ Core Language

Chapter 1: Up and Running This introductory chapter will help you set up a C++ development environment. You'll compile and run your first program, and you'll learn how to debug it.

Chapter 2: Types Here you'll explore the C++ type system. You'll learn about the fundamental types, the foundation upon which all other types are built. Next, you'll learn about plain-old-data types and fully featured classes. You'll delve into the role of constructors, initialization, and destructors.

Chapter 3: Reference Types This chapter introduces you to objects that store the memory addresses of other objects. These types are the cornerstone of many important programming patterns, and they allow you to produce flexible, efficient code.

Chapter 4: The Object Life Cycle The discussion of class invariants and the constructor is continued within the context of storage durations. The destructor is introduced alongside the resource acquisition is initialization (RAII) paradigm. You'll learn about exceptions and how they enforce class invariants and complement RAII. After a discussion of move and copy semantics, you'll explore how to operationalize them with constructors and assignment operators.

Chapter 5: Runtime Polymorphism Here you'll be introduced to interfaces, a programming concept that allows you to write code that's polymorphic at runtime. You'll learn the basics of inheritance and object composition, which underpin how you can operationalize interfaces in C++.

Chapter 6: Compile-Time Polymorphism This chapter introduces templates, a language feature that allows you to write polymorphic code. You'll also explore concepts, a language feature that will be added to a future C++ release, and named conversion functions, which allow you to convert objects from one type to another.

Chapter 7: Expressions Now you'll dive deeply into operands and operators. With a firm grasp of types, the object life cycle, and templates, you'll be ready to plunge into the core components of the C++ language, and expressions are the first waypoint.

Chapter 8: Statements This chapter explores the elements that comprise functions. You'll learn about expression statements, compound statements, declaration statements, iteration statements, and jump statements.

Chapter 9: Functions The final chapter of Part I expands on the discussion of how to arrange statements into units of work. You'll learn the details of function definitions, return types, overload resolution, variadic functions, variadic templates, and function pointers. You'll also learn how to create invokable user-defined types using the function call operator and lambda expressions. You'll explore `std::function`, a class that provides a uniform container for storing invokable objects.

Part II: C++ Libraries and Frameworks

Chapter 10: Testing This chapter introduces you to the wonderful world of unit testing and mocking frameworks. You'll practice test-driven development to develop software for an autonomous driving system while learning about frameworks, such as Boost Test, Google Test, Google Mock, and others.

Chapter 11: Smart Pointers The special utility classes that the stdlib provides for handling ownership of dynamic objects are explained.

Chapter 12: Utilities Here you'll get an overview of the types, classes, and functions at your disposal in the stdlib and Boost libraries for tackling common programming problems. You'll learn about data structures, numeric functions, and random number generators.

Chapter 13: Containers This chapter surveys the many special data structures in the Boost libraries and stdlib that help you organize data. You'll learn about sequence containers, associative containers, and unordered associative containers.

Chapter 14: Iterators This is the interface between the containers you learned about in the previous chapter and the strings of the next chapter. You'll learn about the different kinds of iterators and how their design provides you with incredible flexibility.

Chapter 15: Strings This chapter teaches you how to handle human language data in a single family of containers. You'll also learn about the special facilities built into strings that allow you to perform common tasks.

Chapter 16: Streams You'll be introduced here to the major concept underpinning input and output operations. You'll learn how to handle input and output streams with formatted and unformatted operations, as well as how to employ manipulators. You'll also learn how to read and write data from and to files.

Chapter 17: Filesystems Here you'll get an overview of the facilities in the stdlib for manipulating filesystems. You'll learn how to construct and manipulate paths, inspect files and directories, and enumerate directory structures.

Chapter 18: Algorithms This is a quick reference to the dozens of problems you can solve easily from within the stdlib. You'll learn about the impressive scope of the high-quality algorithms available to you.

Chapter 19: Concurrency and Parallelism This chapter teaches you some simple methods for multithreaded programming that are part of the stdlib. You'll learn about futures, mutexes, condition variables, and atomics.

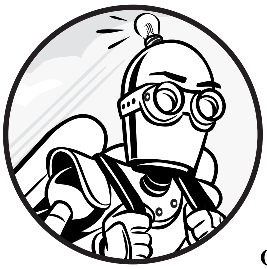
Chapter 20: Network Programming with Boost Asio Here you'll learn how to build high-performance programs that communicate over networks. You'll see how to use Boost Asio with blocking and non-blocking input and output.

Chapter 21: Writing Applications This final chapter rounds out the book with a discussion of several important topics. You'll learn about program support facilities that allow you to hook into the application life cycle. You'll also learn about Boost ProgramOptions, a library that makes writing console applications that accept user input straightforward.

NOTE *Visit the companion site* https://ccc.codes/ *to access the code listings contained in this book.*

AN OVERTURE TO C PROGRAMMERS

Arthur Dent: What's the matter with him?
Hig Hurtenflurst: His feet are the wrong size for his shoes.
—Douglas Adams, The Hitchhiker's Guide to the Galaxy, *"Fit the Eleventh"*



This preface is meant for experienced C programmers who are considering whether or not to read this book. Non–C programmers are welcome to skip this prelude.

Bjarne Stroustrup developed C++ from the C programming language. Although C++ isn't completely compatible with C, well-written C programs are often also valid C++ programs. Case in point, every example in *The C Programming Language* by Brian Kernighan and Dennis Ritchie is a legal C++ program.

One primary reason for C's ubiquity in the system-programming community is that C allows programmers to write at a higher level of abstraction than assembly programming does. This tends to produce clearer, less error-prone, and more maintainable code.

Generally, system programmers aren't willing to pay overhead for programming convenience, so C adheres to the zero-overhead principle: *what you don't use, you don't pay for*. The strong type system is a prime example of a zero-overhead abstraction. It's used only at compile time to check for program correctness. After compile time, the types will have disappeared, and the emitted assembly code will show no trace of the type system.

As a descendant of C, C++ also takes zero-overhead abstraction and direct mapping to hardware very seriously. This commitment goes beyond just the C language features that C++ supports. Everything that C++ builds on top of C, including new language features, upholds these principles, and departures from either are made very deliberately. In fact, some C++ features incur even less overhead than corresponding C code. The `constexpr` keyword is one such example. It instructs the compiler to evaluate the expression at compile time (if possible), as shown in the program in Listing 1.

```
#include <cstdio>

constexpr int isqrt(int n) {
  int i=1;
  while (i*i<n) ++i;
  return i-(i*i!=n);
}

int main() {
  constexpr int x = isqrt(1764); ❶
  printf("%d", x);
}
```

Listing 1: A program illustrating `constexpr`

The `isqrt` function computes the square root of the argument `n`. Starting at `1`, the function increments the local variable `i` until `i*i` is greater than or equal to `n`. If `i*i == n`, it returns `i`; otherwise, it returns `i-1`. Notice that the invocation of `isqrt` has a literal value, so the compiler could theoretically compute the result for you. The result will only ever take on one value ❶.

Compiling Listing 1 on GCC 8.3 targeting x86-64 with `-O2` yields the assembly in Listing 2.

```
.LC0:
        .string "%d"
main:
        sub     rsp, 8
        mov     esi, 42 ❶
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Listing 2: The assembly produced after compiling Listing 1

The salient result here is the second instruction in `main` ❶; rather than evaluating the square root of `1764` at runtime, the compiler evaluates it and outputs instructions to treat `x` as `42`. Of course, you could calculate the square root using a calculator and insert the result manually, but using `constexpr` provides lots of benefits. This approach can mitigate many errors associated with manually copying and pasting, and it makes your code more expressive.

NOTE *If you're not familiar with x86 assembly, refer to* The Art of Assembly Language, *2nd Edition, by Randall Hyde and* Professional Assembly Language *by Richard Blum.*

Upgrading to Super C

Modern C++ compilers will accommodate most of your C programming habits. This makes it easy to embrace a few of the tactical niceties that the C++ language affords you while deliberately avoiding the language's deeper themes. This style of C++—let's call it *Super C*—is important to discuss for several reasons. First, seasoned C programmers can immediately benefit from applying simple, tactical-level C++ concepts to their programs. Second, Super C is *not* idiomatic C++. Simply sprinkling references and instances of `auto` around a C program might make your code more robust and readable, but you'll need to learn other concepts to take full advantage of it. Third, in some austere environments (for example, embedded software, some operating system kernels, and heterogeneous computing), the available tool chains have incomplete C++ support. In such situations, it's possible to benefit from at least some C++ idioms, and Super C is likely to be supported. This section covers some Super C concepts you can apply to your code immediately.

NOTE *Some C-supported constructs won't work in C++. See the links section of this book's companion site,* https://ccc.codes.

Function Overloading

Consider the following conversion functions from the standard C library:

```
char* itoa(int value, char* str, int base);
char* ltoa(long value, char* buffer, int base);
char* ultoa(unsigned long value, char* buffer, int base);
```

These functions achieve the same goal: they convert an integral type to a C-style string. In C, each function must have a unique name. But in C++ functions can share names as long as their arguments differ; this is called *function overloading*. You can use function overloading to create your own conversion functions, as Listing 3 illustrates.

```
char* toa(int value, char* buffer, int base) {
  --snip--
}

char* toa(long value, char* buffer, int base)
  --snip--
}

char* toa(unsigned long value, char* buffer, int base) {
  --snip--
}
```

```
int main() {
  char buff[10];
  int a = 1; ❶
  long b = 2; ❷
  unsigned long c = 3; ❸
  toa(a, buff, 10);
  toa(b, buff, 10);
  toa(c, buff, 10);
}
```

Listing 3: Calling overloaded functions

The data type of the first argument in each of the functions differs, so the C++ compiler has enough information from the arguments passed into `toa` to call the correct function. Each `toa` call is to a unique function. Here, you create the variables `a` ❶, `b` ❷, and `c` ❸, which are different types of `int` objects that correspond with one of the three `toa` functions. This is more convenient than defining separately named functions, because you just need to remember one name and the compiler figures out which function to call.

References

Pointers are a crucial feature of C (and by extension most system programming). They enable you to handle large amounts of data efficiently by passing around data addresses instead of the actual data. Pointers are equally crucial to C++, but you have additional safety features available that defend against null dereferences and unintentional pointer reassignments.

References are a major improvement to handling pointers. They're similar to pointers, but with some key differences. Syntactically, references differ from pointers in two important ways. First, you declare them with `&` rather than `*`, as Listing 4 illustrates.

```
struct HolmesIV {
  bool is_sentient;
  int sense_of_humor_rating;
};
void mannie_service(HolmesIV*); // Takes a pointer to a HolmesIV
void mannie_service(HolmesIV&); // Takes a reference to a HolmesIV
```

Listing 4: Code illustrating how to declare functions taking pointers and references

Second, you interact with members using the dot operator `.` rather than the arrow operator `->`, as Listing 5 illustrates.

```
void make_sentient(HolmesIV* mike) {
  mike->is_sentient = true;
}

void make_sentient(HolmesIV& mike) {
  mike.is_sentient = true;
}
```

Listing 5: A program illustrating the use of the dot and arrow operators

Under the hood, references are equivalent to pointers because they're also a zero-overhead abstraction. The compiler produces similar code. To illustrate this, consider the results of compiling the `make_sentient` functions on GCC 8.3 targeting x86-64 with `-O2`. Listing 6 contains the assembly generated by compiling Listing 5.

```
make_sentient(HolmesIV*):
        mov     BYTE PTR [rdi], 1
        ret
make_sentient(HolmesIV&):
        mov     BYTE PTR [rdi], 1
        ret
```

Listing 6: The assembly generated from compiling Listing 5

However, at compile time, references provide some safety over raw pointers because, generally speaking, they cannot be null.

With pointers, you might add a `nullptr` check to be safe. For example, you might add a check to `make_sentient`, as in Listing 7.

```
void make_sentient(HolmesIV* mike) {
  if(mike == nullptr) return;
  mike->is_sentient = true;
}
```

Listing 7: A refactor of `make_sentient` from Listing 5 so it performs a `nullptr` check

Such a check is unnecessary when taking a reference; however, this doesn't mean that references are always valid. Consider the following function:

```
HolmesIV& not_dinkum() {
  HolmesIV mike;
  return mike;
}
```

The `not_dinkum` function returns a reference, which is guaranteed to be non-null. But it's pointing to garbage memory (probably in the returned-from stack frame of `not_dinkum`). You must never do this. The result will be utter misery, also known as *undefined runtime behavior:* it might crash, it might give you an error, or it might do something completely unexpected.

One other safety feature of references is that they can't be *reseated.* In other words, once a reference is initialized, it can't be changed to point to another memory address, as Listing 8 shows.

```
int main() {
  int a = 42;
  int& a_ref = a; ❶
  int b = 100;
  a_ref = b; ❷
}
```

Listing 8: A program illustrating that references cannot be reseated

You declare a_ref as a reference to int a ❶. There is no way to reseat a_ref to point to another int. You might try to reseat a with *operator=* ❷, but this actually sets the value of a to the value of b instead of setting a_ref to reference b. After the snippet is run both a and b are equal to 100, and a_ref still points to a. Listing 9 contains equivalent code using pointers instead.

```
int main() {
  int a = 42;
  int* a_ptr = &a; ❶
  int b = 100;
  *a_ptr = b; ❷
}
```

Listing 9: An equivalent program to Listing 8 using pointers

Here, you declare the pointer with a * instead of a & ❶. You assign the value of b to the memory pointed to by a_ptr ❷. With references, you don't need any decoration on the left side of the equal sign. But if you omit the * in *a_ptr, the compiler would complain that you're trying to assign an int to a pointer type.

References are just pointers with extra safety precautions and a sprinkle of syntactic sugar. When you put a reference on the left side of an equal sign, you're setting the pointed-to value equal to the right side of the equal sign.

auto Initialization

C often requires you to repeat type information more than once. In C++, you can express a variable's type information just once by utilizing the auto keyword. The compiler will know the variable's type because it knows the type of the value being used to initialize the variable. Consider the following C++ variable initializations:

```
int x = 42;
auto y = 42;
```

Here, x and y are both of int type. You might be surprised to know that the compiler can deduce the type of y, but consider that 42 is an integer literal. With auto, the compiler deduces the type on the right side of the equal sign = and sets the variable's type to the same. Because an integer literal is of int type, in this example the compiler deduces that the type of y is also an int. This doesn't seem like much of a benefit in such a simple example, but consider initializing a variable with a function's return value, as Listing 10 illustrates.

```
#include <cstdlib>

struct HolmesIV {
  --snip--
};
```

```
HolmesIV* make_mike(int sense_of_humor) {
  --snip--
}

int main() {
  auto mike = make_mike(1000);
  free(mike);
}
```

Listing 10: A toy program initializing a variable with the return value of a function

The `auto` keyword is easier to read and is more amenable to code refactoring than explicitly declaring a variable's type. If you use `auto` freely while declaring a function, there will be less work to do later if you need to change the return type of `make_mike`. The case for `auto` strengthens with more complex types, such as those involved with the template-laden code of the stdlib. The `auto` keyword makes the compiler do all the work of type deduction for you.

NOTE *You can also add* `const`*,* `volatile`*,* `&`*, and* `*` *qualifiers to* `auto`*.*

Namespaces and Implicit typedef of struct, union, and enum

C++ treats type tags as implicit `typedef` names. In C, when you want to use a `struct`, `union`, or `enum`, you have to assign a name to the type you've created using the `typedef` keyword. For example:

```
typedef struct Jabberwocks {
  void* tulgey_wood;
  int is_galumphing;
} Jabberwock;
```

In C++ land, you chortle at such code. Because the `typedef` keyword can be implicit, C++ allows you instead to declare the `Jabberwock` type like this:

```
struct Jabberwock {
  void* tulgey_wood;
  int is_galumphing;
};
```

This is more convenient and saves some typing. What happens if you also want to define a `Jabberwock` function? Well, you shouldn't, because reusing the same name for a data type and a function is likely to cause confusion. But if you're really committed to it, C++ allows you to declare a `namespace` to create different scopes for identifiers. This helps to keep user types and functions tidy, as shown in Listing 11.

```
#include <cstdio>

namespace Creature { ❶
  struct Jabberwock {
    void* tulgey_wood;
    int is_galumphing;
  };
}
namespace Func { ❷
  void Jabberwock() {
    printf("Burble!");
  }
}
```

Listing 11: Using namespaces to disambiguate functions and types with identical names

In this example, `Jabberwock` the `struct` and `Jabberwock` the function now live together in frabjous harmony. By placing each element in its own `namespace`—the `struct` in the `Creature` namespace ❶ and the function in the `Jabberwock` namespace ❷–you can disambiguate which Jabberwock you mean. You can do such disambiguation in several ways. The simplest is to qualify the name with its `namespace`, for example:

```
Creature::Jabberwock x;
Func::Jabberwock();
```

You can also employ a `using` directive to import all the names in a `namespace`, so you'd no longer need to use the fully qualified element name. Listing 12 uses the `Creature` `namespace`.

```
#include <cstdio>

namespace Creature {
  struct Jabberwock {
    void* tulgey_wood;
    int is_galumphing;
  };
}

namespace Func {
  void Jabberwock() {
    printf("Burble!");
  }
}

using namespace Creature; ❶

int main() {
  Jabberwock x; ❷
  Func::Jabberwock();
}
```

Listing 12: Employing `using namespace` to refer to a type within the `Creature` namespace

The `using namespace` ❶ enables you to omit the `namespace` qualification ❷. But you still need a qualifier on `Func::Jabberwock`, because it isn't part of the `Creature` `namespace`.

Use of a namespace is idiomatic C++ and is a zero-overhead abstraction. Just like the rest of a type's identifiers, the namespace is erased by the compiler when emitting assembly code. In large projects, it's incredibly helpful for separating code in different libraries.

Intermingling C and C++ Object Files

C and C++ code can coexist peacefully if you're careful. Sometimes, it's necessary for a C compiler to link object files emitted by a C++ compiler (and vice versa). Although this is possible, it requires a bit of work.

Two issues are related to linking the files. First, the calling conventions in the C and C++ code could potentially be mismatched. For example, the protocols for how the stack and registers are set when you call a function could be different. These calling conventions are language-level mismatches and aren't generally related to how you've written your functions. Second, C++ compilers emit different symbols than C compilers do. Sometimes the linker must identify an object by name. C++ compilers assist by decorating the object, associating a string called a *decorated name* with the object. Because of function overloads, calling conventions, and namespace usage, the compiler must encode additional information about a function beyond just its name through decoration. This is done to ensure that the linker can uniquely identify the function. Unfortunately, there is no standard for how this decoration occurs in C++ (which is why you should use the same tool chain and settings when linking between translation units). C linkers know nothing about C++ name decoration, which can cause problems if decoration isn't suppressed whenever you link against C code within C++ (and vice versa).

The fix is simple. You wrap the code you want to compile with C-style linkages using the statement extern "C", as in Listing 13.

```
// header.h
#ifdef __cplusplus
extern "C" {
#endif
void extract_arkenstone();

struct MistyMountains {
  int goblin_count;
};
#ifdef __cplusplus
}
#endif
```

Listing 13: Employing C-style linkage

This header can be shared between C and C++ code. It works because __cplusplus is a special identifier that the C++ compiler defines (but the C compiler doesn't). Accordingly, the C compiler sees the code in Listing 14 after preprocessing completes. Listing 14 illustrates the code that remains.

```
void extract_arkenstone();

struct MistyMountains {
  int goblin_count;
};
```

Listing 14: The code remaining after the preprocessor processes Listing 13 in a C environment

This is just a simple C header. The code between the `#ifdef __cplusplus` statements is removed during preprocessing, so the `extern "C"` wrapper isn't visible. For the C++ compiler, `__cplusplus` *is* defined in `header.h`, so it sees the contents of Listing 15.

```
extern "C" {
  void extract_arkenstone();

  struct MistyMountains {
    int goblin_count;
  };
}
```

Listing 15: The code remaining after the preprocessor processes Listing 13 in a C++ environment

Both `extract_arkenstone` and `MistyMountains` are now wrapped with `extern "C"`, so the compiler knows to use C linkage. Now your C source can call into compiled C++ code, and your C++ source can call into compiled C code.

C++ Themes

This section takes you on a brief tour of some core themes that make C++ the premier system-programming language. Don't worry too much about the details. The point of the following subsections is to whet your appetite.

Expressing Ideas Concisely and Reusing Code

Well-crafted C++ code has an elegant, compact quality. Consider the evolution from ANSI-C to modern C++ in the following simple operation: looping over some array `v` with `n` elements, as Listing 16 illustrates.

```
#include <cstddef>

int main() {
  const size_t n{ 100 };
  int v[n];

  // ANSI-C
  size_t i;
  for (i=0; i<n; i++) v[i] = 0; ❶
  // C99
  for (size_t i=0; i<n; i++)  v[i] = 0; ❷

  // C++17
  for (auto& x : v) x = 0; ❸
}
```

Listing 16: A program illustrating several ways to iterate over an array

This code snippet shows the different ways to declare loops in ANSI-C, C99, and C++. The index variable `i` in the ANSI-C ❶ and C99 ❷ examples are ancillary to what you're trying to accomplish, which is to access each element of `v`. The C++ version ❸ utilizes a *range-based* `for` loop, which loops over in the range of values in `v` while hiding the details of how iteration is achieved. Like a lot of the zero-overhead abstractions in C++, this construct enables you to focus on meaning rather than syntax. Range-based `for` loops work with many types, and you can even make them work with user-defined types.

Speaking of user-defined types, they allow you to express ideas directly in code. Suppose you want to design a function, `navigate_to`, that tells a hypothetical robot to navigate to some position given x and y coordinates. Consider the following prototype function:

```
void navigate_to(double x, double y);
```

What are `x` and `y`? What are their units? Your user must read the documentation (or possibly the source) to find out. Compare the following improved prototype:

```
struct Position{
--snip--
};
void navigate_to(const Position& p);
```

This function is far clearer. There is no ambiguity about what `navigate_to` accepts. As long as you have a validly constructed `Position`, you know exactly how to call `navigate_to`. Worrying about units, conversions, and so on is now the responsibility of whoever constructs the `Position` class.

You can also come close to this clarity in C99/C11 using a `const` pointer, but C++ also makes return types compact and expressive. Suppose you want to write a corollary function for the robot called `get_position` that—you guessed it—gets the position. In C, you have two options, as shown in Listing 17.

```
Position* get_position(); ❶
void get_position(Position* p); ❷
```

Listing 17: A C-style API for returning a user-defined type

In the first option, the caller is responsible for cleaning up the return value ❶, which has probably incurred a dynamic allocation (although this is unclear from the code). The caller is responsible for allocating a `Position` somewhere and passing it into get_position ❷. This latter approach is more idiomatic C-style, but the language is getting in the way: you're just trying to get a position object, but you have to worry about whether the caller or the called function is responsible for allocating and deallocating memory. C++ lets you do all of this succinctly by returning user-defined types directly from functions, as shown in Listing 18.

```
Position❶ get_position() {
  --snip--
}
void navigate() {
  auto p = get_position(); ❷
  // p is now available for use
  --snip--
}
```

Listing 18: Returning a user-defined type by value in C++

Because get_position returns a value ❶, the compiler can *elide the copy*, so it's as if you've constructed an automatic Position variable directly ❷; there's no runtime overhead. Functionally, you're in very similar territory to the C-style pass by reference of Listing 17.

The C++ Standard Library

The C++ Standard Library (stdlib) is a major reason for migrating from C. It contains high-performance, generic code that is guaranteed to be available right out of the standards-conforming box. The three broad components of the stdlib are containers, iterators, and algorithms.

Containers are the data structures. They're responsible for holding sequences of objects. They're correct, safe, and (usually) at least as efficient as what you could accomplish manually, meaning that writing your own versions of these containers would take great effort and wouldn't turn out better than the stdlib containers. Containers are neatly partitioned into two categories: *sequence containers* and *associative containers*. The sequence containers are conceptually similar to arrays; they provide accesses to sequences of elements. Associative containers contain key/value pairs, so elements in the containers can be looked up by key.

The stdlib *algorithms* are general-purpose functions for common programming tasks, such as counting, searching, sorting, and transforming. Much like containers, the stdlib algorithms are extremely high quality and broadly applicable. Users should very rarely have to implement their own version, and using the stdlib algorithms greatly increases programmer productivity, code safety, and readability.

Iterators connect containers with algorithms. For many stdlib algorithm applications, the data you want to operate on resides in a container. Containers expose iterators to provide an even, common interface, and the algorithms consume the iterators, keeping programmers (including the implementers of the stdlib) from having to implement a custom algorithm for each container type.

Listing 19 shows how to sort a container of values using a few lines of code.

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
  std::vector<int> x{ 0, 1, 8, 13, 5, 2, 3 }; ❶
  x[0] = 21; ❷
  x.push_back(1); ❸
  std::sort(x.begin(), x.end()); ❹
  std::cout << "Printing " << x.size() << " Fibonacci numbers.\n"; ❺
  for (auto number : x) {
    std::cout << number << std::endl; ❻
  }
}
```

Listing 19: Sorting a container of values using the stdlib

A good amount of computation is going on in the background, yet the code is compact and expressive. First, you initialize a `std::vector` container ❶. *Vectors* are the stdlib's dynamically sized arrays. The *initializer braces* (the `{0, 1, ...}`) set the initial values contained in `x`. You can access the elements of a `vector` just like the elements of an array using brackets (`[]`) and the index number. You use this technique to set the first element equal to `21` ❷. Because `vector` arrays are dynamically sized, you can append values to them using the `push_back` method ❸. The seemingly magical invocation of `std::sort` showcases the power of the algorithms in stdlib ❹. The methods `x.begin()` and `x.end()` return iterators that `std::sort` uses to sort `x` in place. The `sort` algorithm is decoupled from `vector` through the use of iterators.

Thanks to iterators, you can use other containers in stdlib similarly. For example, you could use a `list` (the stdlib's doubly linked list) rather than using a `vector`. Because `list` also exposes iterators through `.begin()` and `.end()` methods, you could call `sort` on the list iterators in the same way.

Additionally, Listing 19 uses iostreams. *Iostreams* are the stdlib's mechanism for performing buffered input and output. You use the put-to operator (`<<`) to stream the value of `x.size()` (the number of elements in `x`), some string literals, and the Fibonacci element `number` to `std::cout`, which encapsulates stdout ❺ ❻. The `std::endl` object is an I/O manipulator that writes `\n` and flushes the buffer, ensuring that the entire stream is written to stdout before executing the next instruction.

Now, just imagine all the hoops you'd have to jump through to write an equivalent program in C, and you'll see why the stdlib is such a valuable tool.

Lambdas

Lambdas, also called *unnamed* or *anonymous functions* in some circles, are another powerful language feature that improve the locality of code. In some cases, you should pass pointers to functions to use a pointer as the target of a newly created thread or to perform some transformation on each element of a sequence. It's generally inconvenient to define a one-time-use free function. That's where lambdas come in. A lambda is a new, custom function *defined inline with the other parameters of an invocation*. Consider the following one-liner, which computes the count of even numbers in x:

```
auto n_evens = std::count_if(x.begin(), x.end(),
                             [] (auto number) { return number % 2 == 0; });
```

This snippet uses the stdlib's `count_if` algorithm to count the even numbers in `x`. The first two arguments to `std::count_if` match `std::sort`; they're the iterators that define the range over which the algorithm will operate. The third argument is the lambda. The notation probably looks a bit foreign, but the basics are quite simple:

```
[capture] (arguments) { body }
```

Capture contains any objects you need from the scope where the lambda is defined to perform computation in the body. *Arguments* define the names and types of arguments the lambda expects to be invoked with. The *body* contains any computation that you want completed upon invocation. It might or might not return a value. The compiler will deduce the function prototype based on the types you've implied.

In the `std::count_if` invocation above, the lambda didn't need to capture any variables. All the information it needs is taken as a single argument `number`. Because the compiler knows the type of the elements contained in `x`, you declare the type of `number` with `auto` so the compiler can deduce it for you. The lambda is invoked with each element of `x` passed in as the `number` parameter. In the body, the lambda returns `true` only when `number` is divisible by `2`, so only the even numbers are included in the count.

Lambdas don't exist in C, and it's not really possible to reconstruct them. You'd need to declare a separate function each time you need a function object, and it's not possible to capture objects into a function in the same way.

Generic Programming with Templates

Generic programming is writing code once that works with different types rather than having to repeat the same code multiple times by copying and pasting each type you want to support. In C++, you use *templates* to produce generic code. Templates are a special kind of parameter that tells the compiler to represent a wide range of possible types.

You've already used templates: all of the stdlib's containers use templates. For the most part, the type of the objects in these containers doesn't matter. For example, the logic for determining the number of elements in a container or returning its first element doesn't depend on the element's type.

Suppose you want to write a function that adds three numbers of the same type. You want to accept any addable type. In C++, this is a straightforward generic programming problem that you can solve directly with templates, as Listing 20 illustrates.

```
template <typename T>
T add(T x, T y, T z) { ❶
  return x + y + z;
}

int main() {
  auto a = add(1, 2, 3);       // a is an int
  auto b = add(1L, 2L, 3L);    // b is a long
  auto c = add(1.F, 2.F, 3.F); // c is a float
}
```

Listing 20: Using templates to create a generic `add` *function*

When you declare `add` ❶, you don't need to know `T`. You only need to know that all the arguments and the return value are of type `T` and that `T` is addable. When the compiler encounters `add` being called, it deduces `T` and generates a bespoke function on your behalf. That's some serious code reuse!

Class Invariants and Resource Management

Perhaps the single greatest innovation C++ brings to system programming is the *object life cycle*. This concept has its roots in C, where objects have different storage durations depending on how you declare them in your code.

C++ builds on top of this memory management model with constructors and destructors. These special functions are methods that belong to *user-defined types*. User-defined types are the basic building blocks of C++ applications. Think of them as `struct` objects that can also have functions.

An object's constructor is called just after its storage duration begins, and the destructor is called just before its storage duration ends. Both the constructor and destructor are functions with no return type and the same name as the enclosing class. To declare a destructor, add a ~ to the beginning of the class name, as Listing 21 illustrates.

```
#include <cstdio>

struct Hal {
  Hal() : version{ 9000 } { // Constructor ❶
    printf("I'm completely operational.\n");
  }
  ~Hal() { // Destructor ❷
    printf("Stop, Dave.\n");
  }
  const int version;
};
```

Listing 21: A `Hal` *class containing a constructor and a destructor*

The first method in `Hal` is the *constructor* ❶. It sets up the `Hal` object and establishes its *class invariants*. Invariants are features of a class that don't change once they've been constructed. With some help from the compiler and the runtime, the programmer decides what the invariants of a class are and ensures that their code enforces them. In this case, the constructor sets the version, which is an invariant, to 9000. The *destructor* is the second method ❷. Whenever Hal is about to be deallocated, it prints "Stop, Dave." to the console. (Getting Hal to sing "Daisy Bell" is left as an exercise to the reader.)

The compiler makes sure the constructor and destructor are invoked automatically for objects with static, local, and thread local storage duration. For objects with dynamic storage duration, you use the keywords new and delete to replace malloc and free, Listing 22 illustrates.

```
#include <cstdio>

struct Hal {
--snip--
};

int main() {
  auto hal = new Hal{};  // Memory is allocated, then constructor is called
  delete hal;            // Destructor is called, then memory is deallocated
}
--------------------------------------------------------------------------------
I'm completely operational.
Stop, Dave.
```

Listing 22: A program that creates and destroys a Hal object

If (for whatever reason) the constructor is unable to achieve a good state, it typically throws an *exception*. As a C programmer, you might have dealt with exceptions when programming with some operating system APIs (for example, Windows Structured Exception Handling). When an exception is thrown, the stack unwinds until an exception handler is found, at which point the program recovers. Judicious use of exceptions can clean up code because you only have to check for error conditions where it makes sense to do so. C++ has language-level support for exceptions, as Listing 23 illustrates.

```
#include <exception>

try {
  // Some code that might throw a std::exception ❶
} catch (const std::exception &e) {
  // Recover the program here. ❷
}
```

Listing 23: A try-catch block

You can put your code that might throw an exception in the block immediately following try ❶. If at any point an exception is thrown, the stack will unwind (graciously destructing any objects that go out of scope) and run any code that you've put after the catch expression ❷. If no exception is thrown, this catch code never executes.

Constructors, destructors, and exceptions are closely related to another core C++ theme, which is tying an object's life cycle to the resources it owns.

This is the resource allocation is initialization (RAII) concept (sometimes also called *constructor acquires, destructor releases*). Consider the C++ class in Listing 24.

```
#include <system_error>
#include <cstdio>

struct File {
  File(const char* path, bool write) { ❶
    auto file_mode = write ? "w" : "r"; ❷
    file_pointer = fopen(path, file_mode); ❸
    if (!file_pointer) throw std::system_error(errno, std::system_category()); ❹
  }
  ~File() {
    fclose(file_pointer);
  }
  FILE* file_pointer;
};
```

Listing 24: A `File` class

The constructor of `File` ❶ takes two arguments. The first argument corresponds with the `path` of the file, and the second is a `bool` corresponding to whether the file mode should be open for write (`true`) or read (`false`). This argument's value sets `file_mode` ❷ via the *ternary operator* `? :`. The ternary operator evaluates a Boolean expression and returns one of two values depending on the Boolean value. For example:

```
x ? val_if_true : val_if_false
```

If the Boolean expression `x` is `true`, the expression's value is `val_if_true`. If `x` is `false`, the value is `val_if_false` instead.

In the `File` constructor code snippet in Listing 24, the constructor attempts to open the file at `path` with read/write access ❸. If anything goes wrong, the call will set `file_pointer` to `nullptr`, a special C++ value that's similar to 0. When this happens, you throw a `system_error` ❹. A `system_error` is just an object that encapsulates the details of a system error. If `file_pointer` isn't `nullptr`, it's valid to use. That's this class's invariant.

Now consider the program in Listing 25, which employs `File`.

```
#include <cstdio>
#include <system_error>
#include <cstring>

struct File {
--snip–
};

int main() {
  { ❶
    File file("last_message.txt", true); ❷
    const auto message = "We apologize for the inconvenience.";
    fwrite(message, strlen(message), 1, file.file_pointer);
  } ❸
  // last_message.txt is closed here!
  {
    File file("last_message.txt", false); ❹
    char read_message[37]{};
    fread(read_message, sizeof(read_message), 1, file.file_pointer);
    printf("Read last message: %s\n", read_message);
  }
}
```

```
-------------------------------------------------------------------------------
We apologize for the inconvenience.
```

Listing 25: A program employing the `File` class

The braces ❶ ❸ define a scope. Because the first `file` resides within this scope, the scope defines the lifetime of `file`. Once the constructor returns ❷, you know that `file.file_pointer` is valid thanks to the class invariant; based on the design of the constructor of `File`, you know `file.file_pointer` must be valid for the lifetime of the `File` object. You write a message using `fwrite`. There's no need to call `fclose` explicitly, because `file` expires and the destructor cleans up `file.file_pointer` for you ❷. You open `File` again but this time for read access ❹. As long as the constructor returns, you know that *last_message .txt* was opened successfully and continue on reading into `read_message`. After printing the message, the destructor of `file` is called, and the `file.file_pointer` is again cleaned up.

Sometimes you need the flexibility of dynamic memory allocation, but you still want to lean on the object life cycle of C++ to ensure that you don't leak memory or accidentally "use after free." This is exactly the role of *smart pointers*, which manage the life cycle of dynamic objects through an ownership model. Once no smart pointer owns a dynamic object, the object destructs.

One such smart pointer is `unique_ptr`, which models exclusive ownership. Listing 26 illustrates its basic usage.

```
#include <memory>

struct Foundation{
  const char* founder;
};

int main() {
  std::unique_ptr<Foundation> second_foundation{ new Foundation{} }; ❶
  // Access founder member variable just like a pointer:
  second_foundation->founder = "Wanda";
} ❷
```

Listing 26: A program employing a `unique_ptr`

You dynamically allocate a `Foundation`, and the resulting `Foundation*` pointer is passed into the constructor of `second_foundation` using the braced-initialization syntax ❶. The second_foundation has type unique_ptr, which is just an RAII object wrapping the dynamic Foundation. When second_foundation is destructed ❷, the dynamic Foundation destructs appropriately.

Smart pointers differ from regular, *raw* pointers because a raw pointer is simply a memory address. You must orchestrate all the memory management that's involved with the address manually. On the other hand, smart pointers handle all these messy details. By wrapping a dynamic object with a smart pointer, you can rest assured that memory will be cleaned up appropriately as soon as the object is no longer needed. The compiler knows that the object is no longer needed because the smart pointer's destructor is called when it falls out of scope.

Move Semantics

Sometimes, you want to transfer ownership of an object; this comes up often, for example, with unique_ptr. You can't copy a unique_ptr, because once one of the copies of the unique_ptr is destructed, the remaining unique_ptr would hold a reference to the deleted object. Rather than copying the object, you use the move semantics of C++ to transfer ownership from one unique pointer to another, as Listing 27 illustrates.

```
#include <memory>

struct Foundation{
  const char* founder;
};

struct Mutant {
  // Constructor sets foundation appropriately:
  Mutant(std::unique_ptr<Foundation> foundation)
    : foundation(std::move(foundation)) {}
  std::unique_ptr<Foundation> foundation;
};

int main() {
  std::unique_ptr<Foundation> second_foundation{ new Foundation{} }; ❶
  // ... use second_foundation
  Mutant the_mule{ std::move(second_foundation) }; ❷
  // second_foundation is in a 'moved-from' state
  // the_mule owns the Foundation
}
```

Listing 27: A program moving a unique_ptr

As before, you create unique_ptr<Foundation> ❶. You use it for some time and then decide to transfer ownership to a Mutant object. The move function tells the compiler that you want to make the transfer. After constructing the_mule ❷, the lifetime of Foundation is tied to the lifetime of the_mule through its member variable.

Relax and Enjoy Your Shoes

C++ is *the* premier system programming language. Much of your C knowledge will map directly into C++, but you'll also learn many new concepts. You can start gradually incorporating C++ into your C programs using Super C. As you become competent in some of the deeper themes of C++, you'll find that writing modern C++ brings with it many substantial advantages over C. You'll be able to express ideas concisely in code, capitalize on the impressive stdlib to work at a higher level of abstraction, employ templates to improve runtime performance and code reuse, and lean on the C++ object life cycle to manage resources.

I expect that the investment you'll make learning C++ will yield vast dividends. After reading this book, I think you'll agree.

PART I

THE C++ CORE LANGUAGE

First we crawl. Later we crawl on broken glass.
—*Scott Meyers,* Effective STL

Part I teaches you the crucial concepts in the C++ Core Language. Chapter 1 sets up a working environment and bootstraps some language constructs, including the basics of objects, the primary abstraction you use to program C++.

The next five chapters examine objects and types—the heart and soul of C++. Unlike some other programming books, you won't be building web servers or launching rocket ships from the get-go. All the programs in Part I simply print to the command line. The focus is on building your mental model of the language instead of instant gratification.

Chapter 2 takes an extensive look at types, the language construct that defines your objects.

Chapter 3 extends the discussion of Chapter 2 to discuss reference types, which describe objects that refer to other objects.

Chapter 4 describes the object life cycle, one of the most powerful aspects of C++.

Chapters 5 and 6 explore compile-time polymorphism with templates and runtime polymorphism with interfaces, which allow you to write loosely coupled and highly reusable code.

Armed with a foundation in C++'s object model, you'll be ready to dive into Chapters 7 through 9. These chapters present expressions, statements, and functions, which you use to get work done in the language. It might seem odd that these language constructs appear at the end of Part I, but without a strong knowledge of objects and their life cycles, all but the most basic features of these language constructs would be impossible to understand.

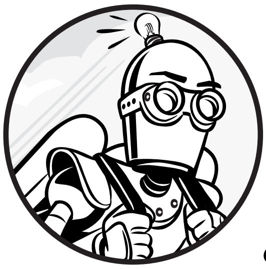
As a comprehensive, ambitious, powerful language, C++ can overwhelm the newcomer. To make it approachable, Part I is sequential, cohesive, and meant to be read like a story.

Part I is an entry fee. All your hard work learning the C++ Core Language buys you admission into the all-you-can-eat buffet of libraries and frameworks in Part II.

1

UP AND RUNNING

. . . with such violence I fell to the ground that I found myself stunned, and in a hole nine fathoms under the grass. . . . Looking down, I observed that I had on a pair of boots with exceptionally sturdy straps. Grasping them firmly, I pulled (repeatedly) with all my might.
—*Rudolph Raspe,* The Singular Adventures of Baron Munchausen



In this chapter, you'll begin by setting up a C++ *development environment*, which is the collection of tools that enables you to develop C++ software. You'll use the development environment to compile your first C++ *console application*, a program that you can run from the command line. Then you'll learn the main components of the development environment along with the role they play in generating the application you'll write. The chapters that follow will cover enough C++ essentials to construct useful example programs.

C++ has a reputation for being hard to learn. It's true that C++ is a big, complex, and ambitious language and that even veteran C++ programmers regularly learn new patterns, features, and usages.

A major source of nuance is that C++ features mesh together so tightly. Unfortunately, this often causes some distress to newcomers. Because C++ concepts are so tightly coupled, it's just not clear where to jump in. Part I of this book charts a deliberate, methodical course through the tumult, but it has to begin somewhere. This chapter covers just enough to get you started. Don't sweat the details too much!

The Structure of a Basic C++ Program

In this section, you'll write a simple C++ program and then compile and run it. You write C++ source code into human-readable text files called *source files*. Then you use a compiler to convert your C++ into executable machine code, which is a program that computers can run.

Let's dive in and create your first C++ source file.

Creating Your First C++ Source File

Open your favorite text editor. If you don't have a favorite just yet, try Vim, Emacs, or gedit on Linux; TextEdit on Mac; or Notepad on Windows. Enter the code in Listing 1-1 and save the resulting file to your desktop as *main.cpp*.

```
#include <cstdio> ❶

int main❷(){
  printf("Hello, world!"); ❸
  return 0; ❹
}
--------------------------------------------------------------------------------
Hello, world! ❸
```

Listing 1-1: Your first C++ program prints `Hello, world!` *to the screen.*

The Listing 1-1 source file compiles to a program that prints the characters `Hello, world!` to the screen. By convention, C++ source files have a *.cpp* extension.

NOTE *In this book, listings will include any program output immediately after the program's source; the output will appear in gray. Numerical annotations will correspond with the line that produced the output. The* `printf` *statement in Listing 1-1, for example, is responsible for the output* `Hello, world!`*, so these share the same annotation* ❸.

Main: A C++ Program's Starting Point

As shown in Listing 1-1, C++ programs have a single entry point called the `main` function ❷. An *entry point* is a function that executes when a user runs a program. *Functions* are blocks of code that can take inputs, execute some instructions, and return results.

Within `main`, you call the function `printf`, which prints the characters `Hello, world!` to the console ❸. Then the program exits by returning the exit code `0` to the operating system ❹. *Exit codes* are integer values that the operating system uses to determine how well a program ran. Generally, a zero (`0`) exit code means the program ran successfully. Other exit codes might indicate a problem. Having a return statement in `main` is optional; the exit code defaults to `0`.

The `printf` function is not defined in the program; it's in the `cstdio` library ❶.

Libraries: Pulling in External Code

Libraries are helpful code collections you can import into your programs to prevent having to reinvent the wheel. Virtually every programming language has some way of incorporating library functionality into a program:

- Python, Go, and Java have `import`.
- Rust, PHP, and C# have `use/using`.
- JavaScript, Lua, R, and Perl have `require/requires`.
- C and C++ have `#include`.

Listing 1-1 included `cstdio` ❶, a library that performs input/output operations, such as printing to the console.

The Compiler Tool Chain

After writing the source code for a C++ program, the next step is to turn your source code into an executable program. The *compiler tool chain* (or *tool chain*) is a collection of three elements that run one after the other to convert source code into a program:

1. The **preprocessor** performs basic source code manipulation. For example, `#include <cstdio>` ❶ is a directive that instructs the preprocessor to include information about the `cstdio` library directly into your program's source code. When the preprocessor finishes processing a source file, it produces a single translation unit. Each translation unit is then passed to the compiler for further processing.
2. The **compiler** reads a translation unit and generates an *object file*. Object files contain an intermediate format called object code. These files contain data and instructions in an intermediate format that most humans wouldn't understand. Compilers work on one translation unit at a time, so each translation unit corresponds to a single object file.
3. The **linker** generates programs from object files. Linkers are also responsible for finding the libraries you've included within your source code. When you compile Listing 1-1, for example, the linker will find the `cstdio` library and include everything your program needs to use the `printf` function. Note that the `cstdio` header is distinct from the `cstdio` library. The header contains information about how to use the library. You'll learn more about libraries and source code organization in Chapter 21.

Setting Up Your Development Environment

All C++ development environments contain a way to edit source code and a compiler tool chain to turn that source code into a program. Often, development environments also contain a *debugger*—an invaluable program that lets you step through a program line by line to find errors.

When all of these tools—the text editor, the compiler tool chain, and the debugger—are bundled into a single program, that program is called an *interactive development environment (IDE)*. For beginners and veterans alike, IDEs can be a huge productivity booster.

NOTE *Unfortunately, C++ doesn't have an interpreter with which to interactively execute C++ code snippets. This is different from other languages like Python, Ruby, and JavaScript, which do have interpreters. Some web applications exist that allow you to test and share small C++ code snippets. See Wandbox (*https://wandbox.org/*), which allows you to compile and run code, and Matt Godbolt's Compiler Explorer (*https://www.godbolt.org/*), which allows you to inspect the assembly code that your code generates. Both work on a variety of compilers and systems.*

Each operating system has its own source code editors and compiler tool chain, so this section is broken out by operating system. Skip to the one that is relevant to you.

Windows 10 and Later: Visual Studio

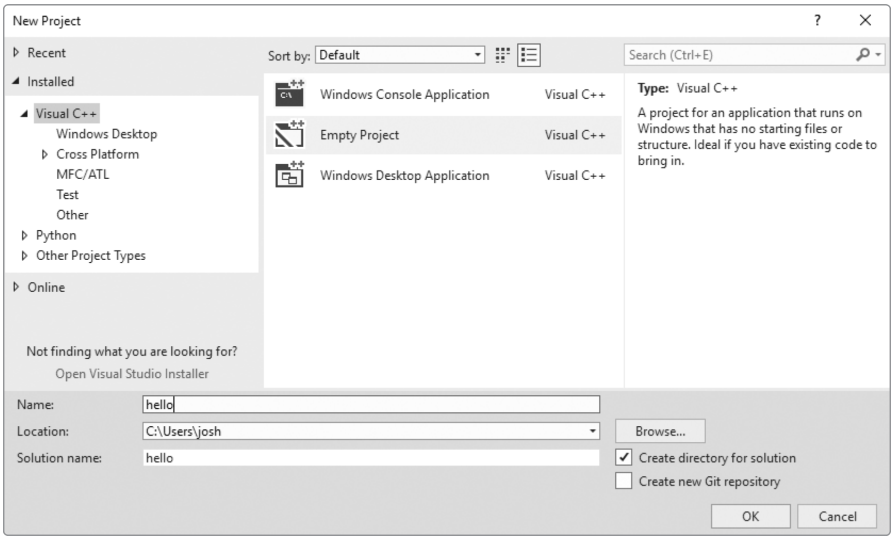
At press time, the most popular C++ compiler for Microsoft Windows is the Microsoft Visual C++ Compiler (MSVC). The easiest way to obtain MSVC is to install the Visual Studio 2017 IDE as follows:

1. Download the Community version of Visual Studio 2017. A link is available at *https://ccc.codes/*.
2. Run the installer, allowing it to update if required.
3. At the Installing Visual Studio screen, ensure that **Desktop Development with C++ Workload** is selected.
4. Click **Install** to install Visual Studio 2017 along with MSVC.
5. Click **Launch** to launch Visual Studio 2017. The entire process might take several hours depending on the speed of your machine and your selections. Typical installations require 20GB to 50GB.

Set up a new project:

1. Select **File ▸ New ▸ Project**.
2. In **Installed**, click **Visual C++** and select **General**. Select **Empty Project** in the center panel.

3. Enter `hello` as the name of your project. Your window should look like Figure 1-1, but the Location will vary depending on your username. Click **OK**.



Type: Visual C++

A project for an application that runs on Windows that has no starting files or structure. Ideal if you have existing code to bring in.

Name: hello

Location: C:\Users\josh

Solution name: hello

[✓] Create directory for solution

[ ] Create new Git repository

Figure 1-1: The Visual Studio 2017 New Project wizard

4. In the **Solution Explorer** pane on the left side of the workspace, right-click the **Source Files** folder and select **Add ▸ Existing Item**. See Figure 1-2.

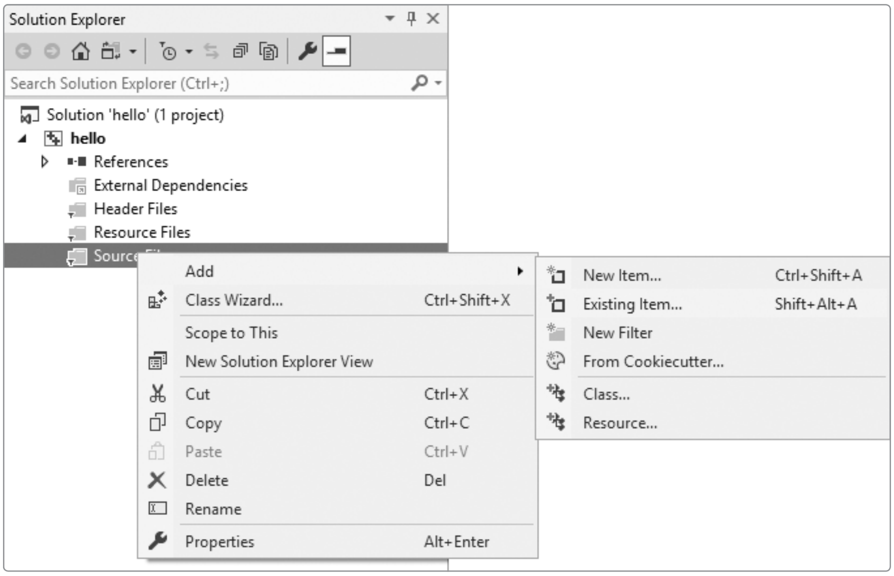


Figure 1-2: Adding an existing source file to a Visual Studio 2017 project