1
A Brief Introduction to C++

This book aims to provide you with a solid foundation to write efficient applications, as well as an insight into strategies for implementing libraries in modern C++. I have tried to take a practical approach to explain how C++ works today, where modern features from C++11 up to C++20 are a natural part of the language, rather than looking at C++ historically.

In this chapter, we will:

- Cover some of the features of C++ that are important for writing robust, high-performance applications
- Discuss the advantages and disadvantages of C++ over competing languages
- Go over the libraries and compilers used in this book

Why C++?

Let's begin by exploring some of the reasons for using C++ today. In short, C++ is a highly portable language that offers zero-cost abstractions. Furthermore, C++ provides programmers with the ability to write and manage large, expressive, and robust code bases. In this section, we'll look at what we mean by *zero-cost abstractions*, compare C++ abstraction with abstraction in other languages, and discuss portability and robustness, and why such features are important.

Let's begin by getting into zero-cost abstractions.

Zero-cost abstractions

Active code bases grow. The more developers working on a code base, the larger the code base becomes. In order to manage the growing complexity of a code base, we need language features such as variables, functions, and classes to be able to create our own abstractions with custom names and interfaces that suppress details of the implementation.

C++ allows us to define our own abstractions but it also comes with built-in abstractions. The concept of a C++ function, for example, is in itself an abstraction for controlling program flow. The range-based `for` -loop is another example of a built-in abstraction that makes it possible to iterate over a range of values more directly. As programmers, we add new abstraction continuously

while developing programs. Similarly, new versions of C++ introduce new abstractions to the language and the standard library. But constantly adding abstractions and new levels of indirection comes at a price – efficiency. This is where zero-cost abstractions play its role. A lot of the abstractions offered by C++ come at a very low runtime cost with respect to space and time.

With C++, you are free to talk about memory addresses and other computer-related low-level terms when needed. However, in a large-scale software project, it is desirable to express code in terms that deal with whatever the application is doing, and let the libraries handle the computer-related terminology. The source code of a graphics application may deal with pencils, colors, and filters, whereas a game may deal with mascots, castles, and mushrooms. Low-level computer-related terms, such as memory addresses, can stay hidden in C++ library code where performance is critical.

Programming languages and machine code abstractions

In order to relieve programmers from the need to deal with computer-related terms, modern programming languages use abstractions so that a list of strings, for example, can be handled and thought of as a list of strings rather than a list of addresses that we may easily lose track of if we make the slightest typo. Not only do the abstractions relieve the programmers from bugs, they

also make the code more expressive by using concepts from the domain of the application. In other words, the code is expressed in terms that are closer to a spoken language than if expressed with abstract programming keywords.

C++ and C are two completely different languages nowadays. Still, C++ is highly compatible with C and has inherited a lot of its syntax and idioms from C. To give you some examples of C++ abstractions, I will show how a problem can be solved in both C and C++.

Take a look at the following C/C++ code snippets, which correspond to the question: "How many copies of Hamlet are in this list of books?"

We will begin with the C version:

```
// C version
struct string_elem_t { const char* str_; string_elem_t* next_; };
int num_hamlet(string_elem_t* books) {
  const char* hamlet = "Hamlet";
  int n = 0;
  string_elem_t* b;
  for (b = books; b != 0; b = b->next_)
    if (strcmp(b->str_, hamlet) == 0)
      ++n;
```

```
  return n;
}
```

The equivalent version using C++ would look something like this:

```
// C++ version
int num_hamlet(const std::forward_list<std::string>& books) {
  return std::count(books.begin(), books.end(), "Hamlet");
}
```

Although the C++ version is still more of a robot language than a human language, a lot of programming lingo is gone thanks to the higher levels of abstraction. Here are some of the noticeable differences between the preceding two code snippets:

- The pointers to raw memory addresses are not visible at all
- The `std::forward_list<std::string>` container replaces the hand crafted linked list using `string_elem_t`
- The `std::count()` function replaces both the `for` -loop and the `if` -statement
- The `std::string` class provides a higher-level abstraction over `char*` and `strcmp()`

Basically, both versions of `num_hamlet()` translate to roughly the same machine code, but the language features of C++ makes it possible to let the libraries hide computer-related terminology

such as pointers. Many of the modern C++ language features can be seen as abstractions on top of basic C functionality.

Abstractions in other languages

Most programming languages are based on abstractions, which are transformed into machine code to be executed by the CPU. C++ has evolved into a highly expressive language, just like many of the other popular programming languages of today. What distinguishes C++ from most other languages is that while the other languages have implemented these abstractions at the cost of runtime performance, C++ has always strived to implement its abstractions at zero cost at runtime. This doesn't mean that an application written in C++ is by default faster than the equivalent in, say, C#. Rather, it means that by using C++, you'll have fine-grained control of the emitted machine code instructions and memory footprint if needed.

To be fair, optimal performance is very rarely required today, and compromising performance for lower compilation times, garbage collection, or safety, like other languages do, is in many cases more reasonable.

The zero-overhead principle

"Zero-cost abstractions" is a commonly used term, but it is afflicted with a problem – most abstractions usually do cost. If not

while running the program, it almost always cost somewhere down the line, such as long compilation times, compilation error messages that are hard to interpret and so forth. What is usually more interesting to talk about is the zero-overhead principle. Bjarne Stroustrup, the inventor of C++, defines the zero-overhead principle like this:

- What you don't use, you don't pay for
- What you do use, you couldn't hand code any better

This a core principle in C++ and a very important aspect of the evolution of the language. Why, you may ask? Abstractions built on this principle will be accepted and used broadly by performance-aware programmers and in a context where performance is highly critical. Finding abstractions that many people agree on and use extensively, makes our code bases easier to read and maintain.

On the contrary, features in the C++ language that don't fully follow the zero-overhead principle tend to be abandoned by programmers, projects, and companies. Two of the most notable features in this category are **exceptions** (unfortunately) and **Run-time Type Information** (**RTTI**). Both these features can have an impact on the performance even when they are not being used. I strongly recommend using exceptions though, unless you have a very good reason not to. The performance overhead is in most

cases negligible compared to using some other mechanism for handling errors.

Portability

C++ has been a popular and comprehensive language for a long time. It's highly compatible with C, and very little has been deprecated in the language, for better or worse. The history and design of C++ has made it into a highly portable language, and the evolution of modern C++ has ensured that it will stay that way for a long time to come. C++ is a living language, and compiler vendors are currently doing a remarkable job to implement new language features rapidly.

Robustness

In addition to performance, expressiveness, and portability, C++ offers a set of language features that gives the programmer the ability to write robust code.

In the experience of the authors, robustness does not refer to strength in the programming language itself – it's possible to write robust code in any language. Rather, strict ownership of resources, `const` correctness, value semantics, type safety, and the deterministic destruction of objects are some of the features offered by C++ that makes it easier to write robust code. That is, the

ability to write functions, classes, and libraries that are easy to use and hard to misuse.

C++ of today

To sum it up, the C++ of today provides programmers with the ability to write an expressive and robust code base while still having the option to target almost any hardware platform or real-time requirements. Among the most commonly used languages today, C++ alone possesses all of these properties.

I've now provided a brief rundown as to why C++ remains a relevant and widely used programming language today. In the next section, we'll look at how C++ compares to other modern programming languages.

C++ compared with other languages

A multitude of application types, platforms, and programming languages have emerged since C++ was first released. Still, C++ remains a widely used language, and its compilers are available for most platforms. The major exception, as of today, is the web platform, where JavaScript and its related technologies are the foundation. However, the web platform is evolving into being

able to execute what was previously only possible in desktop applications, and in that context, C++ has found its way into web applications using technologies such as Emscripten, asm.js, and WebAssembly.

In this section, we'll begin by looking at competing languages in the context of performance. Following this, we'll look at how C++ handles object ownership and garbage collection in comparison to other languages, and how we can avoid null objects in C++. Finally, we'll cover some drawbacks of C++ that users should keep in mind when considering whether the language is appropriate for their requirements.

Competing languages and performance

In order to understand how C++ achieves its performance compared to other programming languages, let's discuss some fundamental differences between C++ and most other modern programming languages.

For simplicity, this section will focus on comparing C++ to Java, although the comparisons for most parts also apply to other programming language based upon a garbage collector, such as C# and JavaScript.

Firstly, Java compiles to bytecode, which is then compiled to machine code while the application is executing, whereas the major-

ity of C++ implementations directly compiles the source code to machine code. Although bytecode and just-in-time compilers may theoretically be able to achieve the same (or, theoretically, even better) performance than precompiled machine code, as of today, they usually do not. To be fair, though, they perform well enough for most cases.

Secondly, Java handles dynamic memory in a completely different manner from C++. In Java, memory is automatically deallocated by a garbage collector, whereas a C++ program handles memory deallocations manually or by a reference counting mechanism. The garbage collector does prevent memory leaks, but at the cost of performance and predictability.

Thirdly, Java places all its objects in separate heap allocations, whereas C++ allows the programmer to place objects both on the stack and on the heap. In C++, it's also possible to create multiple objects in one single heap allocation. This can be a huge performance gain for two reasons: objects can be created without always allocating dynamic memory, and multiple related objects can be placed adjacent to one another in memory.

Take a look at how memory is allocated in the following example. The C++ function uses the stack for both objects and integers; Java places the objects on the heap:

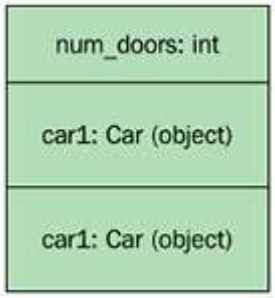
C++

```cpp
class Car {
public:
  Car(int doors)
      : doors_(doors) {}
private:
  int doors_{};
};
auto some_func() {
  auto num_doors = 2;
  auto car1 = Car{num_doors};
  auto car2 = Car{num_doors};
  // ...
}
```

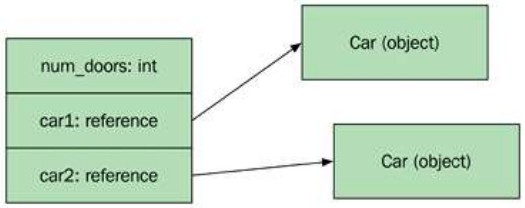
Java

```java
class Car {
  public Car(int doors) {
    doors_ = doors;
  }
  private int doors_;
  static void some_func() {
    int numDoors = 2;
    Car car1 = new Car(numDoors);
    Car car2 = new Car(numDoors);
    // ...
  }
}
```

C++ places everything on the stack:



Java places the `Car` objects on the heap:



Now take a look at the next example and see how an array of `Car` objects are placed in memory when using C++ and Java, respectively:

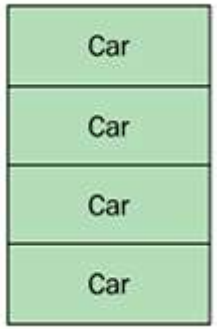
C++

Java

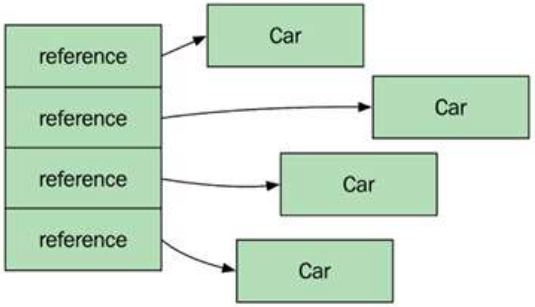
```cpp
auto n = 4;
auto cars = std::vector<Car>{};
cars.reserve(n);
for (auto i=0; i<n;++i) {
  cars.push_back(Car{2});
}
```

```java
int n = 4;
ArrayList<Car> cars =
 new ArrayList<Car>();
for (int i=0; i<n; i++) {
 cars.addElement(new Car(2));
}
```

The following diagram shows how the `Car` objects are laid out in memory in C++:



The following diagram shows how the `Car` objects are laid out in memory in Java:



The C++ vector contains the actual `Car` objects placed in one contiguous memory block, whereas the equivalent in Java is a

contiguous memory block of *references* to `Car` objects. In Java, the objects have been allocated separately, which means that they can be located anywhere on the heap.

This affects the performance, as Java, in this example, effectively has to execute five allocations in the Java heap space. It also means that whenever the application iterates the list, there is a performance win for C++, since accessing nearby memory locations is faster than accessing several random spots in memory.

Non-performance-related C++ language features

It's tempting to believe that C++ should only be used if performance is a major concern. Isn't it the case that C++ just increases the complexity of the code base due to manual memory handling, which may result in memory leaks and hard-to-track bugs?

This may have been true several C++ versions ago, but a modern C++ programmer relies on the provided containers and smart pointer types, which are part of the standard library. A substantial part of the C++ features added over the last 10 years has made the language both more powerful and simpler to use.

I would like to highlight some old but powerful features of C++ here that relate to robustness rather than performance, which are

easily overlooked: value semantics, `const` correctness, ownership, deterministic destruction, and references.

Value semantics

C++ supports both value semantics and reference semantics. Value semantics lets us pass objects by value instead of just passing references to objects. In C++, value semantics is the default, which means that when you pass an instance of a class or struct, it behaves in the same way as passing an `int`, `float`, or any other fundamental type. To use reference semantics, we need to explicitly use references or pointers.

The C++ type system gives us the ability to explicitly state the ownership of an object. Compare the following implementations of a simple class in C++ and Java. We will start with the C++ version:

```
// C++
class Bagel {
public:
  Bagel(std::set<std::string> ts) : toppings_(std::move(ts)) {}
private:
  std::set<std::string> toppings_;
};
```

The corresponding implementation in Java could look like this:

```java
// Java
class Bagel {
  public Bagel(ArrayList<String> ts) { toppings_ = ts; }
  private ArrayList<String> toppings_;
}
```

In the C++ version, the programmer states that the `toppings` are completely encapsulated by the `Bagel` class. Had the programmer intended the topping list to be shared among several bagels, it would have been declared as a pointer of some kind: `std::shared_ptr` if the ownership is shared among several bagels, or `std::weak_ptr` if someone else owns the topping list and is supposed to modify it as the program executes.

In Java, objects reference each other with shared ownership. Therefore, it's not possible to distinguish whether the topping list is intended to be shared among several bagels or not, or whether it is handled somewhere else or, if it is, as in most cases, completely owned by the `Bagel` class.

Compare the following functions; as every object is shared by default in Java (and most other languages), programmers have to take precautions for subtle bugs such as this:

C++ Java

```
// Note how the bagels do
// not share toppings:
auto t = std::set<std::string>{};
t.insert("salt");
auto a = Bagel{t};
// 'a' is not affected
// when adding pepper
t.insert("pepper");
// 'a' will have salt
// 'b' will have salt & pepper
auto b = Bagel{t};
// No bagel is affected
t.insert("oregano");
```

```
// Note how both the bagels
// share toppings:
TreeSet<String> t =
  new TreeSet<String>();
t.add("salt");
Bagel a = new Bagel(t);
// Now 'a' will subtly
// also have pepper
t.add("pepper");
// 'a' and 'b' share the
// toppings in 't'
Bagel b = new Bagel(t);
// Both bagels are affected
toppings.add("oregano");
```

Const correctness

Another powerful feature of C++, which Java and many other languages lack, is the ability to write fully `const` correct code. Const correctness means that each member function signature of a class explicitly tells the caller whether the object will be modified or not; and it will not compile if the caller tries to modify an object declared `const`. In Java, it is possible to declare constants using

the `final` keyword, but this lacks the ability to declare member functions as `const`.

Here is an example of how we can use `const` member functions to prevent unintentional modifications of objects. In the following `Person` class, the member function `age()` is declared `const` and is therefore not allowed to mutate the `Person` object, whereas `set_age()` mutates the object and *cannot* be declared `const`:

```
class Person {
public:
  auto age() const { return age_; }
  auto set_age(int age) { age_ = age; }
private:
  int age_{};
};
```

It's also possible to distinguish between returning mutable and immutable references to members. In the following `Team` class, the member function `leader()` `const` returns an immutable `Person`, whereas `leader()` returns a `Person` object that may be mutated:

```
class Team {
public:
  auto& leader() const { return leader_; }
  auto& leader() { return leader_; }
```

```
 private:
  Person leader_{};
};
```

Now let's see how the compiler can help us find errors when we try to mutate immutable objects. In the following example, the function argument `teams` is declared `const`, explicitly showing that this function is not allowed to modify them:

```
void nonmutating_func(const std::vector<Team>& teams) {
  auto tot_age = 0;

  // Compiles, both leader() and age() are declared const
  for (const auto& team : teams)
    tot_age += team.leader().age();
  // Will not compile, set_age() requires a mutable object
  for (auto& team : teams)
    team.leader().set_age(20);
}
```

If we want to write a function that *can* mutate the `teams` object, we simply remove `const`. This signals to the caller that this function may mutate the `teams`:

```
void mutating_func(std::vector<Team>& teams) {
  auto tot_age = 0;
```

```
  // Compiles, const functions can be called on mutable objects
  for (const auto& team : teams)
    tot_age += team.leader().age();
  // Compiles, teams is a mutable variable
  for (auto& team : teams)
    team.leader().set_age(20);
}
```

Object ownership

Except in very rare situations, a C++ programmer should leave the memory handling to containers and smart pointers, and never have to rely on manual memory handling.

To put it clearly, the garbage collection model in Java could almost be emulated in C++ by using `std::shared_ptr` for every object. Note that garbage-collecting languages don't use the same algorithm for allocation tracking as `std::shared_ptr`. The `std::shared_ptr` is a smart pointer based on a reference-counting algorithm that will leak memory if objects have cyclic dependencies. Garbage-collecting languages have more sophisticated methods that can handle and free cyclic dependent objects.

However, rather than relying on a garbage collector, forcing a strict ownership delicately avoids subtle bugs that may result from sharing objects by default, as in the case of Java.

If a programmer minimizes shared ownership in C++, the resulting code is easier to use and harder to abuse, as it can force the user of the class to use it as it is intended.

Deterministic destruction in C++

The destruction of objects is deterministic in C++. That means that we (can) know exactly when an object is being destroyed. This is not the case for garbage-collected languages like Java where the garbage collector decides when an unreferenced object is being finalized.

In C++, we can reliably reverse what has been done during the lifetime of an object. At first, this might seem like a small thing. But it turns out to have a great impact on how we can provide exception safety guarantees and handle resources (such as memory, file handles, mutex locks, and more) in C++.

Deterministic destruction is also one of the features that makes C++ predictable. Something that is highly valued among programmers and a requirement for performance-critical applications.

We will spend more time talking about object ownership, lifetimes, and resource management later on in the book. So don't be too worried if this doesn't make much sense at the moment.

Avoiding null objects using C++ references

In addition to strict ownership, C++ also has the concept of references, which is different from references in Java. Internally, a reference is a pointer that is not allowed to be null or repointed; therefore, no copying is involved when passing it to a function.

As a result, a function signature in C++ can explicitly restrict the programmer from passing a null object as a parameter. In Java, the programmer must use documentation or annotations to indicate non-null parameters.

Take a look at these two Java functions for computing the volume of a sphere. The first one throws a runtime exception if a null object is passed to it, whereas the second one silently ignores null objects.

This first implementation in Java throws a runtime exception if passed a null object:

```
// Java
float getVolume1(Sphere s) {
  float cube = Math.pow(s.radius(), 3);
  return (Math.PI * 4 / 3) * cube;
}
```

This second implementation in Java silently handles null objects:

```
// Java
float getVolume2(Sphere s) {
  float rad = s == null ? 0.0f : s.radius();
  float cube = Math.pow(rad, 3);
  return (Math.PI * 4 / 3) * cube;
}
```

In both functions implemented in Java, the caller of the function has to inspect the implementation of the function in order to determine whether null objects are allowed or not.

In C++, the first function signature explicitly accepts only initialized objects by using a reference that cannot be null. The second version using a pointer as an argument explicitly shows that null objects are handled.

C++ arguments passed as references indicates that null values are not allowed:

```
auto get_volume1(const Sphere& s) {
  auto cube = std::pow(s.radius(), 3.f);
  auto pi = 3.14f;
  return (pi * 4.f / 3.f) * cube;
}
```

C++ arguments passed as pointers indicates that null values are being handled:

```
auto get_volume2(const Sphere* s) {
  auto rad = s ? s->radius() : 0.f;
  auto cube = std::pow(rad, 3);
  auto pi = 3.14f;
  return (pi * 4.f / 3.f) * cube;
}
```

Being able to use references or values as arguments in C++ instantly informs the C++ programmer how the function is intended to be used. Conversely, in Java, the user must inspect the implementation of the function, as objects are always passed as pointers, and there's a possibility that they could be null.

Drawbacks of C++

Comparing C++ with other programming languages wouldn't be fair without mentioning some of its drawbacks. As mentioned earlier, C++ has more concepts to learn, and is therefore harder to use correctly and to its full potential. However, if a programmer can master C++, the higher complexity turns into an advantage and the code base becomes more robust and performs better.

There are, nonetheless, some shortcomings of C++, which are simply just shortcomings. The most severe of those shortcomings are long compilation times and the complexity of importing libraries. Up until C++20, C++ has relied on an outdated import system where imported headers are simply pasted into whatever includes them. C++ modules, which are being introduced in C++20, will solve some of the problems of the system, which is based on including header files, and will also have a positive impact on compilation times for large projects.

Another apparent drawback of C++ is the lack of provided libraries. While other languages usually come with all the libraries needed for most applications, such as graphics, user interfaces, networking, threading, resource handling, and so on, C++ provides, more or less, nothing more than the bare minimum of algorithms, threads, and, as of C++17, file system handling. For everything else, programmers have to rely on external libraries.

To summarize, although C++ has a steeper learning curve than most other languages, if used correctly, the robustness of C++ is an advantage compared to many other languages. So, despite the compilation times and lack of provided libraries, I believe that C++ is a well-suited language for large-scale projects, even for projects where performance is not the highest priority.

Libraries and compilers used in this book

As mentioned earlier, C++ does not provide more than the bare necessities in terms of libraries. In this book, we will, therefore, have to rely on external libraries where necessary. The most commonly used library in the world of C++ is probably the Boost library (**http://www.boost.org**).

Some parts of this book use the Boost library where the standard C++ library is not enough. We will only use the header-only parts of the Boost library, which means that using them yourself does not require any specific build setup; rather, you just have to include the specified header file.

In addition, we will use Google Benchmark, a microbenchmark support library, to evaluate the performance of small code snippets. Google Benchmark will be introduced in *Chapter 3*, *Analyzing and Measuring Performance*.

The repository available at **https://github.com/PacktPublishing/Cpp-High-Performance-Second-Edition** with the accompanying source code of the book uses the Google Test framework to make it easier for you to build, run, and test the code.

It should also be mentioned that this book uses a lot of new features from C++20. At the time of writing, some of these features are not fully implemented by the compilers we use (Clang, GCC, and Microsoft Visual C++). Some of the features presented are completely missing or are only supported experimentally. An excellent up-to-date summary of the current status of the major C++ compilers can be found at **https://en.cppreference.com/w/cpp/compiler_support**.

Summary

In this chapter, I have highlighted some features and drawbacks of C++ and how it has evolved to the state it is in today. Further, we discussed the advantages and disadvantages of C++ compared with other languages, both from the perspective of performance and robustness.

In the next chapter, we will explore some modern and essential C++ features that have had a major impact on how the language has developed.