# C++11/14/17/20 On the Fly

Changkun Ou (hi@changkun.us)

Last update: July 16, 2019

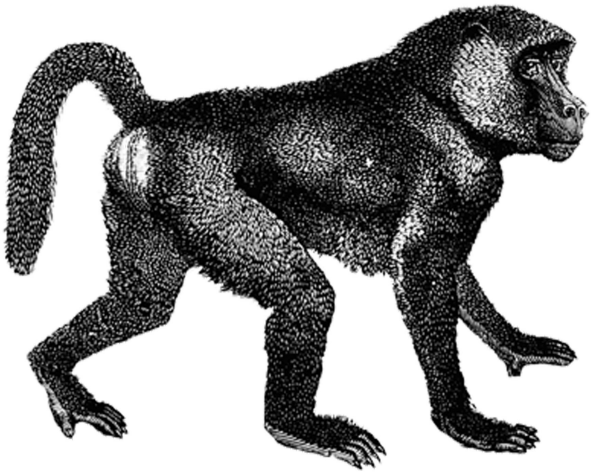
### License



*Everything is compiler.*

C++11/14/17/20
On the Fly
Second Edition

*The Fastest Guide towards Modern C++*



Ou Changkun
github.com/changkun/modern-cpp-tutorial

# Contents

# Preface

## Introduction

C++ user group is a fairly large. From the advent of C++98 to the official finalization of C++11, it has accumulated over a decade. C++14/17 is an important complement and optimization for C++11, and C++20 brings this language to the door of modernization. The extended features of all these new standards are given to the C++ language. Infused with new vitality. C++ programmers, who are still using **traditional C++** (this book refers to C++98 and its previous C++ standards as traditional C++), may even amzed by the fact that they are not using the same language while reading C++11/14/17/20 code.

**Modern C++** (this book refers to C++11/14/17/20) introduces a lot of features into traditional C++, which makes the whole C++ become language that modernized. Modern C++ not only enhances the usability of the C++ language itself, but the modification of the `auto` keyword semantics gives us more confidence in manipulating extremely complex template types. At the same time, a lot of enhancements have been made to the language runtime. The emergence of Lambda expressions has made C++ have the "closure" feature of "anonymous functions", which is almost in modern programming languages (such as Python/Swift/.. It has become commonplace, and the emergence of rvalue references has solved the problem of temporary object efficiency that C++ has long been criticized.

C++17 is the direction that has been promoted by the C++ community in the past three years. It also points out an important development direction of **modern C++** programming. Although it does not appear as much as C++11, it contains a large number of small and beautiful languages and features (such as structured binding), and the appearance of these features once again corrects our programming paradigm in C++.

Modern C++ also adds a lot of tools and methods to its own standard library, such as `std::thread` at the level of the language itself, which supports concurrent programming and no longer depends on the underlying system on different platforms. The API implements cross-platform support at the language level; `std::regex` provides full regular expression support and more. C++98 has been proven to be a very successful "paradigm", and the emergence of modern C++ further promotes this paradigm, making C++ a better language for system programming and library development. Concepts provide verification on the compile-time of template parameters, further enhancing the usability of the language.

In conclusion, as an advocate and practitioner of C++, we always maintain an open mind to accept new things, and we can promote the development of C++ faster, making this old and novel language more vibrant.

## Targets

- This book assumes that readers are already familiar with traditional C++ (i.e. C++98 or earlier), at least they do not have any difficulty in reading traditional C++ code. In other words, those who have long experience in traditional C++ and people who desire to quickly understand the features

of modern C++ in a short period of time are well suited to read the book;

- This book introduces to a certain extent of the dark magic of modern C++. However, these magics are very limited, they are not suitable for readers who want to learn advanced C++. The purpose of this book is offering a quick start for modern C++. Of course, advanced readers can also use this book to review and examine themselves on modern C++.

### Purpose

The book claims "On the Fly". Its intent is to provide a comprehensive introduction to the relevant features regarding modern C++ (before 2020s). Readers can choose interesting content according to the following table of content to learn and quickly familiarize the new features you would like to learn. Readers should aware that all of these features are not required. It should be leart when you really need it.

At the same time, instead of grammar-only, the book introduces the historical background as simple as possible of its technical requirements, which provides great help in understanding why these features comes out.

In addition, The author would like to encourage that readers should be able to use modern C++ directly in their new projects and migrate their old projects to modern C++ gradually after read the book.

### Code

Each chapter of this book has a lot of code. If you encounter problems when writing your own code with the introductory features of the book, you might as well read the source code attached to the book. You can find the book here. All the code organized by chapter, the folder name is the chapter number.

### Exercises

There are few exercises At the end of each chapter of the book. It is for testing whether you can use the knowledge points in the current chapter. You can find the possible answer to the problem from here. The folder name is the chapter number.

## Chapter 01: Towards Modern C++

**Compilation Environment**: This book will use `clang++` as the only compiler used, and always use the `-std=c++2a` compilation flag in your code.

```
> clang++ -v
Apple LLVM version 10.0.1 (clang-1001.0.46.4)
Target: x86_64-apple-darwin18.6.0
```

```
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

## 1.1 Deprecated Features

Before learning modern C++, let's take a look at the main features that have been deprecated since C++11:

> **Note**: Deprecation is not completely unusable, it is only intended to imply that programmers will disappear from future standards and should be avoided. However, the deprecated features are still part of the standard library, and most of the features are actually "permanently" reserved for compatibility reasons.

- **The string literal constant is no longer allowed to be assigned to a `char *`. If you need to assign and initialize a `char *` with a string literal constant, you should use `const char *` or `auto`.** `cpp char *str = "hello world!"; // A deprecation warning will appear`
- **C++98 exception description, `unexpected_handler`, `set_unexpected()` and other related features are deprecated and should use `noexcept`.**
- **`auto_ptr` is deprecated and `unique_ptr` should be used.**
- **`register` keyword is deprecated and can be used but no longer has any practical meaning.**
- **The `++` operation of the `bool` type is deprecated.**
- **If a class has a destructor, the properties for which it generates copy constructors and copy assignment operators are deprecated.**
- **C language style type conversion is deprecated (ie using `(convert_type)`) before variables, and `static_cast`, `reinterpret_cast`, `const_cast` should be used for type conversion.**
- **In particular, some of the C standard libraries that can be used are deprecated in the latest C++17 standard, such as `<ccomplex>`, `<cstdalign>`, `<cstdbool>` and `<ctgmath>` Wait**
- … and many more

There are also other features such as parameter binding (C++11 provides `std::bind` and `std::function`), `export`, and etc. are also deprecated. These features mentioned above **If you have never used or heard of it, please don't try to understand them. You should move closer to the new standard and learn new features directly**. After all, technology is moving forward.

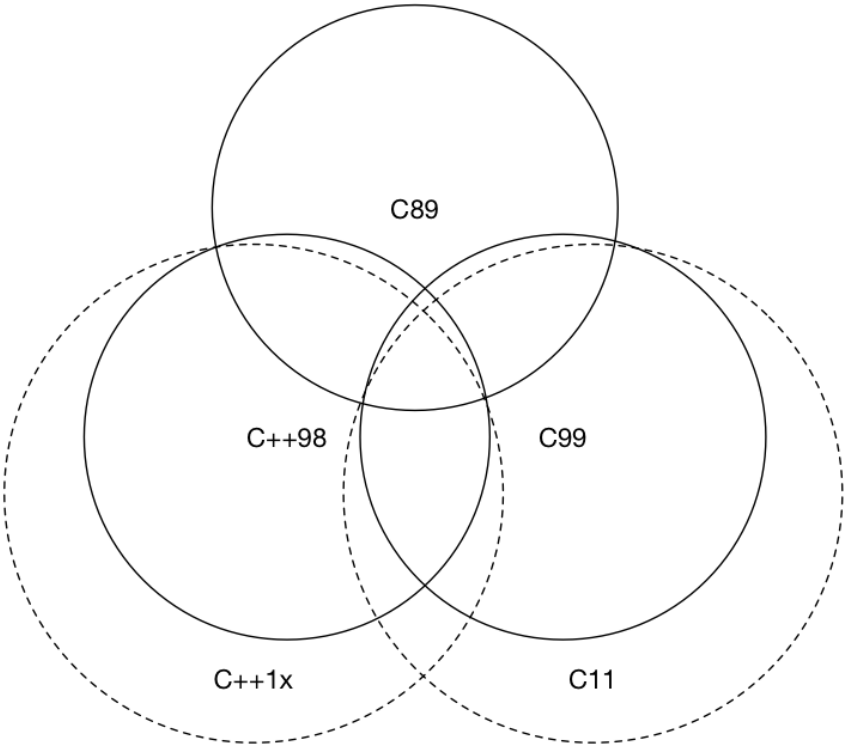



Figure 1: Figure 1.2: Compatabilities between ISO C and ISO C++

## 1.2 Compatibilities with C

For some force majeure and historical reasons, we had to use some C code (even old C code) in C++, for example, Linux system calls. Before the advent of modern C++, most people talked about "what is the difference between C and C++". Generally speaking, in addition to answering the object-oriented class features and the template features of generic programming, there is no other opinion, or even a direct answer. "Almost" is also a lot of people. The Wayne diagram in Figure 1.2 roughly answers the C and C++ related compatibility.

From now on, you should have the idea that "C++ is **not** a superset of C" in your mind (and not from the beginning, later [References for further reading] (# further reading references) The difference between C++98 and C99 is given). When writing C++, you should also avoid using program styles such as `void*` whenever possible. When you have to use C, you should pay attention to the use of `extern "C"`, separate the C language code from the C++ code, and then unify the link, for instance:

```
1  // foo.h
2  #ifdef __cplusplus
3  extern "C" {
4  #endif
5
6  int add(int x, int y);
7
8  #ifdef __cplusplus
9  }
10 #endif
```

```

// foo.c
int add(int x, int y) {
    return x+y;
}

// 1.1.cpp
#include "foo.h"
#include <iostream>
#include <functional>

int main() {
    [out = std::ref(std::cout << "Result from C code: " << add(1, 2))](){
        out.get() << ".\n";
    }();
    return 0;
}
```

You should first compile the C code with `gcc`:

```
gcc -c foo.c
```

Comple and output the `foo.o` file, and link the C++ code to the `.o` file using `clang++` (or both compile to `.o` and then unlink them together):

```
clang++ 1.1.cpp foo.o -std=c++2a -o 1.1
```

Of course, you can use `Makefile` to compile the above code:

```
C = gcc
CXX = clang++

SOURCE_C = foo.c
OBJECTS_C = foo.o

SOURCE_CXX = 1.1.cpp

TARGET = 1.1
LDFLAGS_COMMON = -std=c++2a

all:
    $(C) -c $(SOURCE_C)
    $(CXX) $(SOURCE_CXX) $(OBJECTS_C) $(LDFLAGS_COMMON) -o $(TARGET) clean: rm -rf *.o $(
    TARGET)
```

Note: Indentation in `Makefile` is a tab instead of a space character. If you copy this code directly into your editor, the tab may be automatically replaced. Please ensure the indentation

in the `Makefile`. It is done by tabs.

If you don't know the use of `Makefile`, it doesn't matter. In this tutorial, you won't build code that is written too complicated. You can also read this book by simply using `clang++ -std=c++2a` on the command line.

If you are new to modern C++, you probably still don't understand the following small piece of code above, namely:

```
[out = std::ref(std::cout << "Result from C code: " << add(1, 2))](){
    out.get() << ".\n";
}();
```

Don't worry at the moment, we will come to meet them in our later chapters.

## Further Readings

- A Tour of C++ (2nd Edition) Bjarne Stroustrup
- C++ History
- C++ compiler support
- Incompatibilities Between ISO C and ISO C++

# Chapter 02: Language Usability Enhancements

### Further Readings

# Chapter 03: Language Runtime Enhancements

### Further Readings

# Chapter 04 Standard Library: Containers

### Further Readings

# Chapter 05 Standard Library: Pointers

### Further Readings

# Chapter 06 Regular Expression

## 6.1 Introduction

Regular expressions are not part of the C++ language and therefore we only briefly introduced it here.

Regular expressions describe a pattern of string matching. The general use of regular expressions is mainly to achieve the following three requirements:

1. Check if a string contains some form of substring;
2. Replace the matching substrings;
3. Take the eligible substring from a string.

Regular expressions are text patterns consisting of ordinary characters (such as a to z) and special characters. A pattern describes one or more strings to match when searching for text. Regular expressions act as a template to match a character pattern to the string being searched.

#### Ordinary characters

Normal characters include all printable and unprintable characters that are not explicitly specified as metacharacters. This includes all uppercase and lowercase letters, all numbers, all punctuation, and some other symbols.

**Special characters**

A special character is a character with special meaning in a regular expression, and is also the core matching syntax of a regular expression. See the table below:

| Special characters | Description |
|---|---|
| `$` | Matches the end position of the input string. |
| `(,)` | Marks the start and end of a subexpression. Subexpressions can be obtained for later use. |
| `*` | Matches the previous subexpression zero or more times. |
| `+` | Matches the previous subexpression one or more times. |
| `.` | Matches any single character except the newline character `\n`. |
| `[` | Marks the beginning of a bracket expression. |
| `?` | Matches the previous subexpression zero or one time, or indicates a non-greedy qualifier. |
| `\` | Marks the next character as either a special character, or a literal character, or a backward reference, or an octal escape character. For example, `n` Matches the character `n`. `\n` matches newline characters. The sequence `\\` Matches the `'\'` character, while `\(` matches the `'('` character. |
| `^` | Matches the beginning of the input string, unless it is used in a square bracket expression, at which point it indicates that the set of characters is not accepted. |
| `{` | Marks the beginning of a qualifier expression. |
| `\` | Indicates a choice between the two. |

**Quantifiers**

The qualifier is used to specify how many times a given component of a regular expression must appear to satisfy the match. See the table below:

| Character | Description |
|---|---|
| `*` | matches the previous subexpression zero or more times. For example, `foo*` matches `fo` and `foooo`. `*` is equivalent to `{0,}`. |
| `+` | matches the previous subexpression one or more times. For example, `foo+` matches `foo` and `foooo` but does not match `fo`. `+` is equivalent to `{1,}`. |
| `?` | matches the previous subexpression zero or one time. For example, `Your(s)?` can match `Your` in `Your` or `Yours`. `?` is equivalent to `{0,1}`. |
| `{n}` | `n` is a non-negative integer. Matches the determined `n` times. For example, `o{2}` cannot match `o` in `for`, but can match two `o` in `foo`. |
| `{n,}` | `n` is a non-negative integer. Match at least `n` times. For example, `o{2,}` cannot match `o` in `for`, but matches all `o` in `fooooo`. `o{1,}` is equivalent to `o+`. `o{0,}` is equivalent to `o*`. |

| Character | Description |
|---|---|
| {n,m} | m and n are non-negative integers, where n is less than or equal to m. Matches at least n times and matches up to m times. For example, o{1,3} will match the first three o in foooooo. o{0,1} is equivalent to o?. Note that there can be no spaces between the comma and the two numbers. |

With these two tables, we can usually read almost all regular expressions.

## 6.2 std::regex and Its Related

The most common way to match string content is to use regular expressions. Unfortunately, in traditional C++, regular expressions have not been supported by the language level, and are not included in the standard library. C++ is a high-performance language. In the development of background services, the use of regular expressions is also used when judging URL resource links. The most mature and common practice in industry.

The general solution is to use the regular expression library of boost. C++11 officially incorporates the processing of regular expressions into the standard library, providing standard support from the language level and no longer relying on third parties.

The regular expression library provided by C++11 operates on the std::string object, and the pattern std::regex (essentially std::basic_regex) is initialized and matched by std::regex_match Produces std::smatch (essentially the std::match_results object).

We use a simple example to briefly introduce the use of this library. Consider the following regular expression:

- [az]+\.txt: In this regular expression, [az] means matching a lowercase letter, + can match the previous expression multiple times, so [az]+ can Matches a string of lowercase letters. In the regular expression, a . means to match any character, and \. means to match the character ., and the last txt means to match txt exactly three letters. So the content of this regular expression to match is a text file consisting of pure lowercase letters.

std::regex_match is used to match strings and regular expressions, and there are many different overloaded forms. The simplest form is to pass std::string and a std::regex to match. When the match is successful, it will return true, otherwise it will return false. For example:

```
1 #include <iostream>
2 #include <string>
3 #include <regex>
4
5 int main() {
6     std::string fnames[] = {"foo.txt", "bar.txt", "test", "a0.txt", "AAA.txt"};
```

```
    // In C++, `\` will be used as an escape character in the string. In order for `\.` to be passed as a regular expression, it is necessary to perform second escaping of `\`, thus we have `\\.`
    std::regex txt_regex("[a-z]+\\.txt");
    for (const auto &fname: fnames)
        std::cout << fname << ": " << std::regex_match(fname, txt_regex) << std::endl;
}
```

Another common form is to pass in the three arguments `std::string`/`std::smatch`/`std::regex`. The essence of `std::smatch` is actually `std::match_results`. In the standard library, `std::smatch` is defined as `std::match_results<std::string::const_iterator>`, which means `match_results` of a substring iterator type. Use `std::smatch` to easily get the matching results, for example:

```
std::regex base_regex("([a-z]+)\\.txt");
std::smatch base_match;
for(const auto &fname: fnames) {
    if (std::regex_match(fname, base_match, base_regex)) {
        // the first element of std::smatch matches the entire string
        // the second element of std::smatch matches the first expression with brackets
        if (base_match.size() == 2) {
            std::string base = base_match[1].str();
            std::cout << "sub-match[0]: " << base_match[0].str() << std::endl;
            std::cout << fname << " sub-match[1]: " << base << std::endl;
        }
    }
}
```

The output of the above two code snippets is:

```
foo.txt: 1
bar.txt: 1
test: 0
a0.txt: 0
AAA.txt: 0
sub-match[0]: foo.txt
foo.txt sub-match[1]: foo
sub-match[0]: bar.txt
bar.txt sub-match[1]: bar
```

## Conclusion

This section briefly introduces the regular expression itself, and then introduces the use of the regular expression library through a practical example based on the main requirements of using regular

expressions.

## Exercise

In web server development, we usually want to serve some routes that satisfy a certain condition. Regular expressions are one of the tools to accomplish this. Given the following request structure:

```
struct Request {
    // request method, POST, GET; path; HTTP version
    std::string method, path, http_version;
    // use smart pointer for reference counting of content
    std::shared_ptr<std::istream> content;
    // hash container, key-value dict
    std::unordered_map<std::string, std::string> header;
    // use regular expression for path match
    std::smatch path_match;
};
```

Requested resource type:

```
typedef std::map<
    std::string, std::unordered_map<
        std::string,std::function<void(std::ostream&, Request&)>>> resource_type;
```

And server template:

```
template <typename socket_type>
class ServerBase {
public:
    resource_type resource;
    resource_type default_resource;

    void start() {
        // TODO
    }
protected:
    Request parse_request(std::istream& stream) const {
        // TODO
    }
}
```

Please implement the member functions `start()` and `parse_request`. Enable server template users to specify routes as follows:

```
template<typename SERVER_TYPE>
void start_server(SERVER_TYPE &server) {

```

```
    // process GET request for /match/[digit+numbers], e.g. GET request is /match/abc123, will return abc123
    server.resource["fill_your_reg_ex"]["GET"] = [](ostream& response, Request& request) {
        string number=request.path_match[1];
        response << "HTTP/1.1 200 OK\r\nContent-Length: " << number.length() << "\r\n\r\n" << number;
    };

    // peocess default GET request; anonymous function will be called if no other matches
    // response files in folder web/
    // default: index.html
    server.default_resource["fill_your_reg_ex"]["GET"] = [](ostream& response, Request& request) {
        string filename = "www/";

        string path = request.path_match[1];

        // forbidden use `..` access content outside folder web/
        size_t last_pos = path.rfind(".");
        size_t current_pos = 0;
        size_t pos;
        while((pos=path.find('.', current_pos)) != string::npos && pos != last_pos) {
            current_pos = pos;
            path.erase(pos, 1);
            last_pos--;
        }

        // (...)
    };

    server.start();
}
```

An suggested solution can be found here.

## Further Readings

1. Comments from `std::regex`'s author
2. Library document of Regular Expression

# Chapter 07 Standard Library: Threads and Concurrency

### Further Readings

# Chapter 08 File System

The file system library provides functions related to the operation of the file system, path, regular files, directories, and so on. Similar to the regular expression library, he was one of the first libraries to be launched by boost and eventually merged into the C++ standard.

## 8.1 Document and Link

TODO:

## 8.2 std::filesystem

TODO:

### Further Readings

# Chapter 09 Minor Features

## 9.1 New Type

`long long int`

`long long int` is not the first to be introduced in C++11. In fact, as early as C99, `long long int` has been included in the C standard, so most compilers already support it. C++11 now formally incorporate it into the standard library, specifying a `long long int` type with at least 64 bits.

## 9.2 `noexcept` and Its Operations

One of the big advantages of C++ over C is that C++ itself defines a complete set of exception handling mechanisms. However, before C++11, almost no one used to write an exception declaration expression after the function name. Starting from C++11, this mechanism was deprecated, so we will not discuss or introduce the previous mechanism. How to work and how to use it, you should not take the initiative to understand it.

C++11 simplifies exception declarations into two cases:

1. The function may throw any exceptions

2. The function can't throw any exceptions

And use noexcept to limit these two behaviors, for example:

```
void may_throw();           // May throw any exception
void no_throw() noexcept;   // Cannot throw any exception
```

If a function modified with noexcept is thrown, the compiler will use std::terminate() to immediately terminate the program.

noexcept can also be used as an operator to manipulate an expression. When the expression has no exception, it returns true, otherwise it returns false.

```
#include <iostream>
void may_throw() {
    throw true;
}
auto non_block_throw = []{
    may_throw();
};
void no_throw() noexcept {
    return;
}

auto block_throw = []() noexcept {
    no_throw();
};
int main()
{
    std::cout << std::boolalpha
        << "may_throw() noexcept? " << noexcept(may_throw()) << std::endl
        << "no_throw() noexcept? " << noexcept(no_throw()) << std::endl
        << "lmay_throw() noexcept? " << noexcept(non_block_throw()) << std::endl
        << "lno_throw() noexcept? " << noexcept(block_throw()) << std::endl;
    return 0;
}
```

noexcept can modify the function of blocking exceptions after modifying a function. If an exception is generated internally, the external will not trigger. For instance:

```
try {
    may_throw();
} catch (...) {
    std::cout << "exception captured from my_throw()" << std::endl;
}
try {
    non_block_throw();
```

```
} catch (...) {
    std::cout << "exception captured from non_block_throw()" << std::endl;
}
try {
    block_throw();
} catch (...) {
    std::cout << "exception captured from block_throw()" << std::endl;
}
```

The final output is:

```
exception captured, from my_throw()
exception captured, from non_block_throw()
```

## 9.3 Literal

### Raw String Literal

In traditional C++, it is very painful to write a string full of special characters. For example, a string containing HTML ontology needs to add a large number of escape characters. For example, a file path on Windows often as: `C:\\Path\\To\\File`.

C++11 provides the original string literals, which can be decorated with `R` in front of a string, and the original string is wrapped in parentheses, for example:

```
#include <iostream>
#include <string>

int main() {
    std::string str = R"(C:\Path\To\File)";
    std::cout << str << std::endl;
    return 0;
}
```

### Custom Literal

C++11 introduces the ability to customize literals by overloading the double quotes suffix operator:

```
// String literal customization must be set to the following parameter list
std::string operator"" _wow1(const char *wow1, size_t len) {
    return std::string(wow1)+"woooooooooow, amazing";
}

std::string operator"" _wow2 (unsigned long long i) {
    return std::to_string(i)+"woooooooooow, amazing";
```

```
}

int main() {
    auto str = "abc"_wow1;
    auto num = 1_wow2;
    std::cout << str << std::endl;
    std::cout << num << std::endl;
    return 0;
}
```

Custom literals support four literals:

1. Integer literal: When overloading, you must use `unsigned long long`, `const char *`, and template literal operator parameters. The former is used in the above code;
2. Floating-point literals: You must use `long double`, `const char *`, and template literals when overloading;
3. String literals: A parameter table of the form `(const char *, size_t)` must be used;
4. Character literals: Parameters can only be `char`, `wchar_t`, `char16_t`, `char32_t`.

### Conclusion

Several of the features introduced in this section are those that use more frequent features from modern C++ features that have not yet been introduced. `noexcept` is the most important feature. One of its features is to prevent the spread of anomalies, effective Let the compiler optimize our code to the maximum extent possible.

## Chapter 10 Outlook: Introduction of C++20

C++20 seems to be an exciting update. For example, as early as C++11, the `Concept`, which was eager to call for high-altitude but ultimately lost, is now on the line. The C++ Organizing Committee decided to vote to finalize C++20 with many proposals, such as **Concepts/Module/Coroutine/Ranges/** and so on. In this chapter we'll take a look at some of the important features that C++20 will introduce.

### Concept

Concept is a further enhancement to C++ template programming. In simple terms, the concept is a compile-time feature. It allows the compiler to evaluate template parameters at compile time, greatly enhancing our experience with template programming in C++. When programming with templates, we often encounter a variety of heinous errors. This is because we have so far been unable to check and limit template parameters. For example, the following two lines of code can cause a lot of almost unreadable compilation errors:

```cpp
#include <list>
#include <algorithm>
int main() {
    std::list<int> l = {1, 2, 3};
    std::sort(l.begin(), l.end());
    return 0;
}
```

The root cause of this code error is that `std::sort` must provide a random iterator for the sorting container, otherwise it will not be used, and we know that `std::list` does not support random access. In the conceptual language, the iterator in `std::list` does not satisfy the constraint of the concept of random iterators in `std::sort`. After introducing the concept, we can constrain the template parameters like this:

```cpp
template <typename T>
requires Sortable<T> // Sortable is a concept
void sort(T& c);
```

abbreviate as:

```cpp
template<Sortable T> // T is a Sortable typename
void sort(T& c)
```

Even use it directly as a type:

```cpp
void sort(Sortable& c); // c is a Sortable type object
```

Let's look at a practical example.

TODO:

## Module

TODO:

## Contract

TODO:

## Range

TODO:

### Coroutine

TODO:

### Conclusion

In general, I finally saw the exciting features of Concepts/Ranges/Modules in C++20. This is still full of charm for a programming language that is already in its thirties.

### Further Readings

- Why Concepts didn't make C++17
- C++11/14/17/20 Compiler Support
- C++ History

## Appendix 1: Further Study Materials

First of all, congratulations on reading this book! I hope this book has raised your interest in modern C++.

As mentioned in the introduction to this book, this book is just a book that takes you quickly to the new features of modern C++ 11/14/17/20, rather than the advanced learning practice of C++ "Black Magic". The author of course also thinks about this demand, but the content is very difficult and there are few audiences. Here, the author lists some materials that can help you learn more about modern C++ based on this book. I hope I can help you:

- C++ Reference
- CppCon YouTube Channel
- Ulrich Drepper. What Every Programmer Should Know About Memory. 2007
- to be added

## Appendix 2: Modern C++ Best Practices

In this appendix we will briefly talk about the best practices of modern C++. In general, the author's thoughts on C++'s best practices are mainly absorbed from Effective Modern C++ and C++ Style Guide. In this appendix, we will briefly discuss and use the actual examples to illustrate the methods, and introduce some of **the author's personal**, **non-common**, **non-sensible** best practices, and how to ensure the overall quality of the code.

## Common Tools

TODO:

## Coding Style

TODO:

## Overall Performance

TODO:

## Code Security

TODO:

## Maintainability

TODO:

## Portability

TODO: