Modern C++ Tutorial: C++11/14/17/20 On the Fly

Changkun Ou (hi[at]changkun.de)

Last update: June 7, 2026

Notice

The content in this PDF file may outdated, please check our website or GitHub repository for the latest book updates.

License

This work was written by Ou Changkun and licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

`https://creativecommons.org/licenses/by-nc-nd/4.0/`

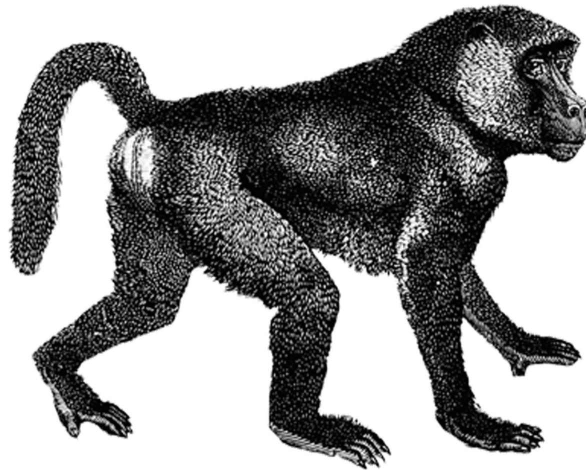
Everything is compiler.

2nd Edition

Modern C++ Tutorial

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

The Fastest Guide towards Modern C++



Ou Changkun

github.com/changkun/modern-cpp-tutorial

Contents

Preface **9**

Introduction . . . 9

Targets . . . 9

How to read this book . . . 10

Purpose . . . 10

Code . . . 11

Exercises . . . 11

Chapter 01: Towards Modern C++ **11**

1.1 Deprecated Features . . . 11

1.2 Compatibilities with C . . . 12

Further Readings . . . 15

Chapter 02: Language Usability Enhancements **15**

2.1 Constants . . . 15

nullptr . . . 15

constexpr . . . 17

2.2 Variables and initialization . . . 19

if-switch . . . 19

Initializer list . . . 20

Structured binding . . . 22

2.3 Type inference . . . 23

auto . . . 23

decltype . . . 24

tail type inference . . . 25

decltype(auto) . . . 26

2.4 Control flow . . . 27

if constexpr . . . 27

Range-based for loop . . . 28

2.5 Templates 29
Extern templates 29
The “>” 30
Type alias templates 30
Variadic templates 31
Fold expression 34
Non-type template parameter deduction 34
SFINAE and `std::enable_if` 35
2.6 Object-oriented 36
Delegate constructor 36
Inheritance constructor 37
Explicit virtual function overwrite 37
override 38
final 38
Explicit delete default function 39
Strongly typed enumerations 39
2.7 Other Language Features 40
Inline variables 40
Nested namespace definitions 41
constexpr lambda 41
Single-argument static_assert 41
New aggregate rules 42
Boolean logic metafunctions 42
`__has_include` 42
Conclusion 43
Exercises 43
Chapter 03: Language Runtime Enhancements **44**
3.1 Lambda Expression 44
Basics 44

Generic Lambda . . . 46

3.2 Function Object Wrapper . . . 47

`std::function` . . . 47

`std::bind` and `std::placeholder` . . . 48

3.3 rvalue Reference . . . 49

lvalue, rvalue, prvalue, xvalue . . . 49

rvalue reference and lvalue reference . . . 50

Move semantics . . . 52

Perfect forwarding . . . 54

Guaranteed copy elision . . . 57

Conclusion . . . 58

Further Readings . . . 58

Chapter 04: Containers **58**

4.1 Linear Container . . . 58

`std::array` . . . 58

`std::forward_list` . . . 61

4.2 Unordered Container . . . 61

4.3 Tuples . . . 62

Basic Operations . . . 62

Runtime Indexing . . . 64

Merge and Iteration . . . 65

4.4 `std::string_view` and `std::byte` . . . 66

`std::string_view` . . . 66

`std::byte` . . . 66

4.5 Associative container improvements . . . 67

4.6 Polymorphic allocators `std::pmr` . . . 67

Conclusion . . . 68

Chapter 05: Smart Pointers and Memory Management **68**

5.1 RAII and Reference Counting 68

5.2 `std::shared_ptr` 69

5.3 `std::unique_ptr` 70

5.4 `std::weak_ptr` 72

Conclusion 74

Further Readings 74

Chapter 06: Regular Expression **74**

6.1 Introduction 74

Ordinary characters 74

Special characters 74

Quantifiers 77

6.2 `std::regex` and Its Related 77

Conclusion 79

Exercise 79

Further Readings 81

Chapter 07: Parallelism and Concurrency **82**

7.1 Basic of Parallelism 82

7.2 Mutex and Critical Section 82

7.3 Future 84

7.4 Condition Variable 85

7.5 Atomic Operation and Memory Model 87

Atomic Operation 88

Consistency Model 90

Memory Orders 92

Conclusion 95

Exercises 95

Further Readings 95

Chapter 08: File System **96**

8.1 The path `std::filesystem::path` . . . 96

8.2 Querying file status . . . 96

8.3 Iterating directories . . . 97

8.4 Creating, copying, and removing . . . 97

8.5 A complete example . . . 98

Further Readings . . . 99

Chapter 09: Minor Features **99**

9.1 New Type . . . 99

`long long int` . . . 99

9.2 `noexcept` and Its Operations . . . 99

9.3 Literal . . . 101

Raw String Literal . . . 101

Custom Literal . . . 101

9.4 Memory Alignment . . . 102

Dynamic allocation of over-aligned types . . . 103

9.5 Type punning and `std::bit_cast` . . . 103

9.6 Mathematical special functions . . . 104

Conclusion . . . 104

Chapter 10: C++20 **104**

Concept . . . 105

Modules . . . 106

Ranges . . . 107

Coroutines . . . 107

A note on Contracts and Transactional Memory . . . 108

Conclusion . . . 109

Further Readings . . . 109

Appendix 1: Further Study Materials **109**

Appendix 2: Modern C++ Best Practices **109**

Common Tools 110

Coding Style 110

Overall Performance 110

Code Security 111

Maintainability 111

Portability 111

Preface

Introduction

The C++ programming language owns a fairly large user group. From the advent of C++98 to the official finalization of C++11, it has continued to stay relevant. 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 integrated into the C++ language and infuse it with new vitality. C++ programmers who are still using **traditional C++** (this book refers to C++98 and its previous standards as traditional C++) may even amazed by the fact that they are not using the same language while reading modern C++ code.

Modern C++ (this book refers to C++11 through C++26) introduces many features into traditional C++ which bring the entire language to a new level of modernization. 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 given C++ the "closure" feature of "anonymous functions", which are in almost all modern programming languages (such as Python, Swift, etc). It has become commonplace, and the emergence of rvalue references has solved the problem of temporary object efficiency that C++ has long been criticized for.

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 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 verify 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 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 to offer a quick start for modern C++. Of course, advanced readers can also use this book to review and examine themselves on modern C++.

How to read this book

Modern C++ is large, and you do not need to read every page in order. This book assumes you are already comfortable with the basics of C++ — roughly the C++11-era language: classes, templates, and the common standard-library containers. If that describes you (for example, you learned some C++ at university), feel free to skim the parts you already know and focus on what is new to you.

Several aids are built in to help you navigate:

- Each feature is marked with the standard that introduced it (for example, *(since C++17)*), so you can tell at a glance what is newer than the C++ you already know.
- Appendix 3 is a feature index that maps each feature to the chapter where it appears and the standard it came from — handy for jumping straight to "what's new in C++20", or for looking a feature up later.
- The book is organized in three parts: the **language core** (Chapters 1–3), the **standard library** (Chapters 4–9), and a **tour by standard version** (Chapters 10–12, covering C++20, C++23, and the forthcoming C++26).

Rather than trying to memorize every feature, treat this book as a reference you return to: reach for a feature when a real problem calls for it.

Purpose

The book claims "On the Fly". It intends to provide a comprehensive introduction to the relevant features of modern C++, from C++11 all the way to C++26. Readers can choose interesting content according to the following table of contents to learn and quickly familiarize themselves with the new features that are available. Readers should aware that all of these features are not required. It should be learned when you 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 come out.

Also, 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 reading 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 is 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 deprecated since C++11:

> **Note**: Deprecation is not completely unusable, it is only intended to imply that features will disappear from future standards and should be avoided. But, 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`.**

  ```
  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>` etc.**
- … and many more

There are also other features such as parameter binding (C++11 provides `std::bind` and `std::function`), `export` 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.

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 Venn 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 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:

```
// foo.h
#ifdef __cplusplus
extern "C" {
#endif
```

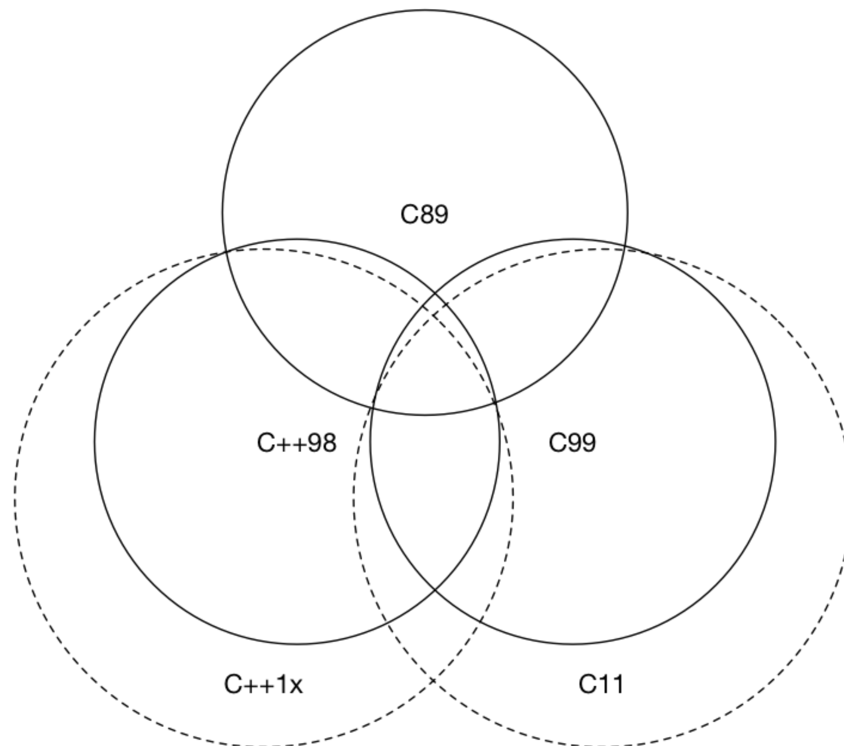


Figure 1: Figure 1.2: Compatabilities between ISO C and ISO C++

```
int add(int x, int y);

#ifdef __cplusplus
}
#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
```

Compile and output the `foo.o` file, and link the C++ code to the `.o` file using `clang++` (or both compile to `.o` and then link 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` 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 History of C++
- C++ compiler support
- Incompatibilities Between ISO C and ISO C++

Chapter 02: Language Usability Enhancements

When we declare, define a variable or constant, and control the flow of code, object-oriented functions, template programming, etc., before the runtime, it may happen when writing code or compiler compiling code. To this end, we usually talk about **language usability**, which refers to the language behavior that occurred before the runtime.

2.1 Constants

nullptr

(since C++11)

The purpose of `nullptr` appears to replace `NULL`. There are **null pointer constants** in the C and C++ languages, which can be implicitly converted to null pointer value of any pointer type, or null member pointer value of any pointer-to-member type in C++. `NULL` is provided by the standard library implementation and defined as an implementation-defined null pointer constant. In C, some standard libraries defines `NULL` as `((void*)0)` and some define it as `0`.

C++ **does not allow** to implicitly convert `void *` to other types, and thus `((void*)0)` is not a valid implementation of `NULL`. If the standard library tries to define `NULL` as `((void*)0)`, then compilation error would occur in the following code:

```
char *ch = NULL;
```

C++ without the `void *` implicit conversion has to define `NULL` as `0`. This still creates a new problem. Defining `NULL` to `0` will cause the overloading feature in `C++` to be confusing. Consider the following two `foo` functions:

```
void foo(char*);
void foo(int);
```

Then the behavior of the `foo(NULL);` statement depends on how `NULL` is implemented: when `NULL` is defined as `0` (for example, on MSVC), it calls `foo(int)`, which is counterintuitive; when `NULL` is defined as the GCC/Clang builtin `__null`, the call `foo(NULL)` becomes ambiguous between the `char*` and `int` overloads and fails to compile. In either case, `NULL` does not behave the way a proper null pointer should during overload resolution.

To solve this problem, C++11 introduced the `nullptr` keyword, which is specifically used to distinguish null pointers, `0`. The type of `nullptr` is `nullptr_t`, which can be implicitly converted to any pointer or member pointer type, and can be compared equally or unequally with them.

You can try to compile the following code using clang++:

```
#include <iostream>
#include <type_traits>

void foo(char *);
void foo(int);

int main() {
    if (std::is_same<decltype(NULL), decltype(0)>::value)
        std::cout << "NULL == 0" << std::endl;
    if (std::is_same<decltype(NULL), decltype((void*)0)>::value)
        std::cout << "NULL == (void *)0" << std::endl;
    if (std::is_same<decltype(NULL), std::nullptr_t>::value)
        std::cout << "NULL == nullptr" << std::endl;

    foo(0);          // will call foo(int)
    // foo(NULL);    // doesn't compile
    foo(nullptr);    // will call foo(char*)
    return 0;
}

void foo(char *) {
    std::cout << "foo(char*) is called" << std::endl;
}
void foo(int i) {
    std::cout << "foo(int) is called" << std::endl;
}
```

The outputs are:

```
foo(int) is called
foo(char*) is called
```

From the output we can see that `NULL` is different from `0` and `nullptr`. So, develop the habit of using `nullptr` directly.

In addition, in the above code, we used `decltype` and `std::is_same` which are modern C++ syntax. In simple terms, `decltype` is used for type derivation, and `std::is_same` is used to compare the equality of the two types. We will discuss them in detail later in the decltype section.

constexpr

(since C++11; relaxed in C++14)

C++ itself already has the concept of constant expressions, such as 1+2, 3*4. Such expressions always produce the same result without any side effects. If the compiler can directly optimize and embed these expressions into the program at compile-time, it will increase the performance of the program. A very obvious example is in the definition phase of an array:

```cpp
#include <iostream>
#define LEN 10

int len_foo() {
    int i = 2;
    return i;
}
constexpr int len_foo_constexpr() {
    return 5;
}

constexpr int fibonacci(const int n) {
    return n == 1 || n == 2 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}


int main() {
    char arr_1[10];                      // legal
    char arr_2[LEN];                     // legal

    int len = 10;
    // char arr_3[len];                  // illegal

    const int len_2 = len + 1;
    constexpr int len_2_constexpr = 1 + 2 + 3;
    // char arr_4[len_2];                // illegal, but ok for most of the compilers
    char arr_4[len_2_constexpr];         // legal

    // char arr_5[len_foo()+5];          // illegal
    char arr_6[len_foo_constexpr() + 1]; // legal

    // 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
    std::cout << fibonacci(10) << std::endl;
```

```
    return 0;
}
```

In the above example, `char arr_4[len_2]` may be confusing because `len_2` has been defined as a constant. Why is `char arr_4[len_2]` still illegal? This is because the length of the array in the C++ standard must be a constant expression, and for `len_2`, this is a `const` constant, not a constant expression, so even if this behavior is supported by most compilers, but it is an illegal behavior, we need to use the `constexpr` feature introduced in C++11, which will be introduced next, to solve this problem; for `arr_5`, before C++98 The compiler cannot know that `len_foo()` actually returns a constant at runtime, which causes illegal production.

> Note that some compilers (e.g. GCC, Clang) have compiler extensions enabled by default, supporting a C feature called "variable-length arrays", which allows defining an array whose length is a non-constant expression and causes the above commented out illegal code to be compilable. To disable the extension, add the compilation option `-pedantic-errors` (available for both GCC and Clang).

C++11 provides `constexpr` to let the user explicitly declare that the function or object constructor will become a constant expression at compile time. This keyword explicitly tells the compiler that it should verify that `len_foo` should be a compile-time constant expression.

In addition, the function of `constexpr` can use recursion:

```
constexpr int fibonacci(const int n) {
    return n == 1 || n == 2 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}
```

Starting with C++14, the constexpr function can use simple statements such as local variables, loops, and branches internally. For example, the following code cannot be compiled under the C++11 standard:

```
constexpr int fibonacci(const int n) {
    if(n == 1) return 1;
    if(n == 2) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

To do this, we can write a simplified version like this to make the function available from C++11:

```
constexpr int fibonacci(const int n) {
    return n == 1 || n == 2 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}
```

2.2 Variables and initialization

if-switch

(since C++17)

In traditional C++, the declaration of a variable can declare a temporary variable `int` even though it can be located anywhere, even within a `for` statement, but there is always no way to declare a temporary variable in the `if` and `switch` statements. E.g:

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4};

    // before C++17, can be simplified by using `auto`
    const std::vector<int>::iterator itr = std::find(vec.begin(), vec.end(), 2);
    if (itr != vec.end()) {
        *itr = 3;
    }

    // need to define a new variable
    const std::vector<int>::iterator itr2 = std::find(vec.begin(), vec.end(), 3);
    if (itr2 != vec.end()) {
        *itr2 = 4;
    }

    // will output: 1, 4, 3, 4; can be simplified using `auto`
    for (std::vector<int>::iterator element = vec.begin(); element != vec.end();
        ++element)
        std::cout << *element << std::endl;
}
```

In the above code, we can see that the `itr` variable is defined in the scope of the entire `main()`, which causes us to rename the other when a variable need to traverse the entire `std::vector` again. C++17 eliminates this limitation so that we can do this in if(or switch):

```
// put the temporary variable into the if-statement
if (const std::vector<int>::iterator itr = std::find(vec.begin(), vec.end(), 3);
    itr != vec.end()) {
    *itr = 4;
```

```
}
```

Is it similar to the Go?

Initializer list

(since C++11)

Initialization is a very important language feature, the most common one is when the object is initialized. In traditional C++, different objects have different initialization methods, such as ordinary arrays, PODs (**P**lain **O**ld **D**ata, i.e. classes without constructs, destructors, and virtual functions) Or struct type can be initialized with `{}`, which is what we call the initialization list. For the initialization of the class object, you need to use the copy construct, or you need to use `()`. These different methods are specific to each other and cannot be generic. E.g:

```cpp
#include <iostream>
#include <vector>

class Foo {
public:
    int value_a;
    int value_b;
    Foo(int a, int b) : value_a(a), value_b(b) {}
};

int main() {
    // before C++11
    int arr[3] = {1, 2, 3};
    Foo foo(1, 2);
    std::vector<int> vec = {1, 2, 3, 4, 5};

    std::cout << "arr[0]: " << arr[0] << std::endl;
    std::cout << "foo:" << foo.value_a << ", " << foo.value_b << std::endl;
    for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << std::endl;
    }
    return 0;
}
```

To solve this problem, C++11 first binds the concept of the initialization list to the type and calls it `std::initializer_list`, allowing the constructor or other function to use the initialization list like a parameter, which is the initialization of class objects provides a unified bridge between normal arrays and POD initialization methods, such as:

```
#include <initializer_list>
#include <vector>
#include <iostream>

class MagicFoo {
public:
    std::vector<int> vec;
    MagicFoo(std::initializer_list<int> list) {
        for (std::initializer_list<int>::iterator it = list.begin();
             it != list.end(); ++it)
            vec.push_back(*it);
    }
};
int main() {
    // after C++11
    MagicFoo magicFoo = {1, 2, 3, 4, 5};

    std::cout << "magicFoo: ";
    for (std::vector<int>::iterator it = magicFoo.vec.begin();
        it != magicFoo.vec.end(); ++it)
        std::cout << *it << std::endl;
}
```

This constructor is called the initialize list constructor, and the type with this constructor will be specially taken care of during initialization.

In addition to the object construction, the initialization list can also be used as a formal parameter of a normal function, for example:

```
public:
    void foo(std::initializer_list<int> list) {
        for (std::initializer_list<int>::iterator it = list.begin();
            it != list.end(); ++it) vec.push_back(*it);
    }

magicFoo.foo({6,7,8,9});
```

Second, C++11 also provides a uniform syntax for initializing arbitrary objects, such as:

```
Foo foo2 {3, 4};
```

Structured binding

(since C++17)

Functions frequently need to "return several values at once" — for example, a computed result together with a status flag. In traditional C++ this is not elegant: we either define a dedicated struct for it, or pack the values into a `std::tuple` and return that, but getting the values back out is clumsy — unpacking with `std::tie` forces us to declare every variable in advance and to know exactly how many elements the tuple holds and the type of each, and any mismatch is an error.

C++17's **structured bindings** exist precisely to remove that clumsiness: they let us, in a single line, "unpack" a tuple, a `std::pair`, a raw array, or a struct with public data members, and bind the pieces directly to a set of named variables, with the types deduced by the compiler:

```cpp
#include <iostream>
#include <tuple>

std::tuple<int, double, std::string> f() {
    return std::make_tuple(1, 2.3, "456");
}

int main() {
    auto [x, y, z] = f();
    std::cout << x << ", " << y << ", " << z << std::endl;
    return 0;
}
```

Compared with `std::tie`, structured bindings need no prior declaration and no spelled-out types, and they work not only on tuples but also on raw arrays and aggregate structs. This is especially handy when iterating an associative container: we can bind each key/value pair to meaningful names instead of writing `it->first` / `it->second`:

```cpp
#include <iostream>
#include <map>
#include <string>

std::map<std::string, int> scores{{"alice", 90}, {"bob", 80}};
for (const auto& [name, score] : scores) {
    std::cout << name << ": " << score << '\n';
}
```

The `auto` type derivation is described in the auto type inference section.

2.3 Type inference

In traditional C and C++, the types of parameters must be clearly defined, which does not help us to quickly encode, especially when we are faced with a large number of complex template types, we must indicate the type of variables to proceed. Subsequent coding, which not only slows down our development efficiency but also makes the code stinking and long.

C++11 introduces the two keywords `auto` and `decltype` to implement type derivation, letting the compiler worry about the type of the variable. This makes C++ the same as other modern programming languages, in a way that provides the habit of not having to worry about variable types.

auto

(since C++11)

`auto` has been in C++ for a long time, but it always exists as an indicator of a storage type, coexisting with `register`. In traditional C++, if a variable is not declared as a `register` variable, it is automatically treated as an `auto` variable. And with `register` being deprecated (used as a reserved keyword in C++17 and later used, it doesn't currently make sense), the semantic change to `auto` is very natural.

One of the most common and notable examples of type derivation using `auto` is the iterator. You should see the lengthy iterative writing in traditional C++ in the previous section:

```
// before C++11
// cbegin() returns vector<int>::const_iterator
// and therefore it is type vector<int>::const_iterator
for(vector<int>::const_iterator it = vec.cbegin(); it != vec.cend(); ++it)
```

When we have `auto`:

```
#include <initializer_list>
#include <vector>
#include <iostream>

class MagicFoo {
public:
    std::vector<int> vec;
    MagicFoo(std::initializer_list<int> list) {
        for (auto it = list.begin(); it != list.end(); ++it) {
            vec.push_back(*it);
        }
    }
};
```

```
int main() {
    MagicFoo magicFoo = {1, 2, 3, 4, 5};
    std::cout << "magicFoo: ";
    for (auto it = magicFoo.vec.begin(); it != magicFoo.vec.end(); ++it) {
        std::cout << *it << ", ";
    }
    std::cout << std::endl;
    return 0;
}
```

Some other common usages:

```
auto i = 5;              // i as int
auto arr = new auto(10); // arr as int *
```

Since C++ 14, `auto` can even be used as function arguments in generic lambda expressions, and such functionality is generalized to normal functions in C++ 20. Consider the following example:

```
auto add14 = [](auto x, auto y) -> int {
    return x+y;
}

int add20(auto x, auto y) {
    return x+y;
}

auto i = 5; // type int
auto j = 6; // type int
std::cout << add14(i, j) << std::endl;
std::cout << add20(i, j) << std::endl;
```

Note: `auto` cannot be used to derive array types yet:

```
auto auto_arr2[10] = {arr};   // illegal, can't infer array type
```

```
2.6.auto.cpp:30:19: error: 'auto_arr2' declared as array of 'auto'
    auto auto_arr2[10] = {arr};
```

decltype

(since C++11)

The `decltype` keyword is used to solve the defect that the `auto` keyword can only deduce the type of a variable. Its usage is very similar to `typeof`, a non-standard extension long provided by some compilers (e.g. GCC and Clang) that was eventually standardized in C23 but has never been part of standard C++:

```
decltype(expression)
```

Sometimes we may need to calculate the type of an expression, for example:

```
auto x = 1;
auto y = 2;
decltype(x+y) z;
```

You have seen in the previous example that `decltype` is used to infer the usage of the type. The following example is to determine if the above variables `x, y, z` are of the same type:

```
if (std::is_same<decltype(x), int>::value)
    std::cout << "type x == int" << std::endl;
if (std::is_same<decltype(x), float>::value)
    std::cout << "type x == float" << std::endl;
if (std::is_same<decltype(x), decltype(z)>::value)
    std::cout << "type z == type x" << std::endl;
```

Among them, `std::is_same<T, U>` is used to determine whether the two types `T` and `U` are equal. The output is:

```
type x == int
type z == type x
```

tail type inference

(since C++11)

You may think that whether `auto` can be used to deduce the return type of a function. Still consider an example of an add function, which we have to write in traditional C++:

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

> Note: There is no difference between typename and class in the template parameter list. Before the keyword typename appears, class is used to define the template parameters. However, when defining a variable with nested dependency type in the template, you need to use typename to eliminate ambiguity.

Such code is very ugly because the programmer must explicitly indicate the return type when using this template function. But in fact, we don't know what kind of operation the `add()` function will do, and what kind of return type to get.

This problem was solved in C++11. Although you may immediately react to using `decltype` to derive the type of `x+y`, write something like this:

```
decltype(x+y) add(T x, U y)
```

But in fact, this way of writing can not be compiled. This is because `x` and `y` have not been defined when the compiler reads decltype(x+y). To solve this problem, C++11 also introduces a trailing return type, which uses the auto keyword to post the return type:

```
template<typename T, typename U>
auto add2(T x, U y) -> decltype(x+y){
    return x + y;
}
```

The good news is that from C++14 it is possible to directly derive the return value of a normal function, so the following way becomes legal:

```
template<typename T, typename U>
auto add3(T x, U y){
    return x + y;
}
```

You can check if the type derivation is correct:

```
// after c++11
auto w = add2<int, double>(1, 2.0);
if (std::is_same<decltype(w), double>::value) {
    std::cout << "w is double: ";
}
std::cout << w << std::endl;

// after c++14
auto q = add3<double, int>(1.0, 2);
std::cout << "q: " << q << std::endl;
```

decltype(auto)

(since C++14)

`decltype(auto)` is a slightly more complicated use of C++14.

> To understand it you need to know the concept of parameter forwarding in C++, which we will cover in detail in the Language Runtime Enhancements chapter, and you can come back to the contents of this section later.

In simple terms, `decltype(auto)` is mainly used to derive the return type of a forwarding function or package, which does not require us to explicitly specify the parameter expression of `decltype`. Consider the following example, when we need to wrap the following two functions:

```
std::string  lookup1();
std::string& lookup2();
```

In C++11:

```
std::string look_up_a_string_1() {
    return lookup1();
}
std::string& look_up_a_string_2() {
    return lookup2();
}
```

With `decltype(auto)`, we can let the compiler do this annoying parameter forwarding:

```
decltype(auto) look_up_a_string_1() {
    return lookup1();
}
decltype(auto) look_up_a_string_2() {
    return lookup2();
}
```

2.4 Control flow

if constexpr

(since C++17)

As we saw at the beginning of this chapter, we know that C++11 introduces the `constexpr` keyword, which compiles expressions or functions into constant results. A natural idea is that if we introduce this feature into the conditional judgment, let the code complete the branch judgment at compile-time, can it make the program more efficient? C++17 introduces the `constexpr` keyword into the `if` statement, allowing you to declare the condition of a constant expression in your code. Consider the following code:

```
#include <iostream>
```

```
template<typename T>
auto print_type_info(const T& t) {
    if constexpr (std::is_integral<T>::value) {
        return t + 1;
    } else {
        return t + 0.001;
    }
}
int main() {
    std::cout << print_type_info(5) << std::endl;
    std::cout << print_type_info(3.14) << std::endl;
}
```

At compile time, the actual code will behave as follows:

```
int print_type_info(const int& t) {
    return t + 1;
}
double print_type_info(const double& t) {
    return t + 0.001;
}
int main() {
    std::cout << print_type_info(5) << std::endl;
    std::cout << print_type_info(3.14) << std::endl;
}
```

Range-based for loop

(since C++11)

Finally, C++11 introduces a range-based iterative method, and we can write loops that are as concise as Python, and we can further simplify the previous example:

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4};
    if (auto itr = std::find(vec.begin(), vec.end(), 3); itr != vec.end()) *itr = 4;
    for (auto element : vec)
        std::cout << element << std::endl; // read only
    for (auto &element : vec) {
```

```
        element += 1;                          // writeable
    }
    for (auto element : vec)
        std::cout << element << std::endl; // read only
}
```

A range-based for loop is essentially syntactic sugar. The compiler expands

```
for (range_declaration : range_expression) loop_statement
```

roughly into (since C++17):

```
{
    auto && __range = range_expression;
    auto __begin = begin_expr; // __range for an array; __range.begin() or begin(__range) for a clas
    auto __end = end_expr;      // __range + N for an array; __range.end() or end(__range) for a clas
    for (; __begin != __end; ++__begin) {
        range_declaration = *__begin;
        loop_statement
    }
}
```

Therefore, any type that provides usable `begin()` and `end()` (member functions, or free functions found via ADL) whose returned iterators support `!=`, dereference `*`, and pre-increment `++` can be traversed by a range-based for loop — which is exactly how you make a custom container work with it.

2.5 Templates

C++ templates have always been a special art of the language, and templates can even be used independently as a new language. The philosophy of the template is to throw all the problems that can be processed at compile time into the compile time, and only deal with those core dynamic services at runtime, to greatly optimize the performance of the runtime. Therefore, templates are also regarded by many as one of the black magic of C++.

Extern templates

(since C++11)

In traditional C++, templates are instantiated by the compiler only when they are used. In other words, as long as a fully defined template is encountered in the code compiled in each compilation unit (file), it will be instantiated. This results in an increase in compile time due to repeated instantiations. Also, we have no way to tell the compiler not to trigger the instantiation of the template.

To this end, C++11 introduces an external template that extends the syntax of the original mandatory compiler to instantiate a template at a specific location, allowing us to explicitly tell the compiler when to instantiate the template:

```
template class std::vector<bool>;          // force instantiation
extern template class std::vector<double>; // should not instantiation in current file
```

The “>”

(since C++11)

In the traditional C++ compiler, `>>` is always treated as a right shift operator. But actually we can easily write the code for the nested template:

```
std::vector<std::vector<int>> matrix;
```

This is not compiled under the traditional C++ compiler, and C++11 starts with continuous right angle brackets that become legal and can be compiled successfully. Even the following writing can be compiled by:

```
template<bool T>
class MagicType {
    bool magic = T;
};

// in main function:
std::vector<MagicType<(1>2)>> magic; // legal, but not recommended
```

Type alias templates

(since C++11)

Before you understand the type alias template, you need to understand the difference between “template” and “type”. Carefully understand this sentence: **Templates are used to generate types.** In traditional C++, `typedef` can define a new name for the type, but there is no way to define a new name for the template. Because the template is not a type. E.g:

```
template<typename T, typename U>
class MagicType {
public:
    T dark;
    U magic;
};
```

```
// not allowed
template<typename T>
typedef MagicType<std::vector<T>, std::string> FakeDarkMagic;
```

C++11 uses `using` to introduce the following form of writing, and at the same time supports the same effect as the traditional `typedef`:

> Usually, we use `typedef` to define the alias syntax: `typedef original name new name;`, but the definition syntax for aliases such as function pointers is different, which usually causes a certain degree of difficulty for direct reading.

```
typedef int (*process)(void *);
using NewProcess = int(*)(void *);
template<typename T>
using TrueDarkMagic = MagicType<std::vector<T>, std::string>;

int main() {
    TrueDarkMagic<bool> you;
}
```

Variadic templates

(since C++11)

The template has always been one of C++'s unique **Black Magic**. In traditional C++, both a class template and a function template could only accept a fixed set of template parameters as specified; C++11 added a new representation, allowing any number, template parameters of any category, and there is no need to fix the number of parameters when defining.

```
template<typename... Ts> class Magic;
```

The template class Magic object can accept an unrestricted number of typename as a formal parameter of the template, such as the following definition:

```
class Magic<int,
            std::vector<int>,
            std::map<std::string,
            std::vector<int>>> darkMagic;
```

Since it is arbitrary, a template parameter with a number of 0 is also possible: `class Magic<> nothing;`.

If you do not want to generate 0 template parameters, you can manually define at least one template parameter:

```
template<typename Require, typename... Args> class Magic;
```

The variable length parameter template can also be directly adjusted to the template function. The `printf` function in the traditional C, although it can also reach the call of an indefinite number of formal parameters, is not class safe. In addition to the variable-length parameter functions that define class safety, C++11 can also make printf-like functions naturally handle objects that are not self-contained. In addition to the use of `...` in the template parameters to indicate the indefinite length of the template parameters, the function parameters also use the same representation to represent the indefinite length parameters, which provides a convenient means for us to simply write variable length parameter functions, such as:

```
template<typename... Args> void printf(const std::string &str, Args... args);
```

Then we define variable length template parameters, how to unpack the parameters?

First, we can use `sizeof...` to calculate the number of arguments:

```
#include <iostream>
template<typename... Ts>
void magic(Ts... args) {
    std::cout << sizeof...(args) << std::endl;
}
```

We can pass any number of arguments to the `magic` function:

```
magic();      // 0
magic(1);     // 1
magic(1, ""); // 2
```

Second, the parameters are unpacked. So far there is no simple way to process the parameter package, but there are two classic processing methods:

1. Recursive template function

Recursion is a very easy way to think of and the most classic approach. This method continually recursively passes template parameters to the function, thereby achieving the purpose of recursively traversing all template parameters:

```
#include <iostream>
template<typename T0>
void printf1(T0 value) {
```

```
    std::cout << value << std::endl;
}
template<typename T, typename... Ts>
void printf1(T value, Ts... args) {
    std::cout << value << std::endl;
    printf1(args...);
}
int main() {
    printf1(1, 2, "123", 1.1);
    return 0;
}
```

2. Variable parameter template expansion

You should feel that this is very cumbersome. Added support for variable parameter template expansion in C++17, so you can write `printf` in a function:

```
template<typename T0, typename... T>
void printf2(T0 t0, T... t) {
    std::cout << t0 << std::endl;
    if constexpr (sizeof...(t) > 0) printf2(t...);
}
```

> In fact, sometimes we use variable parameter templates, but we don't necessarily need to traverse the parameters one by one. We can use the features of `std::bind` and perfect forwarding to achieve the binding of functions and parameters, thus achieving success. The purpose of the call.

3. Initialize list expansion

Recursive template functions are standard practice, but the obvious drawback is that you must define a function that terminates recursion.

Here is a description of the black magic that is expanded using the initialization list:

```
template<typename T, typename... Ts>
auto printf3(T value, Ts... args) {
    std::cout << value << std::endl;
    (void) std::initializer_list<T>{([&args] {
        std::cout << args << std::endl;
    }(), value)...};
}
```

In this code, the initialization list provided in C++11 and the properties of the Lambda expression (mentioned in the next section) are additionally used.

By initializing the list, `(lambda expression, value)...` will be expanded. Due to the appearance of the comma expression, the previous lambda expression is executed first, and the output of the parameter is completed. To avoid compiler warnings, we can explicitly convert `std::initializer_list` to `void`.

Fold expression

(since C++17)

In C++ 17, this feature of the variable length parameter is further brought to the expression, consider the following example:

```
#include <iostream>
template<typename ... T>
auto sum(T ... t) {
    return (t + ...);
}
int main() {
    std::cout << sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) << std::endl;
}
```

Non-type template parameter deduction

(since C++17)

What we mainly mentioned above is a form of template parameters: type template parameters.

```
template <typename T, typename U>
auto add(T t, U u) {
    return t+u;
}
```

The parameters of the template `T` and `U` are specific types. But there is also a common form of template parameter that allows different literals to be template parameters, i.e. non-type template parameters:

```
template <typename T, int BufSize>
class buffer_t {
public:
    T& alloc();
    void free(T& item);
private:
    T data[BufSize];
```

```
};

buffer_t<int, 100> buf; // 100 as template parameter
```

In this form of template parameters, we can pass `100` as a parameter to the template. After C++11 introduced the feature of type derivation, we will naturally ask, since the template parameters here. Passing with a specific literal, can the compiler assist us in type derivation, By using the placeholder `auto`, there is no longer a need to explicitly specify the type? Fortunately, C++17 introduces this feature, and we can indeed use the `auto` keyword to let the compiler assist in the completion of specific types of derivation. E.g:

```
template <auto value> void foo() {
    std::cout << value << std::endl;
    return;
}

int main() {
    foo<10>();  // value as int
}
```

SFINAE and `std::enable_if`

(since C++11)

SFINAE stands for "Substitution Failure Is Not An Error". It describes the rule that, when substituting template arguments produces an invalid type or expression in the **immediate context**, the compiler does not raise an error but silently removes that candidate from the overload set. This was the main way to constrain template parameters before C++20's concepts.

The most common tool is `std::enable_if` from `<type_traits>`. The `describe` below is visible only for integral types:

```
#include <type_traits>

template <typename T,
          typename = std::enable_if_t<std::is_integral_v<T>>>
void describe(T) {
    std::cout << "integral" << std::endl;
}

describe(42);    // OK
// describe(3.14); // compile error: floating point does not satisfy the constraint
```

Another common form is **expression SFINAE**, which uses `decltype` to probe whether an expression is valid, thereby detecting at compile time whether a type has a certain capability:

```
// participates only if c.size() is a valid expression
template <typename T>
auto has_size(const T& c) -> decltype(c.size(), std::true_type{}) {
    return std::true_type{};
}
std::false_type has_size(...) { return std::false_type{}; }
```

SFINAE is powerful but obscure to write and produces verbose error messages. C++20's concepts were introduced precisely to express such constraints in a more intuitive and readable way, and can be regarded as the modern replacement for SFINAE.

2.6 Object-oriented

Delegate constructor

(since C++11)

A class often has several constructors that share a lot of the same initialization logic. Before C++11, the only ways to reuse that logic were to copy it into every constructor, or to extract a private init function — but the latter cannot initialize `const` members or reference members. C++11's **delegating constructors** let one constructor delegate its initialization to another constructor of the same class, removing that duplication:

```
#include <iostream>
class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
    }
    Base(int value) : Base() { // delegate Base() constructor
        value2 = value;
    }
};

int main() {
    Base b(2);
    std::cout << b.value1 << std::endl;
```

```
    std::cout << b.value2 << std::endl;
}
```

Inheritance constructor

(since C++11)

In traditional C++, if a derived class wants to reuse its base class's constructors, it must redeclare each of them and forward the arguments one by one to the base, which is tedious and error-prone. C++11 introduces inheriting constructors via the `using` keyword, letting a derived class inherit all of the base class's constructors at once:

```
#include <iostream>
class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
    }
    Base(int value) : Base() { // delegate Base() constructor
        value2 = value;
    }
};
class Subclass : public Base {
public:
    using Base::Base; // inheritance constructor
};
int main() {
    Subclass s(3);
    std::cout << s.value1 << std::endl;
    std::cout << s.value2 << std::endl;
}
```

Explicit virtual function overwrite

(since C++11)

In traditional C++, it is often prone to accidentally overloading virtual functions. E.g:

```
struct Base {
    virtual void foo();
};
```

```
struct SubClass: Base {
    void foo();
};
```

`SubClass::foo` may not be a programmer trying to overload a virtual function, just adding a function with the same name. Another possible scenario is that when the virtual function of the base class is deleted, the subclass owns the old function and no longer overloads the virtual function and turns it into a normal class method, which has catastrophic consequences.

C++11 introduces the two keywords `override` and `final` to prevent this from happening.

override

When overriding a virtual function, introducing the `override` keyword will explicitly tell the compiler to overload, and the compiler will check if the base function has such a virtual function with consistent function signature, otherwise it will not compile:

```
struct Base {
    virtual void foo(int);
};
struct SubClass: Base {
    virtual void foo(int) override; // legal
    virtual void foo(float) override; // illegal, no virtual function in super class
};
```

final

`final` is to prevent the class from being continued to inherit and to terminate the virtual function to continue to be overloaded.

```
struct Base {
    virtual void foo() final;
};
struct SubClass1 final: Base {
}; // legal

struct SubClass2 : SubClass1 {
}; // illegal, SubClass1 has final

struct SubClass3: Base {
    void foo(); // illegal, foo has final
};
```

Explicit delete default function

(since C++11)

In traditional C++, if the programmer does not provide it, the compiler will default to generating default constructors, copy constructs, assignment operators, and destructors for the object. Besides, C++ also defines operators such as `new delete` for all classes. This part of the function can be overridden when the programmer needs it.

This raises some requirements: the ability to accurately control the generation of default functions cannot be controlled. For example, when copying a class is prohibited, the copy constructor and the assignment operator must be declared as `private`. Trying to use these undefined functions will result in compilation or link errors, which is a very unconventional way.

Also, the default constructor generated by the compiler cannot exist at the same time as the user-defined constructor. If the user defines any constructor, the compiler will no longer generate the default constructor, but sometimes we want to have both constructors at the same time, which is awkward.

C++11 provides a solution to the above requirements, allowing explicit declarations to take or reject functions that come with the compiler. E.g:

```
class Magic {
    public:
    Magic() = default; // explicit let compiler use default constructor
    Magic& operator=(const Magic&) = delete; // explicit declare refuse constructor
    Magic(int magic_number);
}
```

Strongly typed enumerations

(since C++11)

In traditional C++, enumerated types are not type-safe, and enumerated types are treated as integers, which allows two completely different enumerated types to be directly compared (although the compiler gives the check, but not all), ** Even the enumeration value names of different enum types in the same namespace cannot be the same**, which is usually not what we want to see.

C++11 introduces an enumeration class and declares it using the syntax of `enum class`:

```
enum class new_enum : unsigned int {
    value1,
    value2,
    value3 = 100,
    value4 = 100
};
```

The enumeration thus defined implements type safety. First, it cannot be implicitly converted to an integer, nor can it be compared to integer numbers, and it is even less likely to compare enumerated values of different enumerated types. But if the values specified are the same between the same enumerated values, then you can compare:

```
if (new_enum::value3 == new_enum::value4) { // true
    std::cout << "new_enum::value3 == new_enum::value4" << std::endl;
}
```

In this syntax, the enumeration type is followed by a colon and a type keyword to specify the type of the enumeration value in the enumeration, which allows us to assign a value to the enumeration (int is used by default when not specified).

And we want to get the value of the enumeration value, we will have to explicitly type conversion, but we can overload the `<<` operator to output, you can collect the following code snippet:

```
#include <iostream>
template<typename T>
std::ostream& operator<<(
    typename std::enable_if<std::is_enum<T>::value,
        std::ostream>::type& stream, const T& e)
{
    return stream << static_cast<typename std::underlying_type<T>::type>(e);
}
```

At this point, the following code will be able to be compiled:

```
std::cout << new_enum::value3 << std::endl
```

2.7 Other Language Features

Inline variables

(since C++17)

Before C++17, a non-const static data member of a class had to be defined separately outside the class, and defining a global variable in a header would cause duplicate-definition link errors when the header was included by multiple translation units. C++17 introduces `inline` variables, which allow a variable (including a static data member) to be defined in a header without violating the One Definition Rule (ODR), even when included by multiple translation units:

```
struct Widget {
    static inline int count = 0; // C++17: define and initialize a static member in-class
```

```
};
inline int global_value = 42;    // safe to place in a header
```

This greatly simplifies writing header-only libraries.

Nested namespace definitions

(since C++17)

C++17 allows writing nested namespace definitions in a single line using `::`, instead of indenting level by level:

```
// before C++17
namespace A {
    namespace B {
        namespace C {
            int value;
        }
    }
}

// since C++17
namespace A::B::C {
    int value;
}
```

constexpr lambda

(since C++17)

Since C++17, a lambda expression that satisfies the requirements of a constant expression is implicitly `constexpr` (and may also be explicitly marked `constexpr`), so it can be evaluated at compile time:

```
constexpr auto add = [](int a, int b) { return a + b; };
static_assert(add(1, 2) == 3, "");

constexpr int result = add(3, 4); // evaluated at compile time, result == 7
```

Single-argument static_assert

(since C++17)

`static_assert` performs a compile-time assertion. Before C++17 it required a diagnostic message as its second argument; since C++17 that message is optional:

```
static_assert(sizeof(int) >= 2);                          // C++17: message optional
static_assert(sizeof(int) >= 2, "int must be >= 2 bytes"); // a message is still allowed
```

New aggregate rules

(since C++17)

C++17 relaxed the definition of an aggregate: an aggregate may now have public base classes (which must themselves be aggregates), and the base subobject can be brace-initialized along with the rest:

```
struct Base { int a; };
struct Derived : Base { int b; };

Derived d{{1}, 2}; // {a}, b - legal since C++17
```

Boolean logic metafunctions

(since C++17)

C++17 added `std::conjunction`, `std::disjunction`, and `std::negation` to `<type_traits>` for composing other type traits with logical AND/OR/NOT at compile time (and `conjunction`/`disjunction` short-circuit):

```
#include <type_traits>

template <typename T>
constexpr bool is_signed_integral =
    std::conjunction_v<std::is_integral<T>, std::is_signed<T>>;

static_assert(is_signed_integral<int>);
static_assert(!is_signed_integral<unsigned>);
static_assert(std::negation_v<std::is_floating_point<int>>);
```

`__has_include`

(since C++17)

C++17 standardized the preprocessor operator `__has_include`, which checks at compile time whether a header is available, enabling portable conditional inclusion:

```
#if __has_include(<optional>)
#  include <optional>
#  define HAS_OPTIONAL 1
#else
#  define HAS_OPTIONAL 0
#endif
```

This is very useful when supporting different standard-library versions, or providing a fallback when a feature might be missing.

Conclusion

This section introduces the enhancements to language usability in modern C++, which I believe are the most important features that almost everyone needs to know and use:

1. Auto type derivation
2. Scope for iteration
3. Initialization list
4. Variable parameter template

Exercises

1. Using structured binding, implement the following functions with just one line of function code:

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

template <typename Key, typename Value, typename F>
void update(std::map<Key, Value>& m, F foo) {
    // TODO:
}
int main() {
    std::map<std::string, long long int> m {
        {"a", 1},
        {"b", 2},
        {"c", 3}
    };
    update(m, [](std::string key) {
        return std::hash<std::string>{}(key);
    });
    for (auto&& [key, value] : m)
```

```
        std::cout << key << ":" << value << std::endl;
    }
```

2. Try to implement a function for calculating the mean with Fold Expression, allowing any arguments to be passed in.

   Refer to the answer see this.

Chapter 03: Language Runtime Enhancements

3.1 Lambda Expression

Lambda expressions are one of the most important features in modern C++, and Lambda expressions provide a feature like anonymous functions. Anonymous functions are used when a function is needed, but you don't want to use a name to call a function. There are many, many scenes like this. So anonymous functions are almost standard in modern programming languages.

Basics

(since C++11)

The basic syntax of a Lambda expression is as follows:

```
[capture list] (parameter list) mutable(optional) exception attribute -> return type {
// function body
}
```

The above grammar rules are well understood except for the things in `[capture list]`, except that the function name of the general function is omitted. The return value is in the form of a `->` (we have already mentioned this in the tail return type earlier in the previous section).

The so-called capture list can be understood as a type of parameter. The internal function body of a lambda expression cannot use variables outside the body of the function by default. At this time, the capture list can serve to transfer external data. According to the behavior passed, the capture list is also divided into the following types:

1. Value capture Similar to parameter passing, the value capture is based on the fact that the variable can be copied, except that the captured variable is copied when the lambda expression is created, not when it is called:

```
void lambda_value_capture() {
    int value = 1;
```

```
    auto copy_value = [value] {
        return value;
    };
    value = 100;
    auto stored_value = copy_value();
    std::cout << "stored_value = " << stored_value << std::endl;
    // At this moment, stored_value == 1, and value == 100.
    // Because copy_value has copied when its was created.
}
```

2. Reference capture Similar to a reference pass, the reference capture saves the reference and the value changes.

```
void lambda_reference_capture() {
    int value = 1;
    auto copy_value = [&value] {
        return value;
    };
    value = 100;
    auto stored_value = copy_value();
    std::cout << "stored_value = " << stored_value << std::endl;
    // At this moment, stored_value == 100, value == 100.
    // Because copy_value stores reference
}
```

3. Implicit capture Manually writing a capture list is sometimes very complicated. This mechanical work can be handled by the compiler. At this point, you can write a `&` or `=` to the compiler to declare the reference or value capture.

To summarize, capture provides the ability for lambda expressions to use external values. The four most common forms of capture lists can be:

- [] empty capture list
- [name1, name2, ...] captures a series of variables
- [&] reference capture, determine the reference capture list from the uses the in function body
- [=] value capture, determine the value capture list from the uses in the function body

4. Expression capture

> This section needs to understand the rvalue references and smart pointers that will be mentioned later.

The value captures and reference captures mentioned above are variables that have been declared in the outer scope, so these capture methods capture the lvalue and not capture the rvalue.

C++14 gives us the convenience of allowing the captured members to be initialized with arbitrary expressions, which allows the capture of rvalues. The type of the captured variable being declared is judged according to the expression, and the judgment is the same as using `auto`:

```
#include <iostream>
#include <memory>  // std::make_unique
#include <utility> // std::move

void lambda_expression_capture() {
    auto important = std::make_unique<int>(1);
    auto add = [v1 = 1, v2 = std::move(important)](int x, int y) -> int {
        return x+y+v1+(*v2);
    };
    std::cout << add(3,4) << std::endl;
}
```

In the above code, `important` is an exclusive pointer that cannot be caught by value capture using `=`. At this time we need to transfer it to the rvalue and initialize it in the expression.

Generic Lambda

(since C++14)

In the previous section, we mentioned that the `auto` keyword cannot be used in the parameter list because it would conflict with the functionality of the template. But lambda expressions are not regular functions, without further specification on the typed parameter list, lambda expressions cannot utilize templates. Fortunately, this trouble only exists in C++11, starting with C++14. The formal parameters of the lambda function can use the `auto` keyword to utilize template generics:

```
void lambda_generic() {
    auto generic = [](auto x, auto y) {
        return x+y;
    };

    std::cout << generic(1, 2) << std::endl;
    std::cout << generic(1.1, 2.2) << std::endl;
}
```

3.2 Function Object Wrapper

Although the features are part of the standard library and not found in runtime, it enhances the runtime capabilities of the C++ language. This part of the content is also very important, so put it here for the introduction.

`std::function`

(since C++11)

In C++, "things that can be called" come in many shapes — ordinary functions, function pointers, lambda expressions, and any object that overloads `operator()` — and they all have different types, which makes them hard to store and pass around uniformly. `std::function` exists precisely to solve this: it is a type-safe "container for callables" that can uniformly store, copy, and invoke any callable target, letting us handle "functions" as ordinary objects.

The essence of a Lambda expression is an object of a class type (called a closure type) that is similar to a function object type (called a closure object). When the capture list of a Lambda expression is empty, the closure object can also be converted to a function pointer value for delivery, for example:

```cpp
#include <iostream>
using foo = void(int);  // function pointer
void functional(foo f) {
    f(1);
}
int main() {
    auto f = [](int value) {
        std::cout << value << std::endl;
    };
    functional(f);  // call by function pointer
    f(1);           // call by lambda expression
    return 0;
}
```

The above code gives two different forms of invocation, one is to call Lambda as a function type, and the other is to directly call a Lambda expression. In C++11, these concepts are unified. The type of object that can be called is collectively called the callable type. This type is introduced by `std::function`.

C++11 `std::function` is a generic, polymorphic function wrapper whose instances can store, copy, and call any target entity that can be called. It is also an existing callable to C++. A type-safe package of entities (relatively, the call to a function pointer is not type-safe), in other words, a container of functions. When we have a container for functions, we can more easily handle functions and function pointers as objects. e.g:

```
#include <functional>
#include <iostream>

int foo(int para) {
    return para;
}

int main() {
    // std::function wraps a function that take int paremeter and returns int value
    std::function<int(int)> func = foo;

    int important = 10;
    std::function<int(int)> func2 = [&](int value) -> int {
        return 1+value+important;
    };
    std::cout << func(10) << std::endl;
    std::cout << func2(10) << std::endl;
}
```

`std::bind` and `std::placeholder`

(since C++11)

And `std::bind` is used to bind the parameters of the function call. It solves the requirement that we may not always be able to get all the parameters of a function at one time. Through this function, we can Part of the call parameters are bound to the function in advance to become a new object, and then complete the call after the parameters are complete. e.g:

```
int foo(int a, int b, int c) {
    return a + b + c;
}
int main() {
    // bind parameter 1, 2 on function foo,
    // and use std::placeholders::_1 as placeholder for the first parameter.
    auto bindFoo = std::bind(foo, std::placeholders::_1, 1, 2);
    // when call bindFoo, we only need one param left
    std::cout << bindFoo(1) << std::endl; // outputs 4
}
```

> **Tip:** Note the magic of the `auto` keyword. Sometimes we may not be familiar with the return type of a function, but we can circumvent this problem by using `auto`.

3.3 rvalue Reference

rvalue references are one of the important features introduced by C++11 that are synonymous with Lambda expressions. Its introduction solves a large number of historical issues in C++. Eliminating extra overhead such as `std::vector`, `std::string`, and making the function object container `std::function` possible.

lvalue, rvalue, prvalue, xvalue

To understand what the rvalue reference is all about, you must have a clear understanding of the lvalue and the rvalue.

> Strictly speaking, **a value category is a property of an *expression*, not of an object**: every C++ expression belongs to exactly one of the three primary value categories — lvalue, prvalue (pure rvalue), and xvalue (expiring value). The phrasings below ("an lvalue is a persistent object", "an rvalue is a temporary object") are intuitive approximations meant to help beginners; for the rigorous definitions, see cppreference: value category.

lvalue, left value, as the name implies, is the value to the left of the assignment symbol. To be precise, an lvalue is a persistent object that still exists after an expression (not necessarily an assignment expression).

Rvalue, right value, the value on the right refers to the temporary object that no longer exists after the expression ends.

In C++11, in order to introduce powerful rvalue references, the concept of rvalue values is further divided into: prvalue, and xvalue.

prvalue, pure rvalue, purely rvalue, either purely literal, such as `10`, `true`; either the result of the evaluation is equivalent to a literal or anonymous temporary object, for example `1+2`. Temporary variables returned by non-references, temporary variables generated by operation expressions, original literals, and Lambda expressions are all pure rvalue values.

Note that a literal (except a string literal) is a prvalue. However, a string literal is an lvalue with type `const char` array. Consider the following examples:

```
#include <type_traits>

int main() {
    // Correct. The type of "01234" is const char [6], so it is an lvalue
    const char (&left)[6] = "01234";

    // Assert success. It is a const char [6] indeed. Note that decltype(expr)
    // yields lvalue reference if expr is an lvalue and neither an unparenthesized
```

```
    // id-expression nor an unparenthesized class member access expression.
    static_assert(std::is_same<decltype("01234"), const char(&)[6]>::value, "");

    // Error. "01234" is an lvalue, which cannot be referenced by an rvalue reference
    // const char (&&right)[6] = "01234";
}
```

However, an array can be implicitly converted to a corresponding pointer.The result, if not an lvalue reference, is an rvalue (xvalue if the result is an rvalue reference, prvalue otherwise):

```
const char*   p    = "01234"; // Correct. "01234" is implicitly converted to const char*
const char*&& pr   = "01234"; // Correct. "01234" is implicitly converted to const char*, which is a
// const char*& pl = "01234"; // Error. There is no type const char* lvalue
```

xvalue, expiring value is the concept proposed by C++11 to introduce rvalue references (so in traditional C++, pure rvalue and rvalue are the same concepts), a value that is destroyed but can be moved.

It would be a little hard to understand the xvalue, let's look at the code like this:

```
std::vector<int> foo() {
    std::vector<int> temp = {1, 2, 3, 4};
    return temp;
}

std::vector<int> v = foo();
```

In such code, as far as the traditional understanding is concerned, the return value `temp` of the function `foo` is internally created and then assigned to `v`, whereas when `v` gets this object, the entire `temp` is copied. And then destroy `temp`, if this `temp` is very large, this will cause a lot of extra overhead (this is the problem that traditional C++ has been criticized for). In the last line, `v` is the lvalue, and the value returned by `foo()` is the rvalue (which is also a pure rvalue).

However, `v` can be caught by other variables, and the return value generated by `foo()` is used as a temporary value. Once copied by `v`, it will be destroyed immediately, and cannot be obtained or modified. The xvalue defines behavior in which temporary values can be identified while being able to be moved.

After C++11, the compiler did some work for us, where the lvalue `temp` is subjected to this implicit rvalue conversion, equivalent to `static_cast<std::vector<int> &&>(temp)`, where `v` here moves the value returned by `foo` locally. This is the move semantics we will mention later.

rvalue reference and lvalue reference

(since C++11)

To get a xvalue, you need to use the declaration of the rvalue reference: `T &&`, where `T` is the type. The statement of the rvalue reference extends the lifecycle of this temporary value, and as long as the variable is alive, the xvalue will continue to survive.

C++11 provides the `std::move` method to unconditionally convert lvalue parameters to rvalues. With it we can easily get a rvalue temporary object, for example:

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

void reference(std::string& str) {
    std::cout << "lvalue" << std::endl;
}
void reference(std::string&& str) {
    std::cout << "rvalue" << std::endl;
}

int main()
{
    std::string  lv1 = "string,";       // lv1 is a lvalue
    // std::string&& r1 = lv1;          // illegal, rvalue can't ref to lvalue
    std::string&& rv1 = std::move(lv1); // legal, std::move can convert lvalue to rvalue
    std::cout << rv1 << std::endl;      // string,

    const std::string& lv2 = lv1 + lv1; // legal, const lvalue reference can
                                        // extend temp variable's lifecycle
    // lv2 += "Test";                   // illegal, const ref can't be modified
    std::cout << lv2 << std::endl;      // string,string,

    std::string&& rv2 = lv1 + lv2;      // legal, rvalue ref extend lifecycle
    rv2 += "string";                    // legal, non-const reference can be modified
    std::cout << rv2 << std::endl;      // string,string,string,string

    reference(rv2);                     // output: lvalue

    return 0;
}
```

`rv2` refers to an rvalue, but since it is a reference, `rv2` is still an lvalue.

Note that there is a very interesting historical issue here, let's look at the following code:

```
#include <iostream>
```

```
int main() {
    // int &a = std::move(1); // illegal, non-const lvalue reference cannot ref rvalue
    const int &b = std::move(1); // legal, const lvalue reference can

    std::cout << b << std::endl;
}
```

The first question, why not allow non-constant references to bind to non-lvalues? This is because there is a logic error in this approach:

```
void increase(int & v) {
    v++;
}
void foo() {
    double s = 1;
    increase(s);
}
```

Since `int&` can't reference a parameter of type `double`, you must generate a temporary value to hold the value of `s`. Thus, when `increase()` modifies this temporary value, `s` itself is not modified after the call is completed.

The second question, why do constant references allow binding to non-lvalues? The reason is simple because Fortran needs it.

Move semantics

(since C++11)

Traditional C++ has designed the concept of copy/copy for class objects through copy constructors and assignment operators, but to implement the movement of resources, The caller must use the method of copying and then destructing first, otherwise, you need to implement the interface of the mobile object yourself. Imagine moving your home directly to your new home instead of copying everything (rebuy) to your new home. Throwing away (destroying) all the original things is a very anti-human thing.

Traditional C++ does not distinguish between the concepts of "mobile" and "copy", resulting in a large amount of data copying, wasting time and space. The appearance of rvalue references solves the confusion of these two concepts, for example:

```
#include <iostream>
class A {
public:
    int *pointer;
    A():pointer(new int(1)) {
```

```
        std::cout << "construct" << pointer << std::endl;
    }
    A(A& a):pointer(new int(*a.pointer)) {
        std::cout << "copy" << pointer << std::endl;
    } // meaningless object copy
    A(A&& a):pointer(a.pointer) {
        a.pointer = nullptr;
        std::cout << "move" << pointer << std::endl;
    }
    ~A(){
        std::cout << "destruct" << pointer << std::endl;
        delete pointer;
    }
};
// avoid compiler optimization
A return_rvalue(bool test) {
    A a,b;
    if(test) return a; // equal to static_cast<A&&>(a);
    else return b;     // equal to static_cast<A&&>(b);
}
int main() {
    A obj = return_rvalue(false);
    std::cout << "obj:" << std::endl;
    std::cout << obj.pointer << std::endl;
    std::cout << *obj.pointer << std::endl;
    return 0;
}
```

In the code above:

1. First construct two `A` objects inside `return_rvalue`, and get the output of the two constructors;
2. After the function returns, it will generate a xvalue, which is referenced by the moving structure of `A` (`A(A&&)`), thus extending the life cycle, and taking the pointer in the rvalue and saving it to `obj`. In the middle, the pointer to the xvalue is set to `nullptr`, which prevents the memory area from being destroyed.

This avoids meaningless copy constructs and enhances performance. Let's take a look at an example involving a standard library:

```
#include <iostream> // std::cout
#include <utility>  // std::move
#include <vector>   // std::vector
```

```
#include <string>   // std::string

int main() {

    std::string str = "Hello world.";
    std::vector<std::string> v;

    // use push_back(const T&), copy
    v.push_back(str);
    // "str: Hello world."
    std::cout << "str: " << str << std::endl;

    // use push_back(const T&&),
    // no copy the string will be moved to vector,
    // and therefore std::move can reduce copy cost
    v.push_back(std::move(str));
    // str is empty now
    std::cout << "str: " << str << std::endl;

    return 0;
}
```

Perfect forwarding

(since C++11)

As we mentioned earlier, the rvalue reference of a declaration is actually an lvalue. This creates problems for us to parameterize (pass):

```
#include <iostream>
#include <utility>
void reference(int& v) {
    std::cout << "lvalue reference" << std::endl;
}
void reference(int&& v) {
    std::cout << "rvalue reference" << std::endl;
}
template <typename T>
void pass(T&& v) {
    std::cout << "          normal param passing: ";
    reference(v);
}
```

```
int main() {
    std::cout << "rvalue pass:" << std::endl;
    pass(1);

    std::cout << "lvalue pass:" << std::endl;
    int l = 1;
    pass(l);

    return 0;
}
```

For `pass(1)`, although the value is the rvalue, since `v` is a reference, it is also an lvalue. Therefore `reference(v)` will call `reference(int&)` and output lvalue. For `pass(l)`, `l` is an lvalue, why is it successfully passed to `pass(T&&)`?

This is based on the **reference collapsing rule**: In traditional C++, we are not able to continue to reference a reference type. However, C++ has relaxed this practice with the advent of rvalue references, resulting in a reference collapse rule that allows us to reference references, both lvalue and rvalue. But follow the rules below:

Function parameter type	Argument parameter type	Post-derivation function parameter type
T&	lvalue ref	T&
T&	rvalue ref	T&
T&&	lvalue ref	T&
T&&	rvalue ref	T&&

Therefore, the use of `T&&` in a template function may not be able to make an rvalue reference, and when a lvalue is passed, a reference to this function will be derived as an lvalue. More precisely, **no matter what type of reference the template parameter is, the template parameter can be derived as a right reference type** if and only if the argument type is a right reference. This makes `v` successful delivery of lvalues.

Perfect forwarding is based on the above rules. The so-called perfect forwarding is to let us pass the parameters, Keep the original parameter type (lvalue reference keeps lvalue reference, rvalue reference keeps rvalue reference). To solve this problem, we should use `std::forward` to forward (pass) the parameters:

```
#include <iostream>
#include <utility>
void reference(int& v) {
    std::cout << "lvalue reference" << std::endl;
}
void reference(int&& v) {
```

```
    std::cout << "rvalue reference" << std::endl;
}
template <typename T>
void pass(T&& v) {
    std::cout << "          normal param passing: ";
    reference(v);
    std::cout << "       std::move param passing: ";
    reference(std::move(v));
    std::cout << "    std::forward param passing: ";
    reference(std::forward<T>(v));
    std::cout << "static_cast<T&&> param passing: ";
    reference(static_cast<T&&>(v));
}
int main() {
    std::cout << "rvalue pass:" << std::endl;
    pass(1);

    std::cout << "lvalue pass:" << std::endl;
    int l = 1;
    pass(l);

    return 0;
}
```

The outputs are:

```
rvalue pass:
          normal param passing: lvalue reference
       std::move param passing: rvalue reference
    std::forward param passing: rvalue reference
static_cast<T&&> param passing: rvalue reference
lvalue pass:
          normal param passing: lvalue reference
       std::move param passing: rvalue reference
    std::forward param passing: lvalue reference
static_cast<T&&> param passing: lvalue reference
```

Regardless of whether the pass parameter is an lvalue or an rvalue, the normal pass argument will forward the argument as an lvalue. So `std::move` will always accept an lvalue, which forwards the call to `reference(int&&)` to output the rvalue reference.

Only `std::forward` does not cause any extra copies and **perfectly forwards** (passes) the arguments of the function to other functions that are called internally.

`std::forward` is the same as `std::move`, and nothing is done. `std::move` simply converts the lvalue to the rvalue. `std::forward` is just a simple conversion of the parameters. From the point of view of the phenomenon, `std::forward<T>(v)` is the same as `static_cast<T&&>(v)`.

Readers may be curious as to why a statement can return values for two types of returns. Let's take a quick look at the concrete implementation of `std::forward`. `std::forward` contains two overloads:

```
template<typename _Tp>
constexpr _Tp&& forward(typename std::remove_reference<_Tp>::type& __t) noexcept
{ return static_cast<_Tp&&>(__t); }

template<typename _Tp>
constexpr _Tp&& forward(typename std::remove_reference<_Tp>::type&& __t) noexcept
{
    static_assert(!std::is_lvalue_reference<_Tp>::value, "template argument"
        " substituting _Tp is an lvalue reference type");
    return static_cast<_Tp&&>(__t);
}
```

In this implementation, the function of `std::remove_reference` is to eliminate references in the type. And `std::is_lvalue_reference` is used to check if the type derivation is correct, in the second implementation of `std::forward`. Check that the received value is indeed an lvalue, which in turn reflects the collapse rule.

When `std::forward` accepts an lvalue, `_Tp` is deduced to the lvalue, so the return value is the lvalue; and when it accepts the rvalue, `_Tp` is derived as an rvalue reference, and based on the collapse rule, the return value becomes the rvalue of `&& + &&`. It can be seen that the principle of `std::forward` is to make clever use of the differences in template type derivation.

At this point, we can answer the question: Why is `auto&&` the safest way to use looping statements? Because when `auto` is pushed to a different lvalue and rvalue reference, the collapsed combination with `&&` is perfectly forwarded.

Guaranteed copy elision

(since C++17)

Before C++17, when an object was initialized from a prvalue of the same type, the compiler *was permitted* (but not required) to omit the copy/move construction — this is known as copy elision. Because it was merely allowed, the object's type still had to have an accessible copy or move constructor, even though it would not actually be called.

C++17 makes copy elision **guaranteed** in this case: when an object is initialized from a prvalue of the same type, there is no temporary object at all — the object is constructed directly in its target location. As a result, returning a prvalue by value is well-formed even for a type that is neither copyable

nor movable:

```
struct NonMovable {
    NonMovable() = default;
    NonMovable(const NonMovable&) = delete; // non-copyable
    NonMovable(NonMovable&&) = delete;      // non-movable
};

NonMovable make() {
    return NonMovable{}; // OK in C++17: guaranteed elision, no copy/move needed
}

int main() {
    NonMovable n = make(); // constructed directly into n
}
```

The code above would not compile before C++17 (an accessible move or copy constructor was required), but is well-formed in C++17. This lets factory functions safely return non-movable types.

Conclusion

This chapter introduces the most important runtime enhancements in modern C++, and I believe that all the features mentioned in this section are worth knowing:

Lambda expression

1. Function object container std::function
2. rvalue reference

Further Readings

- Bjarne Stroustrup, The Design and Evolution of C++

Chapter 04: Containers

4.1 Linear Container

`std::array`

(since C++11)

When you see this container, you will have this problem:

1. Why introduce `std::array` instead of `std::vector` directly?
2. Already have a traditional array, why use `std::array`?

First, answer the first question. Unlike `std::vector`, the size of the `std::array` object is fixed. If the container size is fixed, then the `std::array` container can be used first. Also, since `std::vector` is automatically expanded, when a large amount of data is stored, and the container is deleted, The container does not automatically return the corresponding memory of the deleted element. In this case, you need to manually run `shrink_to_fit()` to release this part of the memory.

```
std::vector<int> v;
std::cout << "size:" << v.size() << std::endl;         // output 0
std::cout << "capacity:" << v.capacity() << std::endl; // output 0

// As you can see, the storage of std::vector is automatically managed and
// automatically expanded as needed.
// But if there is not enough space, you need to redistribute more memory,
// and reallocating memory is usually a performance-intensive operation.
v.push_back(1);
v.push_back(2);
v.push_back(3);
std::cout << "size:" << v.size() << std::endl;         // output 3
std::cout << "capacity:" << v.capacity() << std::endl; // output 4

// The auto-expansion logic here is very similar to Golang's slice.
v.push_back(4);
v.push_back(5);
std::cout << "size:" << v.size() << std::endl;         // output 5
std::cout << "capacity:" << v.capacity() << std::endl; // output 8

// As can be seen below, although the container empties the element,
// the memory of the emptied element is not returned.
v.clear();
std::cout << "size:" << v.size() << std::endl;         // output 0
std::cout << "capacity:" << v.capacity() << std::endl; // output 8

// Additional memory can be returned to the system via the shrink_to_fit() call
v.shrink_to_fit();
std::cout << "size:" << v.size() << std::endl;         // output 0
std::cout << "capacity:" << v.capacity() << std::endl; // output 0
```

The second problem is much simpler. Using `std::array` can make the code more "modern" and encapsulate some manipulation functions, such as getting the array size and checking if it is not empty,

and also using the standard friendly. Container algorithms in the library, such as `std::sort`.

Using `std::array` is as simple as specifying its type and size:

```
std::array<int, 4> arr = {1, 2, 3, 4};

arr.empty(); // check if container is empty
arr.size();  // return the size of the container

// iterator support
for (auto &i : arr)
{
    // ...
}

// use lambda expression for sort
std::sort(arr.begin(), arr.end(), [](int a, int b) {
    return b < a;
});

// array size must be constexpr
constexpr int len = 4;
std::array<int, len> arr = {1, 2, 3, 4};

// illegal, different than C-style array, std::array will not deduce to T*
// int *arr_p = arr;
```

When we started using `std::array`, it was inevitable that we would encounter a C-style compatible interface. There are three ways to do this:

```
void foo(int *p, int len) {
    return;
}

std::array<int, 4> arr = {1,2,3,4};

// C-stype parameter passing
// foo(arr, arr.size()); // illegal, cannot convert implicitly
foo(&arr[0], arr.size());
foo(arr.data(), arr.size());

// use `std::sort`
std::sort(arr.begin(), arr.end());
```

`std::forward_list`

(since C++11)

`std::forward_list` is a list container, and the usage is similar to `std::list`, so we don't spend a lot of time introducing it.

Need to know is that, unlike the implementation of the doubly linked list of `std::list`, `std::forward_list` is implemented using a singly linked list. Provides element insertion of `O(1)` complexity, does not support fast random access (this is also a feature of linked lists), It is also the only container in the standard library container that does not provide the `size()` method. Has a higher space utilization than `std::list` when bidirectional iteration is not required.

4.2 Unordered Container

(since C++11)

We are already familiar with the ordered container `std::map`/`std::set` in traditional C++. These elements are internally implemented by red-black trees. The average complexity of inserts and searches is `O(log(size))`. When inserting an element, the element size is compared according to the `<` operator and the element is determined to be the same. And select the appropriate location to insert into the container. When traversing the elements in this container, the output will be traversed one by one in the order of the `<` operator.

The elements in the unordered container are not sorted, and the internals is implemented by the Hash table. The average complexity of inserting and searching for elements is `O(constant)`, Significant performance gains can be achieved without concern for the order of the elements inside the container.

C++11 introduces two unordered containers: `std::unordered_map`/`std::unordered_multimap` and `std::unordered_set`/`std::unordered_multiset`.

Their usage is basically similar to the original `std::map`/`std::multimap`/`std::set`/`set::multiset` Since these containers are already familiar to us, we will not compare them one by one. Let's compare `std::map` and `std::unordered_map` directly:

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <map>

int main() {
    // initialized in same order
    std::unordered_map<int, std::string> u = {
        {1, "1"},
        {3, "3"},
```

```
        {2, "2"}
    };
    std::map<int, std::string> v = {
        {1, "1"},
        {3, "3"},
        {2, "2"}
    };

    // iterates in the same way
    std::cout << "std::unordered_map" << std::endl;
    for( const auto & n : u)
        std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";

    std::cout << std::endl;
    std::cout << "std::map" << std::endl;
    for( const auto & n : v)
        std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";
}
```

The final output is:

```
std::unordered_map
Key:[2] Value:[2]
Key:[3] Value:[3]
Key:[1] Value:[1]

std::map
Key:[1] Value:[1]
Key:[2] Value:[2]
Key:[3] Value:[3]
```

4.3 Tuples

Programmers who have known Python should be aware of the concept of tuples. Looking at the containers in traditional C++, except for `std::pair` there seems to be no ready-made structure to store different types of data (usually we will define the structure ourselves). But the flaw of `std::pair` is obvious, only two elements can be saved.

Basic Operations

(since C++11)

There are three core functions for the use of tuples:

1. `std::make_tuple`: construct tuple
2. `std::get`: Get the value of a position in the tuple
3. `std::tie`: tuple unpacking

```
#include <tuple>
#include <iostream>

auto get_student(int id) {
    if (id == 0)
        return std::make_tuple(3.8, 'A', "John");
    if (id == 1)
        return std::make_tuple(2.9, 'C', "Jack");
    if (id == 2)
        return std::make_tuple(1.7, 'D', "Ive");

    // it is not allowed to return 0 directly
    // return type is std::tuple<double, char, std::string>
    return std::make_tuple(0.0, 'D', "null");
}

int main() {
    auto student = get_student(0);
    std::cout << "ID: 0, "
              << "GPA: "   << std::get<0>(student) << ", "
              << "Grade: " << std::get<1>(student) << ", "
              << "Name: "  << std::get<2>(student) << '\n';

    double gpa;
    char grade;
    std::string name;

    // unpack tuples
    std::tie(gpa, grade, name) = get_student(1);
    std::cout << "ID: 1, "
              << "GPA: "   << gpa << ", "
              << "Grade: " << grade << ", "
              << "Name: "  << name << '\n';
}
```

`std::get` In addition to using constants to get tuple objects, C++14 adds usage types to get objects

in tuples:

```
std::tuple<std::string, double, double, int> t("123", 4.5, 6.7, 8);
std::cout << std::get<std::string>(t) << std::endl;
std::cout << std::get<double>(t) << std::endl; // illegal, runtime error
std::cout << std::get<3>(t) << std::endl;
```

Runtime Indexing

If you think about it, you might find the problem with the above code. `std::get<>` depends on a compile-time constant, so the following is not legal:

```
int index = 1;
std::get<index>(t);
```

So what do you do? The answer is to use `std::variant<>` (introduced by C++ 17) to provide type template parameters for `variant<>` You can have a `variant<>` to accommodate several types of variables provided (in other languages, such as Python/JavaScript, etc., as dynamic types):

```
#include <variant>
template <size_t n, typename... T>
constexpr std::variant<T...> _tuple_index(const std::tuple<T...>& tpl, size_t i) {
    if constexpr (n >= sizeof...(T))
        throw std::out_of_range(" .");
    if (i == n)
        return std::variant<T...>{ std::in_place_index<n>, std::get<n>(tpl) };
    return _tuple_index<(n < sizeof...(T)-1 ? n+1 : 0)>(tpl, i);
}
template <typename... T>
constexpr std::variant<T...> tuple_index(const std::tuple<T...>& tpl, size_t i) {
    return _tuple_index<0>(tpl, i);
}
template <typename T0, typename ... Ts>
std::ostream & operator<< (std::ostream & s, std::variant<T0, Ts...> const & v) {
    std::visit([&](auto && x){ s << x;}, v);
    return s;
}
```

So we can:

```
int i = 1;
std::cout << tuple_index(t, i) << std::endl;
```

Merge and Iteration

Another common requirement is to merge two tuples, which can be done with `std::tuple_cat`:

```
auto new_tuple = std::tuple_cat(get_student(1), std::move(t));
```

You can immediately see how quickly you can traverse a tuple? But we just introduced how to index a `tuple` by a very number at runtime, then the traversal becomes simpler. First, we need to know the length of a tuple, which can:

```
template <typename T>
auto tuple_len(T &tpl) {
    return std::tuple_size<T>::value;
}
```

This will iterate over the tuple:

```
for(int i = 0; i != tuple_len(new_tuple); ++i)
    // runtime indexing
    std::cout << tuple_index(new_tuple, i) << std::endl;
```

That said, traversing a tuple by "first implementing runtime indexing, then indexing element by element" works but is rather roundabout. If all you want is to apply the same operation to every element, the more direct and idiomatic way is to expand the indices at compile time with `std::index_sequence` (introduced in C++14). In C++17, it can be combined with a fold expression:

```
template <typename Func, typename Tuple, std::size_t... idx>
void iterate_impl(Func&& f, Tuple&& tpl, std::index_sequence<idx...>) {
    (f(std::get<idx>(std::forward<Tuple>(tpl))), ...);
}
template <typename Func, typename Tuple>
void iterate_tuple(Func&& f, Tuple&& tpl) {
    iterate_impl(std::forward<Func>(f), std::forward<Tuple>(tpl),
        std::make_index_sequence<std::tuple_size_v<std::remove_reference_t<Tuple>>>{});
}
```

In C++20, we can further drop the helper function by using a lambda that allows explicitly written template parameters:

```
template <typename Func, typename... Args>
void iterate_tuple(Func f, const std::tuple<Args...>& tpl) {
    [&]<std::size_t... idx>(std::index_sequence<idx...>) {
```

```
        (f(std::get<idx>(tpl)), ...);
    }(std::make_index_sequence<sizeof...(Args)>());
}
```

The call site is then very straightforward, and no runtime indexing is needed beforehand:

```
iterate_tuple([](const auto& v) { std::cout << v << ' '; }, new_tuple);
```

4.4 std::string_view and std::byte

std::string_view

(since C++17)

`std::string_view`, introduced in C++17, is a **non-owning, read-only** view over a sequence of characters; it holds just a pointer and a length. Declaring a parameter as `std::string_view` accepts both a `std::string` and a string literal, **without any copy or allocation**:

```
#include <string_view>

void print(std::string_view sv) {
    std::cout << sv << " (size = " << sv.size() << ")" << std::endl;
}

std::string_view sv = "hello, world";
print(sv.substr(0, 5)); // "hello"; substr does not allocate

std::string s = "from std::string";
print(s);               // implicit conversion, no copy
```

Mind its **lifetime**: a `string_view` does not own the underlying data, so the referenced character sequence must outlive the view, otherwise you get a dangling reference.

std::byte

(since C++17)

`std::byte` represents a single byte of **raw memory**. Unlike `char` or `unsigned char`, it is not an arithmetic type — the standard defines only bitwise operators for it, which prevents accidental arithmetic on raw bytes at the type level:

```
#include <cstddef>
```

```
std::byte b{0b0000'1100};        // 12
b <<= 2;                         // 48
b |= std::byte{0b0000'0001};     // 49
int v = std::to_integer<int>(b); // explicit conversion to an integer: 49
```

4.5 Associative container improvements

(since C++17)

C++17 added several more precise and more efficient operations to associative containers such as `std::map` / `std::unordered_map`:

- `try_emplace`: inserts only when the key is absent; when the key already exists it does **not** modify the existing value nor move from its arguments, which makes it better than `emplace` for "insert if not present".
- `insert_or_assign`: inserts a new element, or **overwrites** the value when the key already exists, returning whether an insertion happened.
- Node-based operations `extract` / `merge`: `extract` detaches a node from the container without copying or moving the element, and `merge` splices nodes from another container directly in.

```
#include <map>
#include <string>

std::map<int, std::string> m;
m.try_emplace(1, "one");
m.try_emplace(1, "uno");          // no effect, key 1 already present
m.insert_or_assign(1, "ONE");     // overwrites with "ONE"

std::map<int, std::string> other;
other.insert(m.extract(1));       // move node 1 into other, no element copy

std::map<int, std::string> more{{3, "three"}};
m.merge(more);                    // splice more's nodes into m
```

4.6 Polymorphic allocators `std::pmr`

(since C++17)

C++17 introduced the `std::pmr` namespace in `<memory_resource>`, providing polymorphic allocators based on a **memory resource**. It decouples the *where to allocate* policy from the container type: the same `pmr` container backed by different memory resources is still a single type, avoiding the type bloat caused by template allocators.

For instance, `std::pmr::monotonic_buffer_resource` can allocate from a pre-prepared buffer (even one on the stack) and frees everything only when the resource is destroyed — ideal for allocation-heavy workloads with a shared lifetime:

```
#include <array>
#include <cstddef>
#include <memory_resource>
#include <vector>

std::array<std::byte, 1024> buffer;
std::pmr::monotonic_buffer_resource pool{buffer.data(), buffer.size()};

std::pmr::vector<int> v{&pool}; // allocates from the stack buffer, not the heap
for (int i = 0; i < 5; ++i) v.push_back(i);
```

Conclusion

This chapter briefly introduces the new containers in modern C++. Their usage is similar to that of the existing containers in C++. It is relatively simple, and you can choose the containers you need to use according to the actual scene, to get better performance.

Although `std::tuple` is effective, the standard library provides limited functionality and there is no way to meet the requirements of runtime indexing and iteration. Fortunately, we have other methods that we can implement on our own.

Chapter 05: Smart Pointers and Memory Management

5.1 RAII and Reference Counting

Programmers who understand `Objective-C/Swift/JavaScript` should know the concept of reference counting. The reference count is counted to prevent memory leaks. The basic idea is to count the number of dynamically allocated objects. Whenever you add a reference to the same object, the reference count of the referenced object is incremented once. Each time a reference is deleted, the reference count is decremented by one. When the reference count of an object is reduced to zero, the pointed heap memory is automatically deleted.

In traditional C++, "remembering" to manually release resources is not always a best practice. Because we are likely to forget to release resources and lead to leakage. So the usual practice is that for an object, we apply for space when constructor, and free space when the destructor (called when leaving the scope). That is, we often say that the RAII resource acquisition is the initialization technology.

There are exceptions to everything, we always need to allocate objects on free storage. In traditional C++ we have to use `new` and `delete` to "remember" to release resources. C++11

introduces the concept of smart pointers, using the idea of reference counting so that programmers no longer need to care about manually releasing memory. These smart pointers include `std::shared_ptr`/`std::unique_ptr`/`std::weak_ptr`, which need to include the header file `<memory>`.

> Note: The reference count is not garbage collection. The reference count can recover the objects that are no longer used as soon as possible, and will not cause long waits during the recycling process. More clearly and indicate the life cycle of resources.

5.2 std::shared_ptr

(since C++11)

`std::shared_ptr` is a smart pointer that records how many `shared_ptr` points to an object, eliminating to call `delete`, which automatically deletes the object when the reference count becomes zero.

But not enough, because using `std::shared_ptr` still needs to be called with `new`, which makes the code a certain degree of asymmetry.

`std::make_shared` can be used to eliminate the explicit use of `new`, so `std::make_shared` will allocate the objects in the generated parameters. And return the `std::shared_ptr` pointer of this object type. For example:

```
#include <iostream>
#include <memory>
void foo(std::shared_ptr<int> i) {
    (*i)++;
}
int main() {
    // auto pointer = new int(10); // illegal, no direct assignment
    // Constructed a std::shared_ptr
    auto pointer = std::make_shared<int>(10);
    foo(pointer);
    std::cout << *pointer << std::endl; // 11
    // The shared_ptr will be destructed before leaving the scope
    return 0;
}
```

`std::shared_ptr` can get the raw pointer through the `get()` method and reduce the reference count by `reset()`. And see the reference count of an object by `use_count()`. E.g:

```
auto pointer = std::make_shared<int>(10);
auto pointer2 = pointer; // reference count+1
auto pointer3 = pointer; // reference count+1
int *p = pointer.get();  // no increase of reference count
```

```
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl;   // 3
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl; // 3
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl; // 3

pointer2.reset();
std::cout << "reset pointer2:" << std::endl;

std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl;   // 2
std::cout << "pointer2.use_count() = "
    << pointer2.use_count() << std::endl;                // pointer2 has reset, 0
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl; // 2

pointer3.reset();
std::cout << "reset pointer3:" << std::endl;

std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl;   // 1
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl; // 0
std::cout << "pointer3.use_count() = "
    << pointer3.use_count() << std::endl;                // pointer3 has reset, 0
```

5.3 std::unique_ptr

(since C++11)

std::unique_ptr is an exclusive smart pointer that prohibits other smart pointers from sharing the same object by copy construction or copy assignment, thus avoiding errors due to repeated destruction or freeing:

```
std::unique_ptr<int> pointer = std::make_unique<int>(10); // make_unique, from C++14
std::unique_ptr<int> pointer2 = pointer; // illegal
```

make_unique is not complicated. C++11 does not provide std::make_unique, which can be implemented by itself:

```
template<typename T, typename ...Args>
std::unique_ptr<T> make_unique( Args&& ...args ) {
  return std::unique_ptr<T>( new T( std::forward<Args>(args)... ) );
}
```

As for why it wasn't provided, Herb Sutter, chairman of the C++ Standards Committee, mentioned in his blog that it was because they were forgotten.

Since it is monopolized, in other words, it cannot be copied. However, we can use `std::move` to transfer it to other `unique_ptr`, for example:

```
#include <iostream>
#include <memory>

struct Foo {
    Foo()      { std::cout << "Foo::Foo" << std::endl;  }
    ~Foo()     { std::cout << "Foo::~Foo" << std::endl; }
    void foo() { std::cout << "Foo::foo" << std::endl;  }
};

void f(const Foo &) {
    std::cout << "f(const Foo&)" << std::endl;
}

int main() {
    std::unique_ptr<Foo> p1(std::make_unique<Foo>());

    // p1 is not empty, prints
    if (p1) p1->foo();
    {
        std::unique_ptr<Foo> p2(std::move(p1));

        // p2 is not empty, prints
        f(*p2);

        // p2 is not empty, prints
        if(p2) p2->foo();

        // p1 is empty, no prints
        if(p1) p1->foo();

        p1 = std::move(p2);

        // p2 is empty, no prints
        if(p2) p2->foo();
        std::cout << "p2 was destroyed" << std::endl;
    }
    // p1 is not empty, prints
    if (p1) p1->foo();
```

```
    // Foo instance will be destroyed when leaving the scope
}
```

5.4 `std::weak_ptr`

(since C++11)

If you think about `std::shared_ptr` carefully, you will still find that there is still a problem that resources cannot be released. Look at the following example:

```cpp
#include <iostream>
#include <memory>

class A;
class B;

class A {
public:
    std::shared_ptr<B> pointer;
    ~A() {
        std::cout << "A was destroyed" << std::endl;
    }
};
class B {
public:
    std::shared_ptr<A> pointer;
    ~B() {
        std::cout << "B was destroyed" << std::endl;
    }
};
int main() {
    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();
    a->pointer = b;
    b->pointer = a;

    return 0;
}
```

The result is that A and B will not be destroyed. This is because the pointer inside a, b also references `a, b`, which makes the reference count of `a, b` becomes 2, leaving the scope. When the `a, b` smart pointer is destructed, it can only cause the reference count of this area to be decremented by

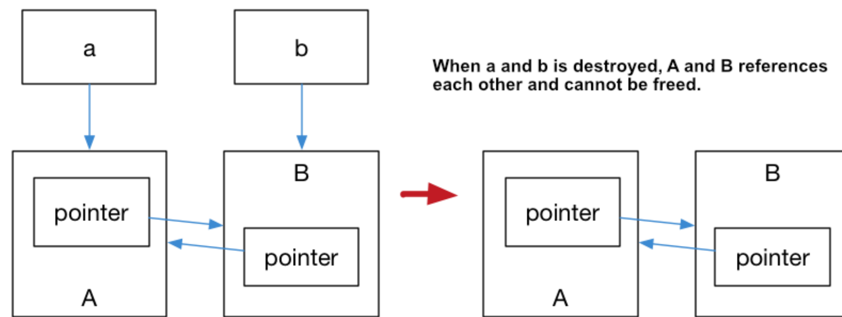


Figure 2: Figure 5.1

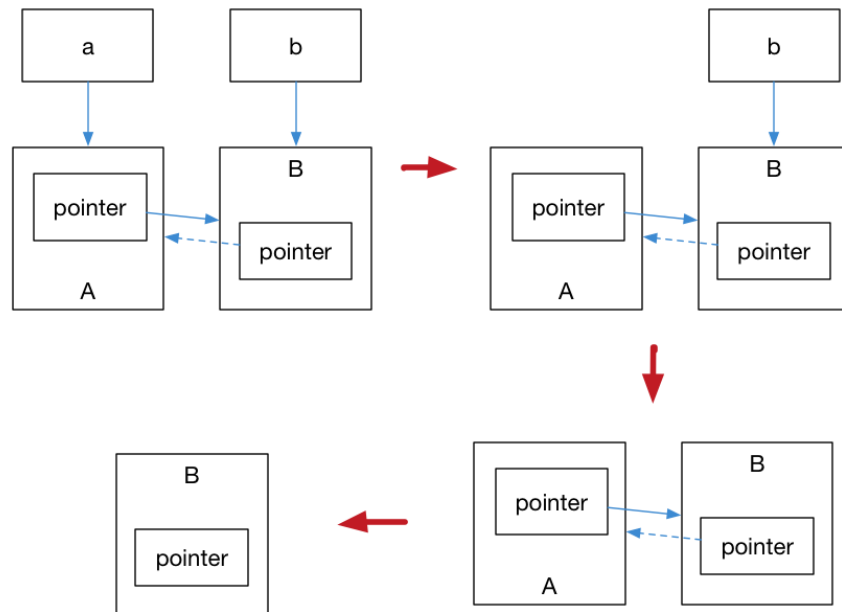


Figure 3: Figure 5.2

one. This causes the memory area reference count pointed to by the `a, b` object to be non-zero, but the external has no way to find this area, it also caused a memory leak, as shown in Figure 5.1:

The solution to this problem is to use the weak reference pointer `std::weak_ptr`, which is a weak reference (compared to `std::shared_ptr` is a strong reference). A weak reference does not cause an increase in the reference count. When a weak reference is used, the final release process is shown in Figure 5.2:

In the above figure, only B is left in the last step, and B does not have any smart pointers to reference it, so this memory resource will also be released.

`std::weak_ptr` has no implemented `*` and `->` operators, therefore it cannot operate on resources. `std::weak_ptr` allows us to check if a `std::shared_ptr` exists or not. The `expired()` method of a `std::weak_ptr` returns `false` when the resource is not released; Otherwise, it returns `true`. Furthermore, it can also be used for the purpose of obtaining `std::shared_ptr`, which points to the original object. The `lock()` method returns a `std::shared_ptr` to the original object when the resource is not released, or `nullptr` otherwise.

Conclusion

The technology of smart pointers is not novel. It is a common technology in many languages. Modern C++ introduces this technology, which eliminates the abuse of `new`/`delete` to a certain extent. It is a more mature technology. Programming paradigm.

Further Readings

- Why does C++11 have `make_shared` but not `make_unique`

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:

Symbol	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.

Symbol	Description
`\`	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:

Symbol	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 `foooooo`. `o{1,}` is equivalent to `o+`. `o{0,}` is equivalent to `o*`.
`{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

(since C++11)

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 the 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:

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

int main() {
    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: Parallelism and Concurrency

7.1 Basic of Parallelism

(since C++11)

Before C++11, the C++ standard library had no notion of threads at all: writing concurrent code meant reaching for platform-specific APIs such as POSIX `pthread` or the Windows thread API, which made that code hard to port. C++11 brought threading into the language standard for the first time, so the same concurrent code now builds across platforms.

`std::thread` is used to create an execution thread instance, so it is the basis for all concurrent programming. It needs to include the `<thread>` header file when using it. It provides a number of basic thread operations, such as `get_id()` to get the thread ID of the thread being created, use `join()` to join a thread, etc., for example:

```
#include <iostream>
#include <thread>

int main() {
    std::thread t([](){
        std::cout << "hello world." << std::endl;
    });
    t.join();
    return 0;
}
```

7.2 Mutex and Critical Section

(since C++11)

We have already learned the basics of concurrency technology in the operating system, or the database, and `mutex` is one of the cores. C++11 introduces a class related to `mutex`, with all related functions in the `<mutex>` header file.

`std::mutex` is the most basic mutex class in C++11, and a mutex can be created by constructing a `std::mutex` object. It can be locked by its member function `lock()`, and `unlock()` can be unlocked. But in the process of actually writing the code, it is best not to directly call the member function, Because calling member functions, you need to call `unlock()` at the exit of each critical section, and of course, exceptions. At this time, C++11 also provides a template class `std::lock_guard` for the RAII mechanism for the mutex.

RAII guarantees the exceptional security of the code while keeping the simplicity of the code.

```
#include <iostream>
```

```
#include <mutex>
#include <thread>

int v = 1;

void critical_section(int change_v) {
    static std::mutex mtx;
    std::lock_guard<std::mutex> lock(mtx);

    // execute contention works
    v = change_v;

    // mtx will be released after leaving the scope
}

int main() {
    std::thread t1(critical_section, 2), t2(critical_section, 3);
    t1.join();
    t2.join();

    std::cout << v << std::endl;
    return 0;
}
```

Because C++ guarantees that all stack objects will be destroyed at the end of the declaration period, such code is also extremely safe. Whether `critical_section()` returns normally or if an exception is thrown in the middle, a stack unwinding is thrown, and `unlock()` is automatically called.

> An exception is thrown and not caught (it is implementation-defined whether any stack unwinding is done in this case).

`std::unique_lock` is more flexible than `std::lock_guard`. Objects of `std::unique_lock` manage the locking and unlocking operations on the `mutex` object with exclusive ownership (no other `unique_lock` objects owning the ownership of a `mutex` object). So in concurrent programming, it is recommended to use `std::unique_lock`.

`std::lock_guard` cannot explicitly call `lock` and `unlock`, and `std::unique_lock` can be called anywhere after the declaration. It can reduce the scope of the lock and provide higher concurrency.

If you use the condition variable `std::condition_variable::wait` you must use `std::unique_lock` as a parameter.

For instance:

```
#include <iostream>
#include <mutex>
#include <thread>

int v = 1;

void critical_section(int change_v) {
    static std::mutex mtx;
    std::unique_lock<std::mutex> lock(mtx);
    // do contention operations
    v = change_v;
    std::cout << v << std::endl;
    // release the lock
    lock.unlock();

    // during this period,
    // others are allowed to acquire v

    // start another group of contention operations
    // lock again
    lock.lock();
    v += 1;
    std::cout << v << std::endl;
}

int main() {
    std::thread t1(critical_section, 2), t2(critical_section, 3);
    t1.join();
    t2.join();
    return 0;
}
```

7.3 Future

(since C++11)

The Future is represented by `std::future`, which provides a way to access the results of asynchronous operations. This sentence is very difficult to understand. To understand this feature, we need to understand the multi-threaded behavior before C++11.

Imagine if our main thread A wants to open a new thread B to perform some of our expected tasks and return me a result. At this time, thread A may be busy with other things and have no time to take into account the results of B. So we naturally hope to get the result of thread B at a certain time.

Before the introduction of `std::future` in C++11, the usual practice is: Create a thread A, start task B in thread A, send an event when it is ready, and save the result in a global variable. The main function thread A is doing other things. When the result is needed, a thread is called to wait for the function to get the result of the execution.

The `std::future` provided by C++11 simplifies this process and can be used to get the results of asynchronous tasks. Naturally, we can easily imagine it as a simple means of thread synchronization, namely the barrier.

To see an example, we use extra `std::packaged_task`, which can be used to wrap any target that can be called for asynchronous calls. For example:

```cpp
#include <iostream>
#include <thread>
#include <future>

int main() {
    // pack a lambda expression that returns 7 into a std::packaged_task
    std::packaged_task<int()> task([](){return 7;});
    // get the future of task
    std::future<int> result = task.get_future();    // run task in a thread
    std::thread(std::move(task)).detach();
    std::cout << "waiting...";
    result.wait(); // block until future has arrived
    // output result
    std::cout << "done!" << std:: endl << "future result is "
              << result.get() << std::endl;
    return 0;
}
```

After encapsulating the target to be called, you can use `get_future()` to get a `std::future` object to implement thread synchronization later.

7.4 Condition Variable

(since C++11)

The condition variable `std::condition_variable` was born to solve the deadlock and was introduced when the mutex operation was not enough. For example, a thread may need to wait for a condition to be true to continue execution. A dead wait loop can cause all other threads to fail to enter the critical section so that when the condition is true, a deadlock occurs. Therefore, the `condition_variable` object is created primarily to wake up the waiting thread and avoid deadlocks. `notify_one()` of `std::condition_variable` is used to wake up a thread; `notify_all()` is to notify all threads. Below is an example of a producer and consumer model:

```cpp
#include <queue>
#include <chrono>
#include <mutex>
#include <thread>
#include <iostream>
#include <condition_variable>


int main() {
    std::queue<int> produced_nums;
    std::mutex mtx;
    std::condition_variable cv;
    bool notified = false;  // notification sign

    auto producer = [&]() {
        for (int i = 0; ; i++) {
            std::this_thread::sleep_for(std::chrono::milliseconds(500));
            std::unique_lock<std::mutex> lock(mtx);
            std::cout << "producing " << i << std::endl;
            produced_nums.push(i);
            notified = true;
            cv.notify_all();
        }
    };
    auto consumer = [&]() {
        while (true) {
            std::unique_lock<std::mutex> lock(mtx);
            while (!notified) {  // avoid spurious wakeup
                cv.wait(lock);
            }

            // temporal unlock to allow producer produces more rather than
            // let consumer hold the lock until its consumed.
            lock.unlock();
            // consumer is slower
            std::this_thread::sleep_for(std::chrono::milliseconds(1000));
            lock.lock();
            if (!produced_nums.empty()) {
                std::cout << "consuming " << produced_nums.front() << std::endl;
                produced_nums.pop();
            }
            notified = false;
```

```
        }
    };

    std::thread p(producer);
    std::thread cs[2];
    for (int i = 0; i < 2; ++i) {
        cs[i] = std::thread(consumer);
    }
    p.join();
    for (int i = 0; i < 2; ++i) {
        cs[i].join();
    }
    return 0;
}
```

It is worth mentioning that although we can use `notify_one()` in the producer, it is not recommended to use it here. Because in the case of multiple consumers, our consumer implementation simply gives up the lock holding, which makes it possible for other consumers to compete for this lock, to better utilize the concurrency between multiple consumers. Having said that, but in fact because of the exclusivity of `std::mutex`, We simply can't expect multiple consumers to be able to produce content in a parallel consumer queue, and we still need a more granular approach.

7.5 Atomic Operation and Memory Model

(since C++11)

Careful readers may be tempted by the fact that the example of the producer-consumer model in the previous section may have compiler optimizations that cause program errors. For example, the compiler may have optimizations for the variable `notified`, such as the value of a register. As a result, the consumer thread can never observe the change of this value. This is a good question. To explain this problem, we need to further discuss the concept of the memory model introduced from C++11. Let's first look at a question. What is the output of the following code?

```
#include <thread>
#include <iostream>

int main() {
    int a = 0;
    volatile int flag = 0;

    std::thread t1([&]() {
        while (flag != 1);
```

```
        int b = a;
        std::cout << "b = " << b << std::endl;
    });

    std::thread t2([&]() {
        a = 5;
        flag = 1;
    });

    t1.join();
    t2.join();
    return 0;
}
```

Intuitively, it seems that `a = 5;` in `t2` always executes before `flag = 1;` and `while (flag != 1)` in `t1`. It looks like there is a guarantee the line `std ::cout << "b = " << b << std::endl;` will not be executed before the mark is changed. Logically, it seems that the value of `b` should be equal to 5. But the actual situation is much more complicated than this, or the code itself is undefined behavior because, for `a` and `flag`, they are read and written in two parallel threads. There has been competition. Also, even if we ignore competing for reading and writing, it is still possible to receive out-of-order execution of the CPU and the impact of the compiler on the rearrangement of instructions. Cause `a = 5` to occur after `flag = 1`. Thus `b` may output 0.

Atomic Operation

`std::mutex` can solve the problem of concurrent read and write, but the mutex is an operating system-level function. This is because the implementation of a mutex usually contains two basic principles:

1. Provide automatic state transition between threads, that is, "lock" state
2. Ensure that the memory of the manipulated variable is isolated from the critical section during the mutex operation

This is a very strong set of synchronization conditions, in other words when it is finally compiled into a CPU instruction, it will behave like a lot of instructions (we will look at how to implement a simple mutex later). This seems too harsh for a variable that requires only atomic operations (no intermediate state).

The research on synchronization conditions has a very long history, and we will not go into details here. Readers should understand that under the modern CPU architecture, atomic operations at the CPU instruction level are provided. Therefore, the `std::atomic` template is introduced in C++11 for the topic of multi-threaded shared variable reading and writing, which enables us to instantiate atomic types,

and minimize an atomic read or write operation from a set of instructions to a single CPU instruction. E.g:

```
std::atomic<int> counter;
```

And provides basic numeric member functions for atomic types of integers or floating-point numbers, for example, Including `fetch_add`, `fetch_sub`, etc., and the corresponding `+`, `-` version is provided by overload. For example, the following example:

```
#include <atomic>
#include <thread>
#include <iostream>

std::atomic<int> count = {0};

int main() {
    std::thread t1([](){
        count.fetch_add(1);
    });
    std::thread t2([](){
        count++;        // identical to fetch_add
        count += 1;     // identical to fetch_add
    });
    t1.join();
    t2.join();
    std::cout << count << std::endl;
    return 0;
}
```

Of course, not all types provide atomic operations because the feasibility of atomic operations depends on the architecture of the CPU and whether the type structure being instantiated satisfies the memory alignment requirements of the architecture, so we can always pass `std::atomic<T>::is_lock_free` to check if the atom type needs to support atomic operations, for example:

```
#include <atomic>
#include <iostream>

struct A {
    float x;
    int y;
    long long z;
};
```

```cpp
int main() {
    std::atomic<A> a;
    std::cout << std::boolalpha << a.is_lock_free() << std::endl;
    return 0;
}
```

Consistency Model

Multiple threads executing in parallel, discussed at some macro level, can be roughly considered a distributed system. In a distributed system, any communication or even local operation takes a certain amount of time, and even unreliable communication occurs.

If we force the operation of a variable `v` between multiple threads to be atomic, that is, any thread after the operation of `v` Other threads can **synchronize** to perceive changes in `v`, for the variable `v`, which appears as a sequential execution of the program, it does not have any efficiency gains due to the introduction of multithreading. Is there any way to accelerate this properly? The answer is to weaken the synchronization conditions between processes in atomic operations.

In principle, each thread can correspond to a cluster node, and communication between threads is almost equivalent to communication between cluster nodes. Weakening the synchronization conditions between processes, usually we will consider four different consistency models:

1. Linear consistency: Also known as strong consistency or atomic consistency. It requires that any read operation can read the most recent write of a certain data, and the order of operation of all threads is consistent with the order under the global clock.

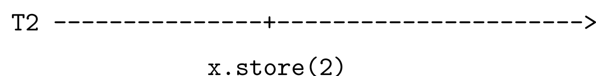
```
        x.store(1)      x.load()
T1 ---------+----------------+------>


T2 -------------------+------------->
                  x.store(2)
```

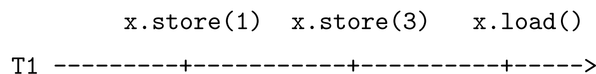
   In this case, thread `T1`, `T2` is twice atomic to `x`, and `x.store(1)` is strictly before `x.store(2)`. `x.store(2)` strictly occurs before `x.load()`. It is worth mentioning that linear consistency requirements for global clocks are difficult to achieve, which is why people continue to study other consistent algorithms under this weaker consistency.

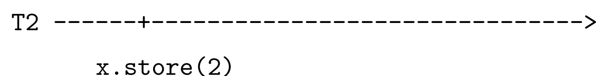
2. Sequential consistency: It is also required that any read operation can read the last data written by the data, but it is not required to be consistent with the order of the global clock.

```
        x.store(1)  x.store(3)   x.load()
T1 ---------+-----------+----------+----->
```



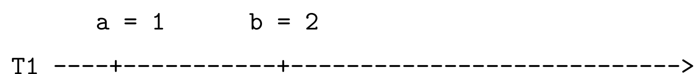
or

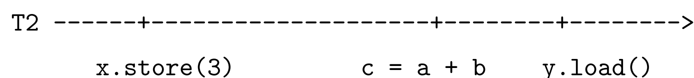




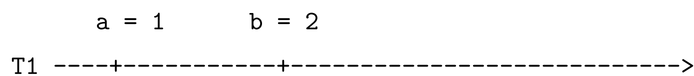
Under the order consistency requirement, `x.load()` must read the last written data, so `x.store(2)` and `x.store(1)` do not have any guarantees, as long as `x.store(2)` of `T2` occurs before `x.store(3)`.

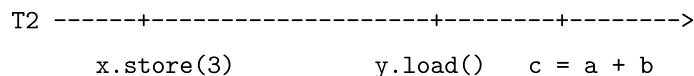
3. Causal consistency: its requirements are further reduced, only the sequence of causal operations is guaranteed, and the order of non-causal operations is not required.



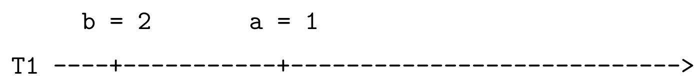


or





or



```
T2 ------+--------------------+--------+-------->
      y.load()            c = a + b  x.store(3)
```

The three examples given above are all causal consistent because, in the whole process, only `c` has a dependency on `a` and `b`, and `x` and `y` are not related in this example. (But in actual situations we need more detailed information to determine that `x` is not related to `y`)

4. Final Consistency: It is the weakest consistency requirement. It only guarantees that an operation will be observed at a certain point in the future, but does not require the observed time. So we can even strengthen this condition a bit, for example, to specify that the time observed for an operation is always bounded. Of course, this is no longer within our discussion.

```
    x.store(3)  x.store(4)
T1 ----+-----------+------------------------------------------->



T2 ---------+------------+--------------------+--------+-------->
         x.read()      x.read()           x.read()   x.read()
```

In the above case, if we assume that the initial value of x is 0, then the four times '`x.read()` in `T2` may be but not limited to the following:

```
3 4 4 4 // The write operation of x was quickly observed
0 3 3 4 // There is a delay in the observed time of the x write operation
0 0 0 4 // The last read read the final value of x,
        // but the previous changes were not observed.
0 0 0 0 // The write operation of x is not observed in the current time period,
        // but the situation that x is 4 can be observed
        // at some point in the future.
```

Memory Orders

To achieve the ultimate performance and achieve consistency of various strength requirements, C++11 defines six different memory sequences for atomic operations. The option `std::memory_order` expresses four synchronization models between multiple threads:

1. Relaxed model: Under this model, atomic operations within a single thread are executed sequentially, and instruction reordering is not allowed, but the order of atomic operations between different threads is arbitrary. The type is specified by `std::memory_order_relaxed`. Let's look at an example:

```
std::atomic<int> counter = {0};
std::vector<std::thread> vt;
for (int i = 0; i < 100; ++i) {
```

```
        vt.emplace_back([&](){
            counter.fetch_add(1, std::memory_order_relaxed);
        });
    }

    for (auto& t : vt) {
        t.join();
    }
    std::cout << "current counter:" << counter << std::endl;
```

2. Release/consumption model: In this model, we begin to limit the order of operations between processes. If a thread needs to modify a value, but another thread will have a dependency on that operation of the value, that is, the latter depends on the former. Specifically, thread A has completed three writes to `x`, and thread `B` relies only on the third `x` write operation, regardless of the first two write behaviors of `x`, then `A` When active `x.release()` (ie using `std::memory_order_release`), the option `std::memory_order_consume` ensures that `B` observes `A` when calling `x.load()` Three writes to `x`. Let's look at an example:

```
// initialize as nullptr to prevent consumer load a dangling pointer
std::atomic<int*> ptr(nullptr);
int v;
std::thread producer([&]() {
    int* p = new int(42);
    v = 1024;
    ptr.store(p, std::memory_order_release);
});
std::thread consumer([&]() {
    int* p;
    while(!(p = ptr.load(std::memory_order_consume)));

    std::cout << "p: " << *p << std::endl;
    std::cout << "v: " << v << std::endl;
});
producer.join();
consumer.join();
```

3. Release/Acquire model: Under this model, we can further tighten the order of atomic operations between different threads, specifying the timing between releasing `std::memory_order_release` and getting `std::memory_order_acquire`. **All** write operations before the release operation is visible to any other thread, i.e., happens before.

   As you can see, `std::memory_order_release` ensures that a write before a release does not occur after the release operation, which is a **backward barrier**, and `std::memory_order_acquire` ensures that a subsequent read or write after a acquire does not occur before the acquire operation,

which is a **forward barrier**. For the `std::memory_order_acq_rel` option, combines the characteristics of the two barriers and determines a unique memory barrier, such that reads and writes of the current thread will not be rearranged across the barrier.

Let's check an example:

```
std::vector<int> v;
std::atomic<int> flag = {0};
std::thread release([&]() {
    v.push_back(42);
    flag.store(1, std::memory_order_release);
});
std::thread acqrel([&]() {
    int expected = 1; // must before compare_exchange_strong
    while(!flag.compare_exchange_strong(expected, 2, std::memory_order_acq_rel))
        expected = 1; // must after compare_exchange_strong
    // flag has changed to 2
});
std::thread acquire([&]() {
    while(flag.load(std::memory_order_acquire) < 2);

    std::cout << v.at(0) << std::endl; // must be 42
});
release.join();
acqrel.join();
acquire.join();
```

In this case we used `compare_exchange_strong`, which is the Compare-and-swap primitive, which has a weaker version, `compare_exchange_weak`, which allows a failure to be returned even if the exchange is successful. The reason is due to a false failure on some platforms, specifically when the CPU performs a context switch, another thread loads the same address to produce an inconsistency. In addition, the performance of `compare_exchange_strong` may be slightly worse than `compare_exchange_weak`. However, in most cases, `compare_exchange_weak` is discouraged due to the complexity of its usage.

4. Sequential Consistent Model: Under this model, atomic operations satisfy sequence consistency, which in turn can cause performance loss. It can be specified explicitly by `std::memory_order_seq_cst`. Let's look at a final example:

```
std::atomic<int> counter = {0};
std::vector<std::thread> vt;
for (int i = 0; i < 100; ++i) {
    vt.emplace_back([&](){
        counter.fetch_add(1, std::memory_order_seq_cst);
```

```
    });
}

for (auto& t : vt) {
    t.join();
}
std::cout << "current counter:" << counter << std::endl;
```

This example is essentially the same as the first loose model example. Just change the memory order of the atomic operation to `memory_order_seq_cst`. Interested readers can write their own programs to measure the performance difference caused by these two different memory sequences.

Conclusion

The C++11 language layer provides support for concurrent programming. This section briefly introduces `std::thread`/`std::mutex`/`std::future`, an important tool that can't be avoided in concurrent programming. In addition, we also introduced the "memory model" as one of the most important features of C++11. They provide a critical foundation for standardized high-performance computing for C++.

Exercises

1. Write a simple thread pool that provides the following features:

```
ThreadPool p(4); // specify four work thread

// enqueue a task, and return a std::future
auto f = pool.enqueue([](int life) {
    return meaning;
}, 42);

// fetch result from future
std::cout << f.get() << std::endl;
```

2. Use `std::atomic<bool>` to implement a mutex.

Further Readings

- C++ Concurrency in Action
- Thread document
- Herlihy, M. P., & Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3), 463–492. https://doi.org/10.1145/78969.78972

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, it was one of the first libraries to be launched by boost and eventually merged into the C++ standard in C++17. Its entire contents live in the `<filesystem>` header under the `std::filesystem` namespace, for which a short alias is conventionally introduced:

```cpp
#include <filesystem>
namespace fs = std::filesystem;
```

On some older toolchains (e.g. GCC before version 8), using the filesystem library also required linking against `-lstdc++fs` (or LLVM's `-lc++fs`); modern compilers no longer need this.

8.1 The path `std::filesystem::path`

`std::filesystem::path` is the heart of the library. It represents a file path in a portable way and hides the differences between operating systems in their path separators (such as `/` and `\`). A `path` can be constructed from a string and composed with `operator/`:

```cpp
fs::path p = "/usr/local";
p /= "bin";                 // p is now /usr/local/bin
fs::path q = p / "clang"; // composes without modifying p
```

It is worth emphasizing that a `path` is only a **syntactic** representation of a path: constructing one neither touches the disk nor requires the path to actually exist. It provides a set of member functions for decomposing a path:

```cpp
fs::path p = "/usr/local/hello.txt";
p.filename();     // "hello.txt"
p.stem();         // "hello"
p.extension();    // ".txt"
p.parent_path(); // "/usr/local"
```

8.2 Querying file status

The library provides a set of **non-member functions** for querying the actual file a path refers to; each takes a `path` argument:

```cpp
fs::exists(p);             // does the path exist
fs::is_regular_file(p);    // is it a regular file
```

```
fs::is_directory(p);      // is it a directory
fs::file_size(p);         // file size in bytes
fs::last_write_time(p);   // last modification time
```

Note that these operations, which genuinely access the file system, **throw a `std::filesystem::filesystem_error`** on failure (e.g. the path does not exist or permission is denied). For nearly every such function the library provides an overload taking a `std::error_code&`, to report errors without exceptions:

```
std::error_code ec;
auto size = fs::file_size(p, ec); // does not throw on error; writes to ec
if (ec) {
    std::cout << "cannot get size: " << ec.message() << std::endl;
}
```

8.3 Iterating directories

`std::filesystem::directory_iterator` iterates over the immediate entries of a directory, while `recursive_directory_iterator` walks the entire directory tree. Both can be used directly in a range-based for loop, yielding `directory_entry` elements:

```
for (const auto& entry : fs::directory_iterator(dir)) {
    std::cout << entry.path() << std::endl;
}

// recursively walk the whole tree
for (const auto& entry : fs::recursive_directory_iterator(dir)) {
    if (entry.is_regular_file())
        std::cout << entry.path() << " (" << entry.file_size() << ")\n";
}
```

A `directory_entry` caches the status of the file it refers to, so member functions such as `entry.is_regular_file()` and `entry.file_size()` are often more efficient than calling the non-member functions again on its path.

8.4 Creating, copying, and removing

The library also offers a set of operations that modify the file system:

```
fs::create_directories(p / "a" / "b"); // create directories recursively
fs::copy_file(src, dst);                // copy a single file
fs::copy(src, dst, fs::copy_options::recursive); // copy a directory recursively
fs::rename(old_path, new_path);         // rename / move
```

```
fs::remove(p);                      // remove a single file or empty directory
fs::remove_all(p);                  // remove recursively, returns the count removed
```

8.5 A complete example

The following example combines the operations above. To keep it self-contained and repeatable, we create a dedicated working directory under the system temporary directory and clean it up at the end:

```
#include <filesystem>
#include <fstream>
#include <iostream>

namespace fs = std::filesystem;

int main() {
    // Work in a private scratch directory so the example is
    // self-contained and repeatable.
    const fs::path base = fs::temp_directory_path() / "modern-cpp-fs-demo";
    fs::remove_all(base);                    // clean up any previous run
    fs::create_directories(base / "sub");   // creates intermediate dirs

    // Create a file.
    std::ofstream(base / "hello.txt") << "hello, filesystem";

    // Path decomposition (no filesystem access required).
    const fs::path p = base / "hello.txt";
    std::cout << "filename:  " << p.filename()    << "\n";
    std::cout << "extension: " << p.extension()   << "\n";
    std::cout << "parent:    " << p.parent_path() << "\n";

    // Query the file.
    std::cout << "exists:          " << fs::exists(p)          << "\n";
    std::cout << "is_regular_file: " << fs::is_regular_file(p) << "\n";
    std::cout << "file_size:       " << fs::file_size(p)       << "\n";

    // Recursively iterate the directory tree.
    std::cout << "entries:\n";
    for (const auto& entry : fs::recursive_directory_iterator(base))
        std::cout << "  " << entry.path() << "\n";

    // Copy, then rename.
    fs::copy_file(p, base / "copy.txt");
```

```
    fs::rename(base / "copy.txt", base / "renamed.txt");

    // Clean up.
    fs::remove_all(base);
    std::cout << "after cleanup, exists: " << fs::exists(base) << "\n";
}
```

Further Readings

Chapter 09: Minor Features

9.1 New Type

(since C++11)

`long long int`

`long long int` is not the first to be introduced in C++11. 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

(since C++11)

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 marked `noexcept` performs an operation that possibly throws an exception, the compiler

will insert a `std::terminate()` call to the path exiting the function via an exception, in order to make the program terminate.

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

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

The final output is:

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

9.3 Literal

(since C++11)

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:

```cpp
#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:

```cpp
// 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`.

9.4 Memory Alignment

(since C++11)

C++ 11 introduces two new keywords, `alignof` and `alignas`, to support control of memory alignment. The `alignof` keyword can get a platform-dependent value of type `std::size_t` to query the alignment of the platform. Of course, we are sometimes not satisfied with this, and even want to customize the alignment of the structure. Similarly, C++ 11 introduces `alignas`. To reshape the alignment of a structure. Let's look at two examples:

```
#include <iostream>

struct Storage {
    char      a;
    int       b;
    double    c;
    long long d;
};

struct alignas(std::max_align_t) AlignasStorage {
    char      a;
```

```
    int       b;
    double    c;
    long long d;
};

int main() {
    std::cout << alignof(Storage) << std::endl;
    std::cout << alignof(AlignasStorage) << std::endl;
    return 0;
}
```

where `std::max_align_t` requires the same alignment for each scalar type, so it has almost no difference in maximum scalars. In turn, the result on most platforms is `long double`, so the alignment requirement for `AlignasStorage` we get here is 8 or 16.

Dynamic allocation of over-aligned types

Before C++17, a `new` expression could not guarantee the alignment requirement of an **over-aligned** type (one whose alignment exceeds `alignof(std::max_align_t)`); using such types often required platform-specific facilities such as `posix_memalign` or `_aligned_malloc`. C++17 introduced `operator new` / `operator delete` overloads taking a `std::align_val_t`, so a `new` expression automatically selects the aligned version when allocating an over-aligned type:

```
struct alignas(64) Aligned {
    double v[8];
};

Aligned* p = new Aligned; // C++17: automatically uses the aligned operator new
// now reinterpret_cast<std::uintptr_t>(p) % 64 == 0
delete p;
```

9.5 Type punning and std::bit_cast

(since C++20)

"Type punning" means reinterpreting the same memory as a different type, common in low-level code (e.g. reading the bit pattern of a floating-point number). Many people reach for `reinterpret_cast` through a pointer or reference:

```
float f = 3.14f;
std::uint32_t bits = *reinterpret_cast<std::uint32_t*>(&f); // undefined behavior!
```

But this violates the **strict-aliasing rule**: except through char, unsigned char, or std::byte, accessing an object via an lvalue of a type incompatible with the object's actual type is undefined behavior, and the optimizer is free to assume it never happens.

The correct, portable approach is std::memcpy (valid under any standard):

```
std::uint32_t bits;
std::memcpy(&bits, &f, sizeof bits); // well-defined
```

C++20 further provides std::bit_cast (in <bit>), which reinterprets the object representation in a well-defined way with clearer semantics and can be used in constant expressions:

```
#include <bit>
auto bits = std::bit_cast<std::uint32_t>(f); // both types must be the same size and trivially copya
float back = std::bit_cast<float>(bits);
```

9.6 Mathematical special functions

(since C++17)

C++17 added a set of mathematical special functions to <cmath> — such as std::riemann_zeta, std::beta, std::assoc_legendre, and std::cyl_bessel_j — useful in scientific computing and machine-learning related domains:

```
#include <cmath>
double z = std::riemann_zeta(2.0); // ~ 1.6449 (i.e. pi^2 / 6)
```

> Note that although these special functions are part of the standard, **standard-library support varies**: libstdc++ (GCC) provides a complete implementation, while libc++ (Clang) did not implement them for a long time. The code above therefore may not compile on every toolchain; check your standard library's support before using them.

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: 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 de-

cided 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

The 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:

```
#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:

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

abbreviate as:

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

Even use it directly as a type:

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

Let's look at a practical example. The following uses a `requires` expression to define a concept `Addable`, requiring that a type support `+` with a result convertible back to the type, and uses it to constrain a function template:

```
#include <concepts>
#include <iostream>

// Concept: T must support a + b, with a result convertible to T
template <typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

template <Addable T>
T sum(T a, T b) { return a + b; }

int main() {
    std::cout << sum(1, 2) << std::endl;     // 3
    std::cout << sum(1.5, 2.5) << std::endl; // 4
    // sum("a", "b"); // compile error: does not satisfy Addable, with a clear message
}
```

When an argument of an unsatisfying type is passed, the compiler tells us directly that the constraint is not satisfied, instead of emitting a long cascade of internal template-instantiation errors.

Modules

Modules aim to solve the many problems of the traditional header mechanism: repeated parsing, macro pollution, include-order sensitivity, and slow compilation. A module is declared with `export module` and explicitly `export`s the entities visible to the outside:

```
// math.cppm - module interface unit
export module math;

export int add(int a, int b) {
    return a + b;
}
```

Consumers use `import` instead of `#include`:

```
// main.cpp
import math;
import <iostream>;

int main() {
```

```
    std::cout << add(1, 2) << std::endl;
}
```

Unlike the other examples in this book, compiling modules requires dedicated toolchain support and usually two steps (compile the module interface unit first, then the consumer); it cannot be built with a single `clang++ file.cpp` command. Consult your compiler's documentation for the exact build procedure.

Ranges

Ranges provide a higher-level, composable abstraction over the standard-library algorithms and iterators. With **range adaptors** and the pipe operator `|`, several lazy transformations can be chained in a declarative style:

```
#include <iostream>
#include <vector>
#include <ranges>

int main() {
    std::vector<int> v{1, 2, 3, 4, 5, 6};
    auto result = v | std::views::filter([](int x) { return x % 2 == 0; })
                    | std::views::transform([](int x) { return x * x; });
    for (int x : result) std::cout << x << ' '; // 4 16 36
    std::cout << std::endl;
}
```

These views are **lazily evaluated**: the filtering and transformation are computed element by element only when `result` is iterated, with no intermediate container created.

Coroutines

A coroutine is a function that can be suspended and resumed. Any function whose body uses `co_await`, `co_yield`, or `co_return` is a coroutine. Note that C++20 provides only the **language machinery** plus the low-level support facilities in `<coroutine>`, leaving the "glue" such as the `promise_type` to the user or a library (a ready-made `std::generator` only arrived in C++23).

Here is a minimal lazy generator that yields values one at a time with `co_yield`:

```
#include <coroutine>
#include <iostream>
#include <optional>
```

```cpp
template <typename T>
struct Generator {
    struct promise_type {
        T current;
        Generator get_return_object() { return Generator{handle::from_promise(*this)}; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(T value) { current = value; return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
    using handle = std::coroutine_handle<promise_type>;
    handle h;
    explicit Generator(handle h) : h(h) {}
    ~Generator() { if (h) h.destroy(); }
    Generator(const Generator&) = delete;
    Generator(Generator&& o) noexcept : h(o.h) { o.h = {}; }
    std::optional<T> next() {
        if (!h || h.done()) return std::nullopt;
        h.resume();
        if (h.done()) return std::nullopt;
        return h.promise().current;
    }
};

Generator<int> range(int a, int b) {
    for (int i = a; i < b; ++i) co_yield i;
}

int main() {
    auto g = range(1, 5);
    while (auto v = g.next()) std::cout << *v << ' '; // 1 2 3 4
    std::cout << std::endl;
}
```

A note on Contracts and Transactional Memory

A common misconception is worth clarifying: **Contracts** and **Transactional Memory** are *not* part of C++20.

- Contracts were once in the C++20 working draft but were removed before the standard was published; they are now an important feature targeting C++26 (see this repository's C++26

tracking issue #318).

- Transactional Memory exists only as a Technical Specification (TS) and was never merged into the C++20 standard.

This chapter therefore no longer presents them as C++20 features.

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
- Modern C++ 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++ (C++11 to C++26), 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 briefly discuss the best practices of modern C++. Many of these ideas are distilled from Effective Modern C++ and the Google C++ Style Guide, as well as the C++ Core Guidelines maintained by Bjarne Stroustrup and Herb Sutter. The goal of this appendix is to summarize widely accepted practices that help ensure the overall quality of your code.

Common Tools

Good tooling catches a large class of problems before they ever reach production:

- **Enable warnings, and treat them as errors.** Compile with `-Wall -Wextra` (and consider `-Wpedantic`); `-Werror` keeps warnings from accumulating.
- **Run sanitizers during testing.** AddressSanitizer (`-fsanitize=address`), UndefinedBehavior-Sanitizer (`-fsanitize=undefined`) and ThreadSanitizer (`-fsanitize=thread`) catch memory errors, undefined behavior, and data races at run time.
- **Format and lint automatically.** `clang-format` keeps style consistent, and `clang-tidy` flags bug-prone patterns and suggests modernizations.
- **Use a modern build system and package manager**, such as CMake together with vcpkg or Conan, to make builds reproducible.
- **Experiment quickly** with Compiler Explorer to inspect the generated assembly and compare compilers and standards.

Coding Style

A consistent style makes a codebase far easier to read and maintain:

- Pick a style guide and apply it consistently across the whole project; consistency matters more than the specific choices.
- Be `const`-correct: mark variables, member functions, and parameters `const` whenever they do not need to mutate state.
- Use `auto` to avoid redundant type spelling, but not at the cost of readability — keep the reader able to tell what a name means.
- Prefer the standard library (containers, algorithms, `std::string`) over hand-rolled equivalents.

Overall Performance

- **Measure before you optimize.** Use a profiler to find the real hot spots instead of guessing; premature optimization wastes effort and harms readability.
- **Avoid unnecessary copies.** Pass large objects by `const&`, or take by value and `std::move` when you need a copy anyway; `reserve()` containers when the final size is known.
- **Let the compiler help.** Return local objects by value and rely on (guaranteed) copy elision rather than returning via output parameters.
- Prefer move semantics for expensive-to-copy types, and remember that `std::move` is a cast, not an operation.

Code Security

- **Manage resources with RAII.** Wrap every resource (memory, files, locks, sockets) in an object whose destructor releases it.
- **Prefer smart pointers** (`std::unique_ptr`, `std::shared_ptr`) over raw `new`/`delete`; avoid owning raw pointers.
- **Avoid undefined behavior**: no out-of-bounds access, no signed overflow, no use-after-free, no strict-aliasing violations (see §9.5). Sanitizers help detect these.
- Prefer `std::span`, `std::array`, and `.at()` over raw pointers and unchecked indexing when bounds matter, and avoid C-style casts in favor of the named C++ casts.

Maintainability

- Keep functions small and focused on a single responsibility; prefer standard algorithms over handwritten loops to express intent clearly.
- Make interfaces hard to misuse: use strong types and enum classes instead of bare `int`/`bool` flags.
- Write tests, and run them continuously so regressions are caught early.
- Document *why*, not *what*: the code already says what it does.

Portability

- Use fixed-width integer types (`std::int32_t`, etc.) from `<cstdint>` when the exact size matters; do not assume `int` is 32 bits or that `char` is signed.
- Avoid implementation-defined and platform-specific behavior; when you must depend on it, isolate it behind a small abstraction.
- Prefer the standard library over platform-specific APIs (e.g. `<filesystem>`, `<thread>`, `<chrono>`) so the same code builds across platforms.
- When byte order matters, query it explicitly (`std::endian` in C++20) rather than assuming little- or big-endian.