C++17 常用新特性

目录

折叠表达式

C++17中引入了折叠表达式，主要是方便模板编程，分为左右折叠

语法

(形参包 运算符 ...) (1)

(... 运算符 形参包) (2)

(形参包 运算符 ... 运算符 初值) (3)

(初值 运算符 ... 运算符 形参包) (4)

折叠表达式的实例化按以下方式展开成表达式 e：

1) 一元右折叠 (E 运算符 ...) 成为 (E1 运算符 (... 运算符 (EN-1 运算符 EN)))

2) 一元左折叠 (... 运算符 E) 成为 (((E1 运算符 E2) 运算符 ...) 运算符 EN)

3) 二元右折叠 (E 运算符 ... 运算符 I) 成为 (E1 运算符 (... 运算符 (EN−1 运算符 (EN 运算符 I))))

4) 二元左折叠 (I 运算符 ... 运算符 E) 成为 ((((I 运算符 E1) 运算符 E2) 运算符 ...) 运算符 EN)

（其中 N 是包展开中的元素数量）

折叠表达式

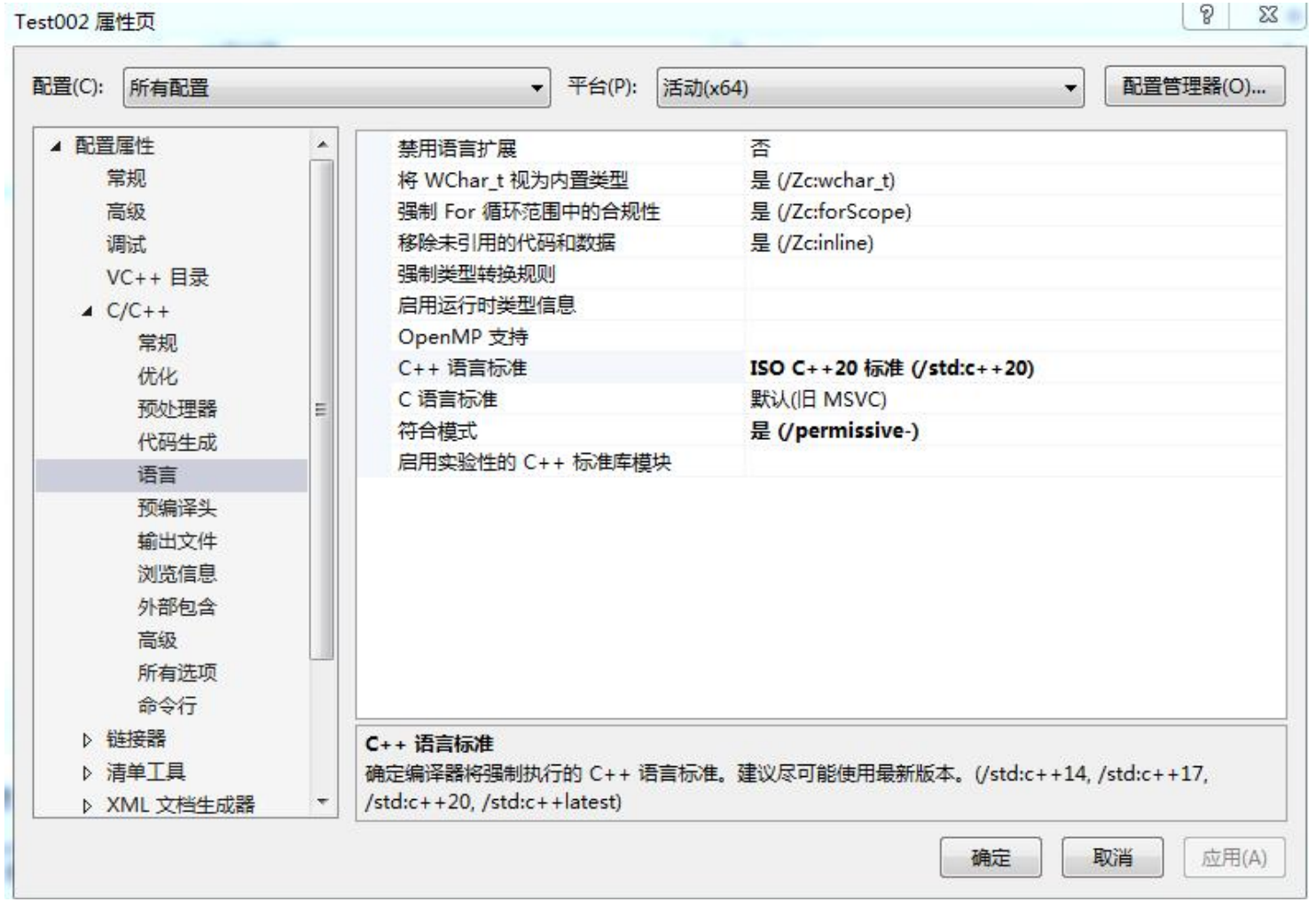
```
template<typename... Args>
bool all(Args... args) { return (... && args); }


bool b = all(true, true, true, false);
// 在 all() 中，一元左折叠展开成
//  return ((true && true) && true) && false;
// b 是 false
```

C++标准设置

解决方案下面的项目名上右键->属性->配置属性->C/C++->语言->C++语言标准

更改成自己的想要的版本



类模板参数推导

类模板实例化时，可以不必显式指定类型，前提是保证类型可以推导

```
#include <iostream>
using namespace std;
template<class T>
class ClassTest
{
public:
        ClassTest(T, T) {};
};
int main() {
        auto y = new ClassTest{ 100, 200 }; // 分配的类型是 ClassTest<int>
        return 0;
}
```

auto 占位的非类型模板形参

```
#include <iostream>
using namespace std;
template <auto T> void func1() {
        cout << T << endl;
}
int main() {
        func1<100>();
        //func1<int>();
        return 0;
}
```

编译期constexpr if语句

```
#include <iostream>
using namespace std;
template <bool ok> constexpr void func2() {
    //在编译期进行判断，if和else语句不生成代码
    if constexpr (ok == true) {
        //当ok为true时，下面的else块不生成汇编代码
        cout << "ok" << endl;
    }
    else {
        //当ok为false时，上面的if块不生成汇编代码
        cout << "not ok" << endl;
    }
}
int main() {
    func2<true>(); //输出ok，并且汇编代码中只有 cout << "ok" << endl;
    func2<false>(); //输出not ok，并且汇编代码中只有 cout << "not ok" << endl;
    return 0;
}
```

inline变量

扩展的inline用法，使得可以在头文件或者类内初始化静态成员变量

```
// mycode.h
inline int value  = 100;
// mycode.cpp
class AAA {
  inline static int value2 = 200;
};
```

结构化绑定

在C++11中，如果需要获取tuple中元素，需要使用get<>()函数或者tie<>函数，这个函数可以把tuple中的元素值转换为可以绑定到tie<>()左值的集合，也就是说需要已分配好的内存去接收；用起来不方便。

```
int main() {
        auto student = make_tuple(string{ "Zhangsan" }, 19, string{ "man" });
        string name;
        size_t age;
        string gender;
        tie(name, age, gender) = student;
        cout << name << ", " << age << ", " << gender << endl;
        // Zhangsan, 19, man
        return 0;
}
```

结构化绑定

C++17中的结构化绑定，大大方便了类似操作，而且使用引用捕获时，还可以修改捕获对象里面的值，代码也会简洁很多

```
int main() {

	auto student = make_tuple(string{ "Zhangsan" }, 19, string{ "man" });
	auto [name, age, gender] = student;
	cout << name << ", " << age << ", " << gender << endl;

	return 0;
}
```

if switch初始化

使用迭代器操作时，可以使代码更紧凑。

```
// C++11
unordered_map<string, int> stu1{ {"zhangsan", 18}, {"wangwu", 19} };
auto iter = stu1.find("wangwu");
if (iter != stu1.end()) {
        cout << iter->second << endl;
}
// C++17
if (auto iter = stu1.find("wangwu"); iter != stu1.end()) {
        cout << iter->second << endl;
}
```

简化的嵌套命名空间

```
// C++17之前
namespace A {
namespace B {
namespace C {
void func1() {}
} // namespace C
} // namespace B
} // namespace A

// C++17
namespace A::B::C {
void func1() {}
} // namespace A::B::C
```

using声明语句可以声明多个名称

```
using std::cout, std::cin;
```

lambda表达式捕获 *this

一般情况下，lambda表达式访问类成员变量时需要捕获this指针，这个this指针指向原对象，即相当于一个引用，在多线程情况下，有可能lambda的生命周期超过了对象的生命周期，此时，对成员变量的访问是未定义的。

因此C++17中增加捕获*this，此时捕获的是对象的副本，也可以理解为只能对原对象进行读操作，没有写权限。

lambda表达式捕获 *this

```
#include <iostream>
using namespace std;
class ClassTest {
public:
    int num;
    void func1() {
        auto lamfunc = [*this]() { cout << num << endl; };
        lamfunc();
    }
};

int main() {
    ClassTest a;
    a.num = 100;
    a.func1();

    return 0;
}
```

简化重复命名空间的属性列表

为类型、对象、代码等引入由实现定义的属性。

[[属性]] [[属性1, 属性2, 属性3(实参)]] [[命名空间::属性(实参)]] alignas说明符
正式而言，语法是

[[属性列表]] (C++11 起)
[[using 属性命名空间 : 属性列表]] (C++17 起)
其中 属性列表 是由逗号分隔的零或更多个 属性 的序列（可以以指示包展开的省略号 ... 结束）

标识符
属性命名空间 :: 标识符
标识符 (实参列表)
属性命名空间 :: 标识符 (实参列表)
1) 简单属性，例如 [[noreturn]]
2) 有命名空间的属性，例如 [[gnu::unused]]
3) 有实参的属性，例如 [[deprecated("because")]]
4) 既有命名空间又有实参列表的属性

简化重复命名空间的属性列表

```
[[gnu::always_inline]] [[gnu::hot]] [[gnu::const]] [[nodiscard]]
inline int f(); // 声明 f 带四个属性

[[gnu::always_inline, gnu::const, gnu::hot, nodiscard]]
int f(); // 同上，但使用含有四个属性的单个属性说明符

// C++17:
[[using gnu:const, always_inline, hot]] [[nodiscard]]
int f [[gnu::always_inline]] (); // 属性可出现于多个说明符中

int f() { return 0; }
```

__has_include

跨平台项目需要考虑不同平台编译器的实现，使用__has_include可以判断当前环境下是否存在某个头文件。

```
int main() {
#if __has_include("iostream")
  cout << "iostream exist." << endl;
#endif

#if __has_include(<cmath>)
  cout << "<cmath> exist." << endl;
#endif
  return 0;
}
```

新增属性

[[fallthrough]]

switch语句中跳到下一条语句，不需要break，让编译器忽略告警。

```
int i = 1;
int result;
switch (i) {
case 0:
    result = 1; // warning
case 1:
    result = 2;
    [[fallthrough]]; // no warning
default:
    result = 0;
    break;
}
```

新增属性

[[nodiscard]]

所修饰的内容不可被忽略，主要用于修饰函数返回值

当用于描述函数的返回值时，如果调用函数的地方没有获取返回值时，编译器会给予警告

```
[[nodiscard]] auto func(int a, int b) { return a + b; }


int main() {
  func(2, 3); // 警告
  return 0;
}
```

新增属性

[[maybe_unused]]

用于描述暂时没有被使用的函数或变量，以避免编译器对此发出警告

```
[[maybe_unused]] void func()       //没有被使用的函数
{
        cout << "test" << endl;
}
int main()
{
        [[maybe_unused]] int num = 0; //没有被使用的变量
        return 0;
}
```

charconv

<charconv>是C++17新的标准库头文件，包含了相关类和两个转换函数。

可以完成传统的整数/浮点和字符串互相转换的功能(atoi、itoa、atof、sprintf等)，同时支持输出格式控制、整数基底设置并且将整数和浮点类型对字符串的转换整合了起来。

是独立于本地环境、不分配、不抛出的。目的是在常见的高吞吐量环境，例如基于文本的交换（ JSON 或 XML ）中，允许尽可能快的实现。

charconv

chars_format

chars_format是作为格式控制的类定义在<charconv>头文件中

```
enum class chars_format {
    scientific = /*unspecified*/,
    fixed = /*unspecified*/,
    hex = /*unspecified*/,
    general = fixed | scientific
};
```

from_chars

from_chars

```
struct from_chars_result {
    const char* ptr;
    std::errc ec;
};
std::from_chars_result from_chars(const char* first, const char* last,
                                  /*see below*/& value, int base = 10);
std::from_chars_result from_chars(const char* first, const char* last, float& value,
                                  std::chars_format fmt = std::chars_format::general);
std::from_chars_result from_chars(const char* first, const char* last, double& value,
                                  std::chars_format fmt = std::chars_format::general);
std::from_chars_result from_chars(const char* first, const char* last, long double& value,
                                  std::chars_format fmt = std::chars_format::general);
```

first, last - 要分析的合法字符范围
value - 存储被分析值的输出参数，若分析成功
base - 使用的整数基底： 2 与 36 间的值（含上下限）。
fmt - 使用的浮点格式， std::chars_format 类型的位掩码

to_chars

```
struct to_chars_result {
    char* ptr;
    std::errc ec;};
std::to_chars_result to_chars(char* first, char* last, value, int base = 10);
```

整数转字符串

first, last - 要写入的字符范围

value - 要转换到其字符串表示的值

base - 使用的整数基底： 取值范围[2,36]。

```
std::to_chars_result to_chars(char* first, char* last, float       value,std::chars_format fmt, int precision);
std::to_chars_result to_chars(char* first, char* last, double      value,  std::chars_format fmt, int precision);
std::to_chars_result to_chars(char* first, char* last, long double value, std::chars_format fmt, int precision);
```

浮点转字符串

fmt - 使用的浮点格式， std::chars_format 类型的位掩码

precision - 使用的浮点精度

charconv案例代码

```
#include <iostream>
#include <charconv>
using namespace std;
int main() {
    string s1{ "123456789" };
    int val = 0;
    auto res = from_chars(s1.data(), s1.data() + 4, val);//把s1的前4个转成整数
    if (res.ec == errc()) {
        cout << "val: " << val << ", distance: " << res.ptr - s1.data() << endl;
        // val: 1234, distance: 4

    }
    else if (res.ec == errc::invalid_argument) {
        cout << "invalid" << endl;
    }

    s1 = string{ "12.34" };
    double value = 0;
    auto format = chars_format::general;
    res = from_chars(s1.data(), s1.data() + s1.size(), value, format); //把"12.34"转成小数
    cout << "value: " << value << endl;
    // value: 12.34

    s1 = string{ "xxxxxxxx" };
    int v = 1234;
    auto result = to_chars(s1.data(), s1.data() + s1.size(), v);//把整数转成字符串
    cout << "str: " << s1 << ", filled: " << result.ptr - s1.data()
        << " characters." << endl;
    // s1: 1234xxxx, filled: 4 characters.
    return 0;
}
```

std::variant

C++17中提供了std::variant类型，意为多变的，可变的类型。

有点类似于加强版的union，里面可以存放复合数据类型，且操作元素更为方便。

可以用于表示多种类型的混合体，但同一时间只能用于代表一种类型的实例。

variant提供了index成员函数，该函数返回一个索引，该索引用于表示variant定义对象时模板参数的索引(起始索引为0)，同时提供了一个函数holds_alternative<T>(v)用于查询对象v当前存储的值类型是否是T

std::variant

```
#include <iostream>
#include <variant>
#include <string>
using namespace std;
int main() {
        variant<int, double, string> d; //int 0  double 1 string 2
        cout << d.index() << endl;    //输出：0
        d = 3.14;
        cout << d.index() << endl;    //输出：1
        d = "hi";
        cout << d.index() << endl;    //输出：2

        cout << holds_alternative<int>(d) << endl;    //输出：0
        cout << holds_alternative<double>(d) << endl; //输出：0
        cout << holds_alternative<string>(d) << endl; //输出：1
        return 0;
}
```

std::optional

在 C 时代以及早期 C++ 时代，语法层面支持的 nullable 类型可以采用指针方式： T* ，如果指针为 NULL （ C++11 之后则使用 nullptr ） 就表示无值状态（ empty value ）。

在编程中，经常遇到这样的情况:可能返回/传递/使用某种类型的对象。也就是说，可以有某个类型的值，也可以没有任何值。因此，需要一种方法来模拟类似指针的语义，在指针中，可以使用nullptr来表示没有值。

处理这个问题的方法是定义一个特定类型的对象，并用一个额外的布尔成员/标志来表示值是否存在。std::optional<>以一种类型安全的方式提供了这样的对象。

注意：每个版本可能对某些特征做了改动。

std::optional

optional是一个模板类：

template <class T>

class optional;

它内部有两种状态，要么有值（T类型），要么没有值（std::nullopt）。有点像T*指针，要么指向一个T类型，要么是空指针(nullptr)。

std::optional有以下几种构造方式：

```
#include <iostream>
#include <optional>
using namespace std;
int main() {
    optional<int> o1;  //什么都不写时默认初始化为nullopt
    optional<int> o2 = nullopt;  //初始化为无值
    optional<int> o3 = 10;  //用一个T类型的值来初始化
    optional<int> o4 = o3;  //用另一个optional来初始化
    return 0;
}
```

std::optional

查看一个optional对象是否有值，可以直接用if，或者用has_value()

```
optional<int> o1;
if (o1) {
    printf("o1 has value\n");
}
if (o1.has_value()) {
    printf("o1 has value\n");
```

std::optional

当一个optional有值时，可以通过用指针的方式(*号和->号)来使用它，或者用.value()拿到它的值：

```
optional<int> o1 = 100;
cout << *o1 << endl;
optional<string> o2 = "orange";
cout << o2->c_str() << endl;
cout << o2.value().c_str() << endl;
```

std::optional

将一个有值的optional变为无值，用.reset()。该函数会将已存储的T类型对象析构掉

```
optional<int> o1 = 500;
o1.reset();
```

std::any

在C++11中引入的auto自动推导类型变量大大方便了编程，但是auto变量一旦声明，该变量类型不可再改变。

C++17中引入了std::any类型，该类型变量可以存储任何类型的值，也可以时刻改变它的类型，类似于python中的变量。

std::any

```
#include <any>
#include <iostream>
using namespace std;
int main() {
    cout << boolalpha; //将bool值用 "true" 和 "false"显示
    any a;  //定义一个空的any，即一个空的容器

    //有两种方法来判断一个any是否是空的
    cout << a.has_value() << endl; // any是空的时，has_value 返回值为 false
    cout << a.type().name() << endl; //any 是空的时，has_value 返回值为 true

    //几种创建any的方式
    any b = 1;                      //b 为存了int类型的值的any
    auto c = make_any<float>(5.0f);    //c为存了float类型的any
    any d(6.0);                     //d为存储了double类型的any
    cout << b.has_value() << endl;   //true
    cout << b.type().name() << endl; //int

    cout << c.has_value() << endl;   //true
    cout << c.type().name() << endl; //float

    cout << d.has_value() << endl;   //true
    cout << d.type().name() << endl; //double
```

std::any

```
//更改any的值
    a = 2;   //直接重新赋值
    auto e = c.emplace<float>(4.0f); //调用emplace函数，e为新生成的对象引用

    //清空any的值
    b.reset();
    cout << b.has_value() << endl;   //false
    cout << b.type().name() << endl; //int

    //使用any的值
    try
    {
        auto f = any_cast<int>(a); //f为int类型，其值为2
        cout << f << endl; //2
    }
    catch (const bad_any_cast& e)
    {
        cout << e.what() << endl;
    }

    try
    {
        auto g = any_cast<float>(a); //抛出std::bad_any_cat 异常
        cout << g << endl; //该语句不会执行
    }
    catch (const bad_any_cast& e)
    {
        cout << e.what() << endl;  //可能输出Bad any_cast
    }
    return 0;
}
```

std::apply

将tuple元组解包，并作为函数的传入参数

```
#include <any>
#include <iostream>
using namespace std;
int add(int a, int b) {
	return a + b;
}
int main() {
	auto add_lambda = [](auto a, auto b, auto c) { return a + b + c; };

	cout << apply(add, pair(2, 3)) << endl; //5
	cout << apply(add_lambda, tuple(2, 3, 4)) << endl; //9
	return 0;
}
```

std::make_from_tuple

解包tuple作为构造函数参数构造对象

```
#include <iostream>
using namespace std;
class ClassTest {
public:
		string _name;
		size_t _age;

		ClassTest(string name, size_t age) : _name(name), _age(age) {
			cout << "name: " << _name << ", age: " << _age << endl;
		}
};

int main() {
		auto param = make_tuple("zhangsan", 19);
		make_from_tuple<ClassTest>(move(param));

		return 0;
}
```

std::string_view

C++中字符串有两种形式，char*和std::string，string类型封装了char*字符串，让我们对字符串的操作方便了很多，但是会有些许性能的损失，而且由char*转为string类型，需要调用string类拷贝构造函数，也就是说需要重新申请一片内存，但如果只是对源字符串做只读操作，这样的构造行为显然是不必要的。

在C++17中，增加了std::string_view类型，它通过char*字符串构造，但是并不会去申请内存重新创建一份该字符串对象，只是char*字符串的一个视图，优化了不必要的内存操作。相应地，对源字符串只有读权限，没有写权限。

std::string_view

```
#include <iostream>
using namespace std;
void func1(string_view str_v) {
	cout << str_v << endl;
	return;
}

int main() {
	const char* charStr = "hello world";
	string str{ charStr };
	string_view str_v(charStr, strlen(charStr));

	cout << "str: " << str << endl;
	cout << "str_v: " << str_v << endl;
	func1(str_v);

	return 0;
}
```

std::as_const

将左值转化为const类型

```
#include <iostream>
using namespace std;
int main() {
	string str={ "C++ as const test" };
	cout << is_const<decltype(str)>::value << endl;

	const string str_const = as_const(str);
	cout << is_const<decltype(str_const)>::value << endl;
	return 0;
}
```

std::filesystem

C++17中引入了filesystem，方便处理文件，提供接口函数很多，用起来也很方便。

一定是C++17标准及以上版本。

项目属性->C/C++->语言->C++语言标准设置为：ISO C++17 标准 (/std:c++17)

std::filesystem

C++17中引入了filesystem，方便处理文件，提供接口函数很多，用起来也很方便。

一定是C++17标准及以上版本。

项目属性->C/C++->语言->C++语言标准设置为：ISO C++17 标准 (/std:c++17)

std::filesystem

头文件及命名空间

```
#include<filesystem>
using namespace std::filesystem
```

常用类

path类：路径处理

directory_entry类：文件入口

directory_iterator类：获取文件系统目录中文件的迭代器容器

file_status类：用于获取和修改文件（或目录）的属性

std::filesystem path类

函数名	功能
path& append(const _Src& source)	在path末尾加入一层结构
path& assign(string_type& source)	赋值（字符串）
void clear()	清空
int compare(const path& other)	进行比较
bool empty()	空判断
path filename()	返回文件名（有后缀）
path stem()	返回文件名（不含后缀）
path extension()	返回文件后缀名
path is_absolute()	判断是否为绝对路径
path is_relative()	判断是否为相对路径
path relative_path()	返回相对路径
path parent_path()	返回父路径
path& replace_extension(const path& replace)	替换文件后缀

std::filesystem

常用函数

std::filesystem::exists(const path& pval):用于判断path是否存在

std::filesystem::copy(const path& from, const path& to)：目录复制

std::filesystem::absolute(const path& pval, const path& base = current_path()):获取相对于base的绝对路径

std::filesystem::create_directory(const path& pval)：当目录不存在时创建目录

std::filesystem::create_directories(const path& pval):形如/a/b/c这样的，如果都不存在，创建目录结构

std::filesystem::file_size(const path& pval)：返回目录的大小

std::filesystem

```
#include <ctime>
#include <iostream>
#include <filesystem>
using namespace std;
int main()
{
    namespace fs = std::filesystem;

    auto testdir = fs::path("./testdir");

    if (!fs::exists(testdir))//文件是否存在
    {
        cout << "file or directory is not exists!" << endl;
    }
```

std::filesystem

```
// none   （默认）跳过符号链接，权限拒绝是错误。
  // follow_directory_symlink              跟随而非跳过符号链接。
  // skip_permission_denied 跳过若不跳过就会产生权限拒绝错误的目录。
  fs::directory_options opt(fs::directory_options::none);

  fs::directory_entry dir(testdir);
  // 遍历当前目录
  std::cout << "show:\t" << dir.path().filename() << endl;
  for (fs::directory_entry const& entry : fs::directory_iterator(testdir, opt))
  {
     if (entry.is_regular_file())
     {
        cout << entry.path().filename()
           << "\t size: " << entry.file_size() << endl;
     }
     else if (entry.is_directory())
     {
        cout << entry.path().filename() << "\t dir" << endl;
     }
  }
  cout << endl;
```

std::filesystem

```
// 递归遍历所有的文件
   cout << "show all:\t" << dir.path().filename() << endl;
   for (fs::directory_entry const& entry : fs::recursive_directory_iterator(testdir, opt))
   {
      if (entry.is_regular_file())
      {
         cout << entry.path().filename() << "\t size: " << entry.file_size() << "\t parent: " << entry.path().parent_path() << endl;
      }
      else if (entry.is_directory())
      {
         cout << entry.path().filename()<< "\t dir" << endl;
      }
   }
   return 0;
}
```

知识点总结

1. 语言特性
2. 库相关

作业

1. 创建一个遍历目录的函数