C++17新特性个人总结

C++17

1 关键字

1.1 constexpr

1.2 static\_assert

1.3 auto

1.4 typename

1.5 inline

2 语法

2.1 折叠表达式

2.2 结构化绑定

2.3 允许非类型模板参数进行常量计算

2.4 条件分支语句初始化

2.5 聚合初始化

2.6 嵌套命名空间

2.7 lambda表达式捕获\*this的值

2.8 枚举[类]对象的构造

2.9 十六进制单精度浮点数字面值

2.10 基于对齐内存的动态内存分配

2.11 细化表达式的计算顺序

2.12 模板类的模板参数自动推导

2.13 简化重复命名空间的属性列表

2.14 不支持、非标准的属性

2.15 改写与继承构造函数

2.16 内联变量

2.17 用auto作为非类型模板参数

3 宏

3.1 \_\_has\_include

4 属性

4.1 fallthrough

4.2 nodiscard

4.3 maybe\_unuse

结语

C++17

编译器版本：GCC 7.1、Clang 5.0

\_\_cplusplus：201703L

编译选项：-std=c++17

1 关键字

1.1 constexpr

扩展constexpr使用范围，可用于if语句中，也可用于lambda表达式中。

例子1：

#include<iostream>

template<bool ok>

constexpr void foo()

{

//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

std::cout << "ok" << std::endl;

}

else

{

//当ok为false时，上面的if块不生成汇编代码

std::cout << "not ok" << std::endl;

}

}

int main()

{

foo<true>();//输出ok，并且汇编代码中只有std::cout << "ok" << std::endl;这一句

foo<false>();//输出not ok，并且汇编代码中只有std::cout << "not ok" << std::endl;这一句

return 0;

}

例子2：

int main()

{

constexpr auto add1 = [](int n, int m){

auto func1 = [=] { return n; }; //func1 lambda表达式

auto func2 = [=] { return m; }; //func2 lambda表达式

return [=] { return func1() + func2(); };

};

constexpr auto add2 = [](int n, int m){

return n + m;

};

auto add3 = [](int n, int m){

return n + m;

};

int sum1 = add1(30, 40)( ); //传入常量值，add1在编译期计算，立即返回70

int sum2 = add2(sum1, 4); //由于传入非constexpr变量，add2的constexpr失效，变成运行期lambda

constexpr int sum3 = add3(1, 2); //sum3为constexpr变量，传入常量值，add3变成编译期lambda，立即返回3

int sum4 = add2(10, 2);//传入常量值，add2在编译期计算，立即返回12

return 0;

}

1.2 static\_assert

扩展static\_assert用法，静态断言的显示文本可选。

如：

static\_assert(true, "");

static\_assert(true);//c++17支持

1.3 auto

扩展auto的推断范围

如：

auto x1 = { 1, 2 }; //推断出std::initializer\_list<int>类型

auto x2 = { 1, 2.0 }; //错误：类型不统一，无法推断

auto x3{ 1, 2 }; //错误：auto的聚合初始化只能一个元素

auto x4 = { 3 }; //推断出std::initializer\_list<int>类型

auto x5{ 3 }; //推断出int类型

1.4 typename

扩展用法，允许出现在模板的模板的参数中。

首先回顾一下typename的用法，①用于模板中，表示模板参数为类型；②用于声明某名字是变量名

如例1：

struct A

{

typedef int Example;

};

//第一种用法：声明模板参数为类型

template<typename T>

struct B { };

struct C

{

typedef typename A::Example E;//第二种用法：声明某名字为一种类型

};

int main()

{

typename A::Example e;//第二种用法：声明某名字为一种类型

return 0;

}

新特性下的typename用法，

如例2：

#include<iostream>

#include<typeinfo>

template<typename T>

struct A

{

int num;

A()

{

std::cout << "A Construct" << std::endl;

std::cout << "template typename is: " << typeid (T).name() << std::endl;

}

};

//此处的T可省略，X代表模板类型，T和X前的typename可替换成class

template<template<typename T> typename X>

struct B

{

X<double> e;

B() { std::cout << "B Construct" << std::endl; }

};

int main()

{

A<B<A>> a;

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;

B<A> b;

return 0;

}

运行结果：

1.5 inline

扩展用法，可用于定义内联变量，功能与内联函数相似。inline可避免函数或变量多重定义的问题，如果已定义相同的函数或变量（且该函数或变量声明为inline），编译器会自动链接到该函数或变量。

如（不发生错误）：

// test.h

inline void print()

{

std::cout << "hello world" << std::endl;

}

inline int num = 0;

// func.h

include "test.h"

inline void add(int arg)

{

num += arg;

print();

}

// main.cpp

include "func.h"

int main()

{

num = 0;

print();

add(10);

return 0;

}

2 语法

2.1 折叠表达式

用于变长参数模板的解包，只支持各种运算符（和操作符），分左、右折叠

如：

#include<string>

template<typename ... T>

auto sum(T ... arg)

{

return (arg + ...);//右折叠

}

template<typename ... T>

double sum\_strong(T ... arg)

{

return (arg + ... + 0);//右折叠

}

template<typename ... T>

double sub1(T ... arg)

{

return (arg - ...);//右折叠

}

template<typename ... T>

double sub2(T ... arg)

{

return (... - arg);//左折叠

}

int main()

{

int s1 = sum(1, 2, 2, 4, 5);//解包：((((1+)2+)3+)4+)5 = 15

double s2 = sum(1.1, 2.2, 3.3, 4.4, 5.5, 6.6);

double s3 = sum(1, 2.2, 3, 4.4, 5);

double s4 = sub1(5, 2, 1, 1);//解包：((((5-)2-)1-)1) = 1

double s5 = sub2(5, 2, 1, 1);//解包：(5-(2-(1-(1)))) = 3

double s6 = sum\_strong();//s6 = 0

std::string str1("he");

std::string str2("ll");

std::string str3("o ");

std::string str4("world");

std::string str5 = sum(str1, str2, str3, str4);//str5 = "hello world"

return 0;

}

2.2 结构化绑定

用一对包含一个或多个变量的中括号，表示结构化绑定，但是使用结构化绑定时，须用auto关键字，即绑定时声明变量

例子1：

/\*

\* 例子：多值返回

\*/

struct S

{

double num1;

long num2;

};

S foo(int arg1, double arg2)

{

double result1 = arg1 \* arg2;

long result2 = arg2 / arg1;

return {result1, result2};//返回结构体S对象

};

int main()

{

auto [num1, num2] = foo(10, 20.2);//自动推导num1为double，num2为long

return 0;

}

例子2：

#include<list>

#include<map>

/\*

\* 例子：循环遍历

\*/

template<typename T, typename U>

struct MyStruct

{

T key;

U value;

};

int main()

{

std::list<MyStruct<int, double>> Container1;

std::map<int, MyStruct<long long, char>> Container2;

for(auto [key, value] : Container1)

{

//key为int类型，value为double类型

}

for(auto [key, value] : Container2)

{

//key为int类型，value为MyStruct<long long, char>类型

//value1为long long类型，value2为char类型

auto [value1, value2] = value;

}

return 0;

}

2.3 允许非类型模板参数进行常量计算

非类型模板参数可传入类的静态成员

如：

class MyClass

{

public:

static int a;

};

template<int \*arg>

void foo() {}

int main()

{

foo<&MyClass::a>();

return 0;

}

2.4 条件分支语句初始化

在if和switch中可进行初始化

如：

template<long value>

void foo(int &ok)

{

if constexpr (ok = 10; value > 0)

{

}

}

int main()

{

int num = 0;

if(int i = 0; i == 0)

{

}

foo<10>(num);

switch(int k = 10; k)

{

case 0:break;

case 1:break;

default:break;

}

return 0;

}

2.5 聚合初始化

在初始化对象时，可用花括号进行对其成员进行赋值

如：

struct MyStruct1

{

int a;

int b;

};

struct MyStruct2

{

int a;

MyStruct1 ms;

};

int main()

{

MyStruct1 a{10};

MyStruct2 b{10, 20};

MyStruct2 c{1, {}};

MyStruct2 d{{}, {}};

MyStruct2 e{{}, {1, 2}};

return 0;

}

2.6 嵌套命名空间

简化多层命名空间的写法

如：

//传统写法

namespace A

{

namespace B

{

namespace C

{

};

};

};

//新写法

namespace A::B::C

{

};

2.7 lambda表达式捕获\*this的值

lambda表达式可捕获\*this的值，但this及其成员为只读

如：

struct MyStruct {

double ohseven = 100.7;

auto f() {

return [this] {

return [\*this] {

this->ohseven = 200.2;//错误，只读变量不可赋值

return ohseven;//正确

};

}();

}

auto g() {

return []{

return [\*this]{};//错误，外层lambda表达式没有捕获this

}();

}

};

2.8 枚举[类]对象的构造

可以给枚举[类]对象赋值

如：

enum MyEnum { value };

MyEnum me {10};//错误：不能用int右值初始化MyEnum类型对象

enum byte : unsigned char { };

byte b { 42 }; //正确

byte c = { 42 }; //错误：不能用int右值初始化byte类型对象

byte d = byte{ 42 }; //正确，其值与b相等

byte e { -1 }; //错误：常量表达式-1不能缩小范围为byte类型

struct A { byte b; };

A a1 = { { 42 } }; //错误：不能用int右值初始化byte类型对象

A a2 = { byte{ 42 } }; //正确

void f(byte);

f({ 42 }); //错误：无类型说明符

enum class Handle : unsigned int { value = 0 };

Handle h { 42 }; //正确

2.9 十六进制单精度浮点数字面值

以0x前缀开头的十六进制数，以f后缀的单精度浮点数，合并，就有了十六进制的单精度浮点数

如：

int main()

{

float value = 0x1111f;

return 0;

}

2.10 基于对齐内存的动态内存分配

谈到动态内存分配，少不了new和delete运算符，新标准中的new和delete运算符新增了按照对齐内存值来分配、释放内存空间的功能（即一个新的带对齐内存值的new、delete运算符重载）

函数原型：

void\* operator new(std::size\_t size, std::align\_val\_t alignment);

void\* operator new[](std::size\_t size, std::align\_val\_t alignment);

void operator delete(void\*, std::size\_t size, std::align\_val\_t alignment);

void operator delete[](void\*, std::size\_t size, std::align\_val\_t alignment);

参数说明：

size —— 分配的字节数。必须为alignment的整数倍。

alignment —— 指定的对齐内存值。必须是实现支持的合法对齐。

new的返回值：

成功，返回指向新分配内存起始地址的指针。

用法例子：

#include<new>

struct alignas(8) A {};

int main()

{

A \*a = static\_cast<A \*>(::operator new(sizeof(A), static\_cast<std::align\_val\_t>(alignof (A))));

::operator delete(a, sizeof(A), static\_cast<std::align\_val\_t>(alignof (A)));

return 0;

}

2.11 细化表达式的计算顺序

为了支持泛型编程和重载运算符的广泛使用，新特性将计算顺序进行的细化

如以下争议代码段：

#include<map>

int main()

{

std::map<int, int> tmp;

//对于std::map的[]运算符重载函数，在使用[]新增key时，std::map就已经插入了一个新的键值对

tmp[0] = tmp.size();//此处不知道插入的是{0, 0}还是{0, 1}

return 0;

}

为了解决该情况，新计算顺序规则为：

①后缀表达式从左到右求值。这包括函数调用和成员选择表达式。

②赋值表达式从右向左求值。这包括复合赋值。

③从左到右计算移位操作符的操作数。

2.12 模板类的模板参数自动推导

定义模板类的对象时，可以不指定模板参数，但必须要在构造函数中能推导出模板参数

如：

template<class T> struct A {

explicit A(const T&, ...) noexcept {} // #1

A(T&&, ...){} // #2

};

int i;

A a1 = { i, i }; //错误，不能根据#1推导为右值引用，也不能通过#1实现复制初始化

A a2{i, i}; //正确，调用#1初始化成功，a2推导为A<int>类型

A a3{0, i}; //正确，调用#2初始化成功，a2推导为A<int>类型

A a4 = {0, i}; //正确，调用#2初始化成功，a2推导为A<int>类型

template<class T> A(const T&, const T&) -> A<T&>; // #3

template<class T> explicit A(T&&, T&&) -> A<T>; // #4

A a5 = {0, 1}; //错误，#1和#2构造函数结果相同（即冲突）。根据#3推导为A<int&>类型

A a6{0, 1}; //正确，通过#2推断为A<int>类型

A a7 = {0, i}; //错误，不能将非静态左值引用绑定到右值。根据#3推导为A<int&>类型

A a8{0, i}; //错误，不能将非静态左值引用绑定到右值。根据#3推导为A<int&>类型

template<class T>

struct B {

template<class U>

using TA = T;//定义别名

template<class U>

B(U, TA<U>);//构造函数

};

B b{(int\*)0, (char\*)0}; //正确，推导为B<char \*>类型

2.13 简化重复命名空间的属性列表

如：

[[ using CC: opt(1), debug ]] void f() {}

//作用相同于 [[ CC::opt(1), CC::debug ]] void f() {}

2.14 不支持、非标准的属性

在添加属性列表时，编译器会忽略不支持的非标准的属性，不会发出警告和错误。

2.15 改写与继承构造函数

在类的继承体系中，构造函数的自动调用是一个令人头疼的问题。新特性引入继承与改写构造函数的用法。

例子1：

#include<iostream>

struct B1

{

B1(int) { std::cout << "B1" << std::endl; }

};

struct D1 : B1 {

using B1::B1;//表示继承B1的构造函数

};

D1 d1(0); //正确，委托基类构造函数进行初始化，调用B1::B1(int)

例子2：

#include<iostream>

struct B1

{

B1(int) { std::cout << "B1" << std::endl; }

};

struct B2

{

B2(int) { std::cout << "B2" << std::endl; }

};

struct D1 : B1, B2 {

using B1::B1;//表示继承B1的构造函数

using B2::B2;//表示继承B2的构造函数

};

D1 d1(0); //错误：函数冲突，

struct D2 : B1, B2

{

using B1::B1;

using B2::B2;

//正确，D2::D2(int)隐藏了B1::B1(int)和B2::B2(int)。另外由于B1和B2没有默认的构造函数，因此必须显式调用B1和B2的构造函数

D2(int) : B1(1), B2(0)

{ std::cout << "D2" << std::endl; }

};

struct D3 : B1

{

using B1::B1;

};

D3 d3(0);//正确，继承B1的构造函数，即利用B1的构造函数来初始化，输出B1

// 程序入口

int main()

{

D2 d(100);//编译通过，输出B1 B2 D2

return 0;

}

例子3：

#include<iostream>

struct B1

{

B1() { std::cout << "B1 default" << std::endl; }

B1(int) { std::cout << "B1" << std::endl; }

};

struct B2

{

B2() { std::cout << "B2 default" << std::endl; }

B2(int) { std::cout << "B2" << std::endl; }

};

struct D1 : B1, B2

{

using B1::B1;

using B2::B2;

//正确，D2::D2(int)隐藏了B1::B1(int)和B2::B2(int)，但必须要显示调用B1和B2的构造函数

D1(int) : B1(1), B2(0)

{ std::cout << "D2" << std::endl; }

//有默认构造函数，在不显示调用基类的构造函数时自动调用基类的默认构造函数

D1() { std::cout << "D2 default" << std::endl; }

};

// 程序入口

int main()

{

D1 d(100);//编译通过，输出B1 B2 D2

D1 dd;

//输出

//B1 default

//B2 default

//D2 default

return 0;

}

2.16 内联变量

见1.5

2.17 用auto作为非类型模板参数

当模板参数为非类型时，可用auto自动推导类型

如：

#include<iostream>

template<auto T>

void foo()

{

std::cout << T << std::endl;

}

int main()

{

foo<100>();//输出100

foo<int>();//no matching function for call to "foo<int>()"

return 0;

}

3 宏

3.1 \_\_has\_include

判断有没有包含某文件

如：

int main()

{

#if \_\_has\_include(<cstdio>)

printf("hehe");

#endif

#if \_\_has\_include("iostream")

std::cout << "hehe" << std::endl;

#endif

return 0;

}

4 属性

4.1 fallthrough

用于switch语句块内，表示会执行下一个case或default

如：

int main()

{

int ok1, ok2;

switch (0)

{

case 0:

ok1 = 0;

[[fallthrough]];

case 1:

ok2 = 1;

[[fallthrough]];

}

return 0;

}

4.2 nodiscard

可用于类声明、函数声明、枚举声明中，表示函数的返回值没有被接收，在编译时会出现警告。

如：

[[nodiscard]] class A {}; //该属性在这其实没用

[[nodiscard]] enum class B {}; //该属性在这其实没用

class C {};

[[nodiscard]] int foo()

{ return 10; }

[[nodiscard]] A func1() { return A(); }

[[nodiscard]] B func2() { return B(); }

[[nodiscard]] C func3() { return C(); }

int main()

{

foo();//warning: ignoring return value

func1();//warning: ignoring return value

func2();//warning: ignoring return value

func3();//warning: ignoring return value

return 0;

}

4.3 maybe\_unused

可用于类、typedef、变量、非静态数据成员、函数、枚举或枚举值中。用于抑制编译器对没用实体的警告。即加上该属性后，对某一实体不会发出“没有用”的警告。

用法例子：

[[maybe\_unused]] class A {};

[[maybe\_unused]] enum B {};

[[maybe\_unused]] int C;

[[maybe\_unused]] void fun();

结语

本次检验C++17新特性使用了GCC编译器，对于Clang的支持性方面没有做出差异测试。若有问题，欢迎指出