# 从C++11到C++20

目录

TODO:

C++11

TODO:

C++14

TODO:

C++17

TODO:

## C++20

* 1. Lambda的变化:
     1. 弃用[=]

与之代替的是[=,this]

Example:

struct lambda{

auto foo(){

return [=](int s){return n;}; //弃用警告

}

int n;

}

struct lambda1{

auto bar(){

return [=,this](int s){return n;}; //可以啦

}

int n;

}

* + 1. 模板lambda

如果你用vector这样的STL做一个lambda的话，可能是这样写的：

auto foo = [](std::vector<...> const& vec){ };

这会让代码变得麻烦。最好的方法是

auto foo = []<typename T>(std::vector<T> const& vec) {

//...

};

这样，模板类的lambda可以轻松完成。

* 1. \_\_VA\_OPT\_\_

与\_\_VA\_ARGS\_\_一起用的。

用法：

#define A(a1,...)  f(a1 \_\_VA\_OPT\_\_( , \_\_VA\_ARGS\_\_))

/\*1.\*/A(a); //替换为 f(a)

/\*2.\*/A(b,c,d); //替换为 f(b,c,d)

//\_\_VA\_OPT\_\_中的内容是\_\_VA\_ARGS\_\_非空的时候

//才会增添进代码中的。在1中，...为空，所以

//,\_\_VA\_ARGS\_\_没有出现。但2中，...有两个参数，

//所以逗号和c,d会出现。

这一新特性在宏定义和语法糖上会十分有用。

* 1. 指定初始化

struct A { int x; int y; int z; };

A a{.y = 2, .x = 1}; // 错误，没按顺序

A b{.x = 1, .z = 2}; // 可以，y被初始化为0.

特别地，union类型只能初始多个成员中的一个。

* 1. 位域初始化

众所周知，这样可以指定变量的位域：

unsigned int uint3\_t\_val : 3;

这说明uint3\_t\_val是一个三位数（二进制意义上的）其数值范围是0-7.

在C++20之前，这样

int x : 3 = 0;

是“病态的”。

现在，我们可以

int x : 3 = 0;

或者

int x : 3 {0};

* 1. const限定指针修复

其实是这样一回事……

struct X { void foo() const&; };

X{}.foo(); //可以

(X{}.\*&X::foo)(); //病态了

之前有规定，如果调用这样函数用\*运算符就病态了，但是上面这个例子中两个其实是一回事……

所以，

在C++20中两种写法都可以。（这好像是修复bug）

* 1. Concept

c++新增特性“四大金刚”之一。

它是要求某些操作合法还是不合法的方式。用法很多。

比如说，你要定义一个函数，接受一个参数food，就像这样：

template<class T>void visit(T &food){if(hungry) food.eat();}

显然，T结构体必须包含一个叫eat的函数。

但是有时这样定义多个这样的函数很麻烦，或用户（或者你）忘了或根本不知道传入的参数的这一规则，这会导致查错等极其麻烦。

来，concept走起~~~

template<typename T>concept bool edible = requires (T food) { food.eat(); };

void visit(edible &food){food.eat();}

好舒服。还有一种方法是

template<typename T>concept edible = requires (T food) { food.eat(); };

template<edible Food>void visit(Food &food){food.eat();}

还可以和constexpr连用（如果要求food.eat是constexpr的）

template<typename T>concept edible = requires (T food) { food.eat(); };

template<edible Food>

void visit(Food &food){

if(constexpr(edible<Food>)) food.eat();

else throw(error);

}

* 1. 三向比较

新增的运算符：<=>

又称飞碟运算符。需要包含<compare>以获得返回类型的库支持。

比较分三种

* + 1. 强比较：算术比较

返回

std::strong\_ordering::equal

std::strong\_ordering::less

std::strong\_ordering::greater

其中的一个。

* + 1. 弱比较：用< == >三个运算符实现的比较

返回

std::strong\_ordering::equal

std::strong\_ordering::less

std::strong\_ordering::greater

其中的一个。

* + 1. 部分比较：浮点数比较（-0==+0;）

返回

std::partial\_ordering::less

std::partial\_ordering::greater

std::partial\_ordering::equivalent

std::partial\_ordering::unordered (NaN<=>任何数 的结果都是unordered)

其中的一个。

* + 1. 其他有趣的用途

在一个class或struct中，你只要声明了<=>运算符，编译器会自动生成绝对正确的< > == <= >= !=运算符！

* 1. 基于range的init-statement

在C++20之前，你要像这样编程：

int i;

for(auto c : “hello”){i=c;cout<<i<<’ ‘<<c;}

但在C++20之后，你可以这样：

for(int i;auto c : “hello”){i=c;cout<<i<<’ ‘<<c;}

* 1. 默认可构造和可分配的无状态的lambda

简单点说，就是可以typeid和decltype的lambda……

如

decltype([&](int i){return i;}) a=[&](int i){return i;};

* 1. 从构造函数推导出模板参数类型

不用指定模板参数也可知道类型的升级版

vector v{vector{1, 2}}; // 正确，v 推导为vector<vector<int>>类型

tuple t{tuple{1, 2}}; //正确，t 推导为tuple<int, int>类型

* 1. 专门的访问检查

在模板类内，可以忽略访问权限而访问到其他类内的嵌套类。

class A{struct im1{int val;};}

class B{A::im1 a;}//error

template<typename T>

class C{A::im1 b;}//ok

* 1. constexpr函数的实例化

意思就是，假设有一个constexpr函数叫A，当调用decltype(A())时，不调用函数，仅仅取类型。

* 1. Lambda扩展

说白了就是lambda函数可以用可变参数（不同之处是三个点写在前面）

例子

template<class A, class... Args>

auto ik(A f, Args... args)

{

return [f=std::move(f), ...args=std::move(args)]() -> decltype(auto)

{

return std::invoke(f, args...);

};

* 1. ADL扩展

变量使用推断

int \_1;

void \_a();

namespace N

{

struct H {};

template<typename T> int \_0(T);

template<typename T> int \_a(T);

template<typename T> int \_1(T);

}

int x = \_0<N::H>(N::H()); // 正确

int y = \_a<N::H>(N::H()); // 正确

int z = \_1<N::H>(N::H()); // 错误

* 1. 放宽基于范围的for循环

以前基于范围的for循环中的range都是必须定义begin()和end()函数的，现在这一限制放宽了，也就是说你可以这么干：

struct S{int a,b,c,d,e;};

S s{1,2,3,4,5};

for(auto x : s){/\*...\*/}

* 1. 类类型的非类型模板参数

以前template<Type T>中Type不能是类。

现在放宽了这一限制。但要求运算能在编译期间完成。

* 1. [[likely]]和[[unlikely]]

用于指示switch分支结构的优化，likely表示“很大可能”落到指定分支，而unlikely表示“很小概率”落到指定分支。

switch(i) {

case 1: [[fallthrough]];

[[likely]] case 2: return 1;

[[unlikely]] case 3: return 2;

}

* 1. [[no\_unique\_address]]

这使得在类中，

同类型的对象不占用同一个地址；

当地址不够分配时，扩展空间，继续为未分配地址该属性成员分配地址，直至全部分配完毕；

该属性对空类型都有效。

如struct C{};struct D{int a;[[no\_unique\_address]] C b,c;};//b与c地址不同

* 1. [[nodiscard]]可以有信息

新增可选信息

[[nodiscard("hahaha")]]

char g(){return '\0';}

int main(){

g();//[Warning] ignoring return value of 'char g()', declared with //attribute 'nodiscard': 'hahaha' [-Wunused-result]

}

* 1. Using enum

enum a{e=3};

using enum a;

* 1. Explicit(bool)

当explicit(false)时explicit不启用

当explicit(true)时explicit启用。

例子：

class A{public:explicit(false) A(){}};

还可以这样用：

Template<bool ExpUse>class A{public:explicit(ExpUse) A(int n){} int n;};

* 1. 虚函数也可以constexpr

如题

* 1. char8\_t

编译器层面上对UTF-8的支持；库支持只有u8string……

const char8\_t \*a=u8”哈哈哈！！！”;

u8string u(a);

* 1. Consteval

即时函数，强制性返回常量，如

consteval int c(int x){return x\*x\*x;}

int x=1;

int y=c(x);//编译失败

int z=c(1);//编译成功

* 1. constexpr不可以用try-catch语块

如题

* 1. Constinit

常量初始化表达式。

例子：

#define Int int

constexpr Int getnum1(){return 1;}

constexpr Int getnum2(){return 2;}

Int \_3=3;

constinit Int a=getnum1();//OK

constinit Int b=getnum2();//OK

constinit Int c=\_3;//ERROR

* 1. Inline namespace

namespace A{inline namespace B{}}

等同于

namespace A::inline B{}

* 1. Module

TODO:

* 1. 协程

TODO:

* 1. 以上只是语法的改进……以下是库支持：（TODO:)