

# Cpp Version(feature) M

2022年10月13日 8:27

=====

C++ 11

[https://blog.csdn.net/weixin\\_46873777/article/details/122948389](https://blog.csdn.net/weixin_46873777/article/details/122948389)

## 1. 列表初始化

C++ 98 中, 允许 花括号 对数组元素进行统一的 列表初始值设定

```
int array1[] = {1, 2, 3, 4, 5};  
int array2[5] = {0};
```

C++ 98对于自定义类型, 无法使用列表初始化, C++ 11 可以

C++ 11 扩大了 花括号包围的列表 (即 初始化列表) 的使用范围, 使其可以用于 所有的内置类型 和用户自定义的类型, 使用 初始化列表时, 可以有等号, 也可以没有。

```
// 内置类型变量  
int x1 = {10};  
int x2{10};//建议使用原来的  
int x3 = 1+2;  
int x4 = {1+2};  
int x5{1+2};  
// 数组  
int arr1[5] {1, 2, 3, 4, 5};  
int arr2[] {1, 2, 3, 4, 5};  
// 动态数组, 在C++98中不支持  
int* arr3 = new int[5]{1, 2, 3, 4, 5};  
// 标准容器  
vector<int> v{1, 2, 3, 4, 5};//这种初始化就很友好, 不用push_back一个一个插入  
map<int, int> m{{1, 1}, {2, 2}, {3, 3}, {4, 4}};
```

支持单个对象的列表初始化

```
class Point  
{  
public:  
    Point(int x = 0, int y = 0): _x(x), _y(y) // 。。是必须这种, 还是可以不带  
    黄色的, 而是在方法体中 初始化  
    {}  
private:  
    int _x;  
    int _y;  
};  
int main()  
{  
    Pointer p = {1, 2};  
    Pointer p{1, 2};//不建议  
    return 0;  
}
```

### 多个对象的列表初始化

要给这个类(模板类)增加一个 带有 `initializer_list` 类型参数 的构造器。

注意: `initializer_list` 是系统自定义的类模板, 该类模板中主要有 3个方法: `begin()`, `end()` 迭代器 以及获取 区间中元素个数的 `size()`。

```
class Date
{
public:
    Date(int year = 0, int month = 1, int day = 1)
        :_year(year)
        , _month(month)
        , _day(day)
    {
        cout << "这是日期类" << endl;
    }

private:
    int _year;
    int _month;
    int _day;
};

int main()
{
    //C++11容器都实现了带有initializer_list类型参数的构造函数
    vector<Date> vd = { { 2022, 1, 17 }, Date{ 2022, 1, 17 }, { 2022, 1, 17 } };
    return 0;
}
```

## 2. decltype 类型推导

为什么需要 `decltype`:

`auto`的使用前提是: 必须要对 `auto` 声明的类型 进行初始化, 否则编译器 无法推导出 `auto` 的实际类型。但有时候 可能需要 根据表达式运行后的结果 的类型 进行推导, 由于编译期间, 代码不会运行, 此时 `auto` 就无能为力。

`decltype`是根据 表达式的 实际类型推导出 定义变量 所需的 类型, 如  
推导表达式作为变量的 定义类型

```
int a = 10, b = 20;
decltype(a + b)c;
cout << typeid(c).name() << endl;
```

推导函数返回值类型

```
template<class T1, class T2>
T1 Add(const T1& left, const T2& right)
{
    return left + right;
}

int main()
{
    cout << typeid(Add(1, 2)).name() << endl;
    return 0;
}
```

## 3. final 和 override

`final`修饰类时, 表示 该类不能被继承。

```
class A final //表示该类是最后一个类
{
private:
    int _year;
```

```

};

class B : public A //无法继承
{

};

final 修饰虚函数时，这个虚函数不能被重写
class A
{
public:
    virtual void fun() final//修饰虚函数
    {
        cout << "this is A" << endl;
    }

private:
    int _year;
};

class B : public A
{
public:
    virtual void fun()//父类虚函数用final修饰，表示最后一个虚函数，无法重写
    {
        cout << "this is B" << endl;
    }
};

```

#### override

检查派生类 虚函数 是否重写了 基类 的虚函数

```

class A
{
public:
    virtual void fun()
    {
        cout << "this is A" << endl;
    }

private:
    int _year;
};

class B : public A
{
public:
    virtual void fun() override
    {
        cout << "this is B" << endl;
    }
};

```

#### 4. 默认成员函数控制

C++ 对于空类 编译 会生成一些默认的 成员函数，如， 构造器，拷贝构造器，运算符重载，析构器，&的重载，const& 重载，移动构造，移动拷贝构造 等函数。

如果类中显式定义了，那么 编译器不会生成 默认版本。

有时，这样的规则会被 忘记，最常见的是 声明了 带参数的构造器，必要时 需要定义 不带参数的 版本 以实例化 无参的对象。

而且，有时 编译器会生成，有时又不生成，容易造成混乱，于是 C++ 11 让 程序员 控制 是否需要 让编译器 生成。

#### 显式缺省函数

C++ 11 中，可以在默认函数定义 或 声明时 加上 =default， 从而显式 指定 编译器 生成 该函数的 默认版本（默认成员函数）。用=default 修饰的 函数 称为 显式缺省函数。

```

class A
{
public:
    _____

```

```

A() = default; //让编译器默认生成无参构造函数
A(int year)    //这样不写缺省值的时候，就不需要自己去实现一个默认的无参构造
函数
    :_year(year)
{
void fun()
{
    cout << "this is A" << endl;
}
private:
    int _year;
};

```

### 删除默认函数

要限制某些默认函数的生成，C++ 98中是，将该函数 设置为 private，并且不给定义，这样调用时 就会 报错。

C++ 11 中，只需要 在该函数的声明上 添加 =delete 即可，这个语法 指示 编译器 不生成对于函数的 默认版本，将 =delete修饰的 函数 称为 **删除函数**

```

class A
{
public:
    A() = default;
    A(int a) : _a(a)
    {}
//C++11
// 禁止编译器生成默认的拷贝构造函数以及赋值运算符重载
    A(const A&) = delete;
    A& operator=(const A&) = delete;
private:
    int _a;
//C++98, 设置成private就可以了
    A(const A&) = delete;
    A& operator=(const A&) = delete;
};

。。。？2个同名的？不重复吗？还是为了 显示 C++ 98 ？

```

## 5. 右值引用和移动语义

左值引用和右值引用

C++ 11 新增 右值引用。

无论 左值引用 还是 右值引用，都是给对象 取别名（与 对象共享一片空间）。

### 左值

左值是一个 表示 数据的 表达式（如变量名 和 解引用的指针）， 我们可以获取它的地址，也可以对它赋值，左值可以出现在 赋值符号的 左边， 右值不可以出现在 左边。

左(。。。值?)引用 加const修饰，不能 对其赋值，但可以取地址，是一种特殊情况

左值引用 就是给左值取别名。

//以下都是左值

```

int* p = new int[10];
int a = 10;
const int b = 20;
//对左值的引用
int*& pp = p;
int& pa = a;
const int& rb = b;

```

### 左值

可以取地址

一般情况下可以修改(const修饰时 不能修改)

### 右值

右值也是一个表示 数据的 表达式，如：字面常量，表达式返回值，传值返回函数的返回值

(不能是 左值引用返回) 等。

右值可以出现在 赋值号 的右边，但不能出现在 左边。

右值引用 就是 给 右值 取别名。

```
double x = 1.1, y = 2.2;
//常见右值
10; // ...
x + y; // ...
add(1, 2);
//右值引用
int&& rr1 = 10;
double&& rr2 = x + y;
double && rr3 = add(1, 2);
//右值引用一般情况不能引用左值，可使用move将一个左值强制转化为右值引用
int &&rr4 = move(x);
//右值不能出现在左边，错误
10 = 1;
x + y = 1.0;
add(1, 2) = 1;
```

**move:** 当需要用 右值引用 引用一个左值时， 可以通过 move 函数 将左值转化为 右值。

C++ 11 中， std::move() 函数 的名字有一定的迷惑性， 它并不 move 任何东西，只是将一个 左值 强制 转化为 右值引用，来实现 移动语义。

## 左值引用和 右值引用比较

左值引用总结：

左值引用只能引用左值， 不能引用右值  
const左值引用 既可以引用左值， 也可以引用右值

```
// 左值引用只能引用左值，不能引用右值
int a = 10;
int& ra1 = a; // ra为a的别名
//int& ra2 = 10; // 编译失败，因为10是右值

//const左值引用既可以引用左值，也可以引用右值
const int& ra3 = 10;
const int& ra4 = a;
```

右值引用总结：

右值引用只能引用右值， 一般情况下不能引用左值  
右值引用可以应用move 以后的 左值

```
int a = 10;
int b = 20;
//不能引用左值
//int&& rr1 = a;
int&& rr2 = 10;
int&& rr3 = move(a); //强制转换为右值引用
```

## 右值引用场景和意义

左值引用 既可以引用左值，也可以引用右值， 为什么C++ 11 还需要 右值引用？ 因为左值引用存在 短板，下面来看 这个短板，以及 右值引用 如何 弥补这个短板

```
void fun1(bit::string s)
{}
void fun2(bit::string& s) // ... bit 是什么命名空间？搜不到，应该是自定义的
{}
int main()
{
    bit::string s("1234");
    //fun1(s);
    //fun2(s); //左值引用提高了效率，不存在拷贝临时对象的问题
```

```

//可以使用左值引用返回，这个对象还在
s += 'a';
//不能使用左值引用返回，这个就是左值引用的一个短板
//函数返回对象出了作用域就不在了，就不能用左值引用返回（因为返回的是本身地址，栈帧已销毁）
//所以会存在拷贝问题
bit::string ret = bit::to_string(1234);
return 0;
}

```

。。是的，`bit`是自定义的，因为运行以后，命令行打印了深拷贝，应该是将各种默认的函数都写出来了，然后加了`cout`来看发生了什么。

右值引用 弥补了这个短板

C++ 移动语义提出：将一个对象中资源移动到另一个对象中的方式

`str`在值返回时，必须创建一个临时对象，临时对象创建好之后，`str`就被销毁了，`str`是一个将亡值，C++ 11 认为它是右值，在用`str`构造临时对象时，就会采用移动构造，即将`str`中的资源转移到临时对象中。

而零食对象也是右值，因此在用临时对象构造`s3`时，也使用移动构造，将临时对象中的资源转移到`ret`中，整个过程，只需要创建一个块堆内存即可，节省了空间，且提高了运行效率

```

V

bit::string to_string(int value)
{
    bit::string str;
    // ...
    return str; // 拷贝构造
}

int main()
{
    bit::string ret1 = bit::to_string(1234); // 本来是两次拷贝构造，然后这种在一个调用中，连续构造
    return 0;
}

bit::string to_string(int value)
{
    bit::string str;
    // ...
    return str; // 拷贝构造
}

int main()
{
    bit::string ret1 = bit::to_string(1234); // 将其移动到 ret1 中
    return 0;
}

```

CSDN @雨轩 (爵丶迹)

这里，我们就可以对右值进行定义：

纯右值： 10, a+b

将亡值： 函数返回的临时对象，匿名对象。

我们将这条语句分开写，会发生什么？

```

bit::string ret;
ret = bit::to_string(1234); // 赋值重载多了一次拷贝构造

```

```

bit::string to_string(int value)
{
    bool flag = true;
    if (value < 0)
    {
        ...
    }

    std::reverse(str.begin(), str.end());
    return str;
} // 拷贝构造

int main()
{
    bit::string ret1;
    // ...
    ret1 = bit::to_string(1234);
    return 0;
}

// Microsoft Visual Studio 调试控制台
string(string&& s) -- 移动构造
string& operator=(string s) -- 深拷贝
string(const string& s) -- 深拷贝

现代写法operator=复用了拷贝构造
string& operator=(const string& s)
{
    cout << "string& operator=(string s) -- 深拷贝" << endl;
    string tmp(s);
    swap(tmp);
    return *this;
}

```

CSDN @雨轩 (爵、迹)

我们将移动赋值 写上，就会进行优化，少一次拷贝构造

```

bit::string to_string(int value)
{
    bool flag = true;
    if (value < 0)
    {
        ...
    }

    std::reverse(str.begin(), str.end());
    return str;
} // 移动构造

int main()
{
    bit::string ret1;
    // ...
    ret1 = bit::to_string(1234);
    return 0;
}

// 移动赋值
string& operator=(string&& s)
{
    cout << "string& operator=(string&& s) -- 移动赋值" << endl;
    this->swap(s);
}

bit::string ret;
ret = bit::to_string(1234);

// Microsoft Visual Studio 调试控制台
string(string&& s) -- 移动构造
string& operator=(string&& s) -- 移动赋值

```

CSDN @雨轩 (爵、迹)

总结一下：右值引用出来以后，并不是直接使用右值引用去减少拷贝，提高效率。而是支持深拷贝的类，提供移动构造和移动赋值，这时这些类的对象进行值返回或参数为右值时，可以用移动构造和移动赋值，转移资源，避免深拷贝，提高效率。

## 场景2

```

//左值, 拷贝构造, 使用左值引用
list<bit::string> lt;
bit::string s("1234");
lt.push_back(s);
//以下传的都是右值, 右值引用, 所以是移动构造
lt.push_back("123");
lt.push_back(bit::string("2121"));
lt.push_back(std::move(s));

```

```

//左值, 拷贝构造
list<bit::string> lt;
bit::string s("1234");
lt.push_back(s);
//以下传的都是右值, 所以是移动构造
lt.push_back("123");
lt.push_back(bit::string("2121"));
lt.push_back(std::move(s));
return 0;

// Microsoft Visual Studio 调试控制台
string(const string& s) -- 深拷贝
string(string&& s) -- 移动构造
string(string&& s) -- 移动构造
string(string&& s) -- 移动构造
E:\C++复习(比特)\String\Debug要在调试停止时自动关闭控制台, 请按任意键关闭此窗口...

```

CSDN @雨轩 (爵、迹)

总结一下，右值引用使用场景2，可以在 容器中插入接口函数中，如果实参是右值，则可以转移它的资源，减少拷贝。

## 6. 完美转发

完美转发 是指 函数模板中，完全依照 模板的 参数的类型，将参数传递给 函数模板中 调用的 另一个函数

```
void Func(int x)
{
    cout << x << endl;
}

template<typename T>
void PerfectForward(T&& t)
{
    Func(t);
}
```

PerfectForward 是转发的 模板函数，Func 为实际目标函数，  
但是 上述转发还不算完美，

完美转发是 目标函数 总希望 将参数 按照传递给转发函数的 实际类型 转给目标函数，而  
不产生额外的开销，就好像 转发者不存在一样。

所谓完美：函数模板在向 其他函数 传递自身形参时，如果相应 实参 是左值，它就应该被  
转发 为 左值；如果相应参数 是右值，它就 应该被转发为 右值。这样做 是为了保留在  
其他 函数针对 转发而来的 参数的 左右值 属性进行不同处理（比如， 参数为左值时 实施  
拷贝语义，参数为右值时 实施移动语义）

## 万能引用

模板中的 && 不代表右值引用，而是 万能引用，既能接受左值，又能接受右值。

模板中的 万能引用 只是提供了 能够 同时接受 左值ref，右值ref 的能力

但是引用类型的 唯一作用就是 限制了 接受的类型，后续使用中 都退化成了 左值

我们希望 能够在 传递过程中，保持它的 左值 或 右值 属性，就需要 使用下面的 完美转发。

C++ 通过 forward 来实现 完美转发

```
void Func(int& x) { cout << "左值引用" << endl; }
void Func(const int& x) { cout << "const 左值引用" << endl; }

void Func(int&& x) { cout << "右值引用" << endl; }
void Func(const int&& x) { cout << "const 右值引用" << endl; }
template<typename T>
void PerfectForward(T&& t)
{
    //Func(t); //没有使用forward保持其右值的属性，退化为左值
    Func(forward<T>(t));
}

int main()
{
    PerfectForward(1); //右值
    int a = 10;
    PerfectForward(a);
    PerfectForward(move(a));

    const int b = 20;
    PerfectForward(b);
    PerfectForward(move(b));
    return 0;
}
```

右值ref的对象，作为实参传递时，会退化为左值，只能匹配左值ref。使用完美转发，可以保持它的右值属性。

## 7. 新的类功能

默认成员函数

原先C++类中，有6个默认成员函数

- 构造函数
- 析构函数
- 拷贝构造函数
- 拷贝赋值重载
- 取地址重载
- const取地址重载

默认成员函数就是我们不写，编译器会生成一个默认的

C++ 11 新增2个：移动构造函数 和 移动赋值运算符重载

如果你没有自己实现移动构造函数，且没有实现析构器，拷贝构造，拷贝赋值重载中的任意一个。那么编译器会自动生成一个默认移动构造。默认生成的移动构造函数，对于内置类型成员会逐成员按字节拷贝，自定义类型成员，则需要看这个成员是否实现了移动构造，实现了就使用移动构造，没有实现就调用拷贝构造。

如果你没有自己实现移动赋值重载函数，且没有实现析构函数，拷贝构造，拷贝复制重载中的任意一个，那么编译器会生成一个默认移动赋值。对于内置类型成员逐字节拷贝，对于自定义类型成员，如果实现了移动赋值就使用移动赋值，没有实现就使用拷贝赋值。（默认移动赋值和上面的移动构造完全类似）

如果你提供了移动构造或移动赋值，编译器不会自动提供拷贝构造和拷贝赋值。

## 8. lambda表达式

之前，我们要比较自定义类型的大小，需要自己实现一个类，并写上仿函数。

```
struct Goods
{
    string _name;
    double _price;
};

struct Compare
{
    bool operator()(const Goods& g1, const Goods& gr)
    {
        return g1._price <= gr._price;
    }
};

int main()
{
    Goods gds[] = { {"苹果", 2.1}, {"香蕉", 3}, {"橙子", 2.2}, {"菠萝", 1.5} };
    sort(gds, gds+sizeof(gds) / sizeof(gds[0]), Compare());
    return 0;
}

lambda:
int main()
{
    Goods gds[] = { {"苹果", 2.1}, {"香蕉", 3}, {"橙子", 2.2}, {"菠萝", 1.5} };
    sort(gds, gds + sizeof(gds) / sizeof(gds[0]), [](const Goods& l, const Goods& r)
        ->bool
    {

```

```

        return l._price < r._price;
    });
return 0;
}

```

### lambda语法

`[capture-list] (parameters) mutable -> return-type { statement }`

**[capture-list]:** 捕捉列表，总是出现在 lambda 函数的开始位置，编译器根据 [] 来判断接下来的代码是不是 lambda 函数，

**(parameters):** 参数列表，和普通函数的参数列表一致，如果不需要参数传递，则可以连同()一起省略。

**mutable:** 默认情况下，lambda 函数 总是一个 const 函数，mutable 可以取消其常量性。使用这个修饰符时，参数列表不可省略(即使为空)

**->return-type:** 返回值类型。没有返回值时可以省略。有返回值时也可以省略，由编译器对返回类型进行推导。

**{statement}:** 函数体。函数体中除了可以使用参数外，还可以使用所有捕获到的变量。

在lambda定义中，参数列表和返回值类型是可选的，而捕捉列表和函数体可以为空。  
所以C++ 11 中最简单的 lambda函数是 [] {}；

```

int main()
{
    [] {};//最简单的lambda表达式上面也不做
    //省略参数列表和返回值类型，返回值类型由编译器推导为int
    int a = 3, b = 4;
    [=] {return a + 3; };

    //省略了返回值类型，无返回值类型
    //引用传递捕捉a 和 b变量
    auto fun1 = [&](int c) {b = a + c; };
    fun1(10);
    cout << a << " " << b << endl;

    //各部分都很完善的lambda函数
    //引用方式捕捉b，值传递捕捉其他所有变量
    auto fun2 = [=, &b](int c) ->int {return b += a + c; };
    cout << fun2(10) << endl;

    //值传递捕捉x
    int x = 10;
    auto add_x = [x](int a) mutable { x *= 2; return a + x; };
    cout << add_x(10) << endl;
    return 0;
    return 0;
}

```

### 捕获列表

描述了上下文中哪些数据可以被 lambda 使用，以及是传值使用还是传址使用。

**[var]:** 表示值传递的方式捕获变量 var

**[=]:** 表示值传递的方式捕获所有父作用域中的变量 (包括 this)

**[&var]:** 表示引用传递捕获变量 var

**[&]:** 表示引用传递捕获所有父作用域中的变量 (包括 this)

父作用域指包含 lambda 函数的语句块。

语法上 捕捉列表可以由多个捕捉项组成，并以逗号分隔。

**[=,&a,&b] :** 以引用传递的方式捕获变量a 和 b，值传递的方式捕获所有其他变量。

[&, a, this]: 值传递的方式捕捉变量 a 和this，引用方式捕捉其他变量。

```
auto fun1 = [&](int c) {b = a + c; };
fun1(10);
cout << a << " " << b << endl;
```

捕捉列表 不允许 变量重复传递，会导致 编译错误，如[=, a]，=已经以值传递的方式捕捉了所有变量，捕捉a 是重复的。

在快作用域以外 的lambda 函数 捕捉列表 必须为空

在块作用域 内的 lambda 函数 仅能捕捉 父作用域中的 局部变量，捕捉 任何 非此作用域或 非局部变量 都会 编译错误。

lambda之间 不能互相赋值

## 9. 可变参数列表

C++11 新特性：可变参数模板 能够让你 创建可以接受 可变参数的 函数模板 和 类模板，相比 C++ 98/03，类模板 和 函数模板 中 只能包含固定数量 模板参数，可变模板参数是一个巨大的进步。

```
//可变参数，你传int, char, 还是自定义都会自动给你推导
//可以包含0-任意个参数
template<class ...Args>
void ShowList(Args... args)
{
    cout << sizeof... (args) << endl;//计算个数
}
int main()
{
    ShowList(1, 2, 3);
    ShowList(1, 'a');
    ShowList(1, 'A', string("sort"));
    return 0;
}
```

如何取值

```
//需要加上结尾函数
void ShowList()
{
    cout << endl;
}
template <class T, class ...Args>
void ShowList(T value, Args... args)
{
    cout << value << " ";
    ShowList(args...); //不断调用自己，直到最后参数为空，调用上面的结尾函数
}
```

可变参数在列表初始化的应用

```
template<class ...Args>
void ShowList(Args... args)
{
    int arr[ ] = { args... };//可变参数初始化列表
    cout << endl;
}
```

如何传递不同类型的 数据？

C++11 利用逗号表达式 调用 另外一个函数，最后的0留给数据。

```

template <class T>
void PrintArg(T t)
{
    cout << t << " ";
}
template <class ...Args>
void ShowList(Args... args)
{
    // 列表初始化
    // {(printarg(args), 0)...} 将会展开成 ((printarg(arg1), 0), (printarg(arg2), 0),
    // (printarg(arg3), 0), etc...)
    int arr[] = { (PrintArg(args), 0)... };
    cout << endl;
}
int main()
{
    ShowList(1, 2, 3);
    ShowList(1, 'a');
    ShowList(1, 'A', string("sort"));
    return 0;
}

```

可以给模板函数设置返回值

```

template <class T>
int PrintArg(T t)
{
    cout << t << " ";
    return 0;
}

template <class ...Args>
void ShowList(Args... args)
{
    // 列表初始化
    // 也可以给模板函数设置一个返回值
    int arr[] = { PrintArg(args)... };
    cout << endl;
}

```

可变参数包 结合完美转发 的好处

直接就是 普通构造器的形式，不存在 移动构造 或 深拷贝构造，节省空间。

if such a call is well formed. Otherwise, it invokes:  
`::new (static_cast<void*>(p)) T (forward<Args>(args)...)`

A specialization for a specific allocator type may provide a different definition.

// 下面我们试一下带有拷贝构造和移动构造的`bit::string`, 再试试呢  
// 我们会发现其实差别也不到，`emplace_back`是直接构造了，`push_back`  
// 是先构造，再移动构造，其实也还好。  
`std::list<std::pair<int, bit::string> > mylist;`  
`std::pair<int, bit::string> kv(20, "sort");`  
`mylist.emplace_back(kv); // 左值 存在深拷贝`  
`mylist.emplace_back(std::pair<int, bit::string>(20, "soft")); // 右值 移动构造`  
`mylist.emplace_back(10, "sort"); // 就是普通构造函数 // 参数包`

```

D:\课堂代码\就业课100期\Debug\C++11.exe
string(char* str)
string(const string& s) -- 深拷贝
string(char* str)
string(char* str)
string(string&& s) -- 移动构造
string(char* str)

```

## 10. 包装器

函数包装器 其实就是 函数指针，用了包装器以后， 函数模板 只会实例化一次。

可调用对象的类型： 函数指针， 仿函数， lambda

```
// 函数模板会被实例化多次
template<class F, class T>
T useF(F f, T x)
{
    static int count = 0;
    cout << "count:" << ++count << endl;
    cout << "count:" << &count << endl;
    return f(x);
}

double func(double i)
{
    return i / 2;
}

struct Functor
{
    double operator()(double d)
    {
        return d / 3;
    }
};

int main()
{
    // 函数名
    cout << useF(func, 11.11) << endl;
    // 函数对象
    cout << useF(Functor(), 11.11) << endl;
    // lambd表达式
    cout << useF([](double d){ return d / 4; }, 11.11) << endl;
    return 0;
}
```

可以看到 静态变量 count ， 每次的地址 都不一样，说明 函数模板 实例化了 3次。

我们可以通过 包装器 让函数模板 只 实例化 一次。

```
int main()
{
    // 函数名 生成一个函数包装器， f1就是函数指针 ==  double (*f1)(double)
    std::function<double(double)> f1 = func;
    cout << useF(f1, 11.11) << endl;

    // 函数对象
    std::function<double(double)> f2 = Functor();
    cout << useF(f2, 11.11) << endl;

    // lambd表达式
    std::function<double(double)> f3 = [](double d){ return d / 4; };
    cout << useF(f3, 11.11) << endl;

    return 0;
}
```

可以看到 count 的值 是累加的，说明函数模板 只实例化了一次。

`std::function` 包装 各种可调用的 对象，统一可调用对象 类型，并且 指定了 参数 和 返回值类型。

为什么有 `std::function`， 因为不包装前 可调用类型 存在很多问题

函数指针 类型 太复杂，不方便使用和理解

仿函数类型 是一个类名，没有 指定 调用参数 和返回值。得去看 `operator()` 的实现 才能看出来。

`lambda` 表达式在语法层，看不到类型。底层有类型，基本都是 `lambda_uuid`，也很难 看。

---

[https://blog.csdn.net/AI\\_ELF/article/details/125781783](https://blog.csdn.net/AI_ELF/article/details/125781783)

C++ 11

扩大了 花括号包围的 列表(初始化列表) 的使用范围。使其可以用于 所有的内置类型 和 用户自定义的类型， 使用初始化列表时，可以添加 等号，也可以不添加。

```
std::initializer_list  
auto il = {1,2,3};  
cout<<typeid(il).name()<<endl;  
输出 class std::initializer_list<int>
```

`initializer_list` 一般是作为 构造器的参数， C++11 对STL 中的 不少容器 增加了 `initializer_list` 作为参数的构造器。

也可以作为 `operator=` 的参数，这样就可以使用 大括号赋值。

C++98 中 `auto` 是 存储类型的说明符，表明 变量 是 局部自动存储类型， 但是局部域中定义的局部变量 默认就是自动存储类型，所以 `auto` 就没有什么价值。

C++11 中废弃了 `auto`原来的用法， 将其用于实现 自动类型 推导。这样要求 必须进行 显示 初始化，让编译器决定 对象的 类型设置 为初始化值的类型。

`decltype`  
`typeid(变量, 类).name()` 获得的是 `string`。不能作为类型。  
`decltype` 将变量的类型 声明为 表达式的类型

```
int i = 10;  
auto p = &i;  
auto pf = strcpy;  
decltype(pf) pf1; //char * (__cdecl*)(char *, char const *)  
vector<decltype(pf)> v;  
  
cout << typeid(p).name() << endl;  
cout << typeid(pf).name() << endl;
```

C++ 中的NULL 被定义为 字面量0，带来一些问题，因为 0既能表示 指针常量，也能表示 整型常量。

C++11 新增 `nullptr`，表示空指针

C++ 11 在 STL 中引入了

```
<array>  
<forward_list>  
<unordered_map>  
<unordered_set>
```

```
array<int, 10> a1;
int a2[10];
cout<<&a1<<" "<<a2<<endl;
a1, a2 没有太大区别，都是从 栈上 开辟空间。
a1 越界会报错，a2不会。
```

forward\_list 就是单链表  
支持向后插入(insert\_after)，向后删除(erase\_after)，不支持 向前删除和插入。  
forward\_list 支持 单向迭代器

基本每个容器都增加了一些 C++11 的方法，但是用的都比较少，比如 cbegin, cend, 提供 const 迭代器。

C++更新后，容器中增加了 右值引用版本

```
std::vector::emplace_back(Args&&... args);
```

针对旧容器，基本都增加移动构造，移动赋值，右值ref版本。

右值引用

完美转发

模板中的 && 万能引用

默认成员函数

构造，析构，拷贝构造，拷贝赋值，取地址重载，const 取地址重载  
移动构造，移动赋值

类成员变量初始化

C++11 允许在类定义时 给成员变量 初始缺省值，默认生成构造函数 会使用这些缺省值 初始化。

static 不能给缺省值，必须去 类 外 面 定义初始化  
const static 可以给值初始化(在类内)。并且不是缺省值。

强制生成默认函数的关键字default

C++11 可以让你更好地控制 要使用的 默认函数。假设你要使用某个默认的 函数，但是因为一些原因 这个函数没有被默认生成。

比如，我们提供了 拷贝构造，就不会生成 移动构造。

我们可以使用 default 关键字 显式 指定 生成 移动构造

```
class Person
{
public:
    Person(const char* name = "", int age = 0)
        :_name(name)
        , _age(age)
    {}
    Person(const Person& p)
        :_name(p._name)
        , _age(p._age)
    {}
    Person(Person&& p) = default;
private:
    std::string _name;
    int _age;
};
int main()
{
    Person s1;
    Person s2 = s1;
```

```
Person s3 = std::move(s1);
return 0;
}
```

禁止生成默认函数的关键字 delete

final

override

可变参数模板

C++ 98/03，类模板和函数模板中只能包含固定数量的模板参数。

C++ 11 特性可变参数模板 能够创建可以接受可变参数的函数模板 和 类模板。

```
template <class ...Args>
void ShowList(Args... args)
{}
```

Args是一个模板参数包，args是一个函数形参参数包；

声明一个参数包Args...args，这个参数包中可以包含0到任意个模板参数。

我们无法直接获取参数包 args 中的每个参数，只能通过展开参数包的方式来获取参数包中的每个参数。语法不支持 args[i] 来获取可变参数。

可以通过 sizeof...(args) 来获取多少个参数，但是无法 args[i]。  
。。。不知道 sizeof 还是 sizeof...。不知道是不是笔误。。

递归函数展开参数包

```
// 递归终止函数
template <class T>
void ShowList(const T& t) {
    cout << t << endl;
}
// 展开函数
template <class T, class ...Args>
void ShowList(T value, Args... args) {
    cout << value << " ";
    ShowList(args...);
}
int main()
{
    ShowList(1);
    ShowList(1, 'A');
    ShowList(1, 'A', std::string("sort"));
    return 0;
}
```

逗号表达式展开参数包

PrintArg 不是递归终止函数，只是一个处理参数包中每个参数的函数。

这种原地展开参数包的方式实现的关键是逗号表达式。

逗号表达式会按顺序执行逗号前面的表达式。

```
template <class T>
void PrintArg(T t) {
    cout << t << " ";
}
// 展开函数
template <class ...Args>
void ShowList(Args... args) {
```

```

    int arr[] = { (PrintArg(args), 0)... };
    cout << endl;
}
int main()
{
    ShowList(1);
    ShowList(1, 'A');
    ShowList(1, 'A', std::string("sort"));
    return 0;
}

```

逗号表达式: (PrintArg(args), 0), 也是按照这个执行顺序, 先执行 PrintArg(args), 再得到逗号表达式的结果 0. 同时还使用了 C++11 的另一个特性: 初始化列表, 通过初始化列表来初始化一个变长数组, {(PrintArg(args), 0)...} 会被展开为  
 $((PrintArg(arg1), 0), (PrintArg(arg2), 0), (PrintArg(arg3), 0) \dots)$   
最终会创建一个元素值都是0的数组 int arr[sizeof... (Args)]

由于是逗号表达式, 在创建数组的过程中会先执行逗号表达式前面的表达式 PrintArg(args) 打印出参数, 也就是说构造 int 数组的过程中就将参数包展开了, 这个数组的目的纯粹是为了在数组构造的过程中展开参数包。

也可以使用 int arr[] = { PrintArg(args)... };

emplace\_back, push\_back

push\_back 是构造+移动构造, emplace\_back 是直接构造

如果元素对象及其成员都实现了移动构造, 本质区别不大, 因为构造出来 + 移动构造, 和直接构造成本差不多。

如果元素对象及其成员没有实现移动构造, 那么 emplace\_back 还是直接构造, push\_back 是构造+拷贝构造 (没有移动构造时, 编译器会调用拷贝构造), 代价大。

lambda

包装器

function 包装器, 也叫做适配器, 本质是一个类模板, 也是一个包装器

bind

std::bind, 是一个函数模板, 就像一个函数包装器(适配器), 接受一个可调用对象 (callable object), 生成一个新的可调用对象来“适应”原对象的参数列表。

一般而言, 用它可以把一个原本接受N个参数的函数 fn, 通过绑定一些参数, 返回一个接受M个(M可以大于N, 但这么做没有什么意义)参数的新函数。

使用 bind 还可以实现参数顺序调整

```

//头文件<functional>
// 原型如下:
template <class Fn, class... Args>
/* unspecified */ bind (Fn& fn, Args&&... args);
// with return type (2)
template <class Ret, class Fn, class... Args>
/* unspecified */ bind (Fn& fn, Args&&... args);

```

可以将 bind 函数看作是一个通用的函数适配器, 它接受一个可调用对象, 生成一个新的可调用对象来“适应”原对象的参数列表。

调用 bind 的一般形式: auto newCallable = bind(callable, arg\_list);

我们调用 newCallable 时, newCallable 会调用 callable, 并传给它 arg\_list 中的参数。arg\_list 中的参数可能包含形如 \_n 的名字, 其中 n 是整数, 这些参数是占位符, 表示 newCallable 的参数, 它们占据了传递给 newCallable 的参数的位置, 数值 n 表示生成的可调用对象中参数的位置, \_1 为 newCallable 的第一个参数, \_2 是第二个。

```

int Plus(int a, int b) {
    return a + b;
}
class Sub
{
public:
    int sub(int a, int b)
    {
        return a - b;
    }
};
int main()
{
    //表示绑定函数plus 参数分别由调用 func1 的第一, 二个参数指定
    // 使用bind进行优化

    std::function<int(int, int)> func1 = std::bind(Plus, placeholders::_1,
                                                    placeholders::_2);
    //auto func1 = std::bind(Plus, placeholders::_1, placeholders::_2);
    //func2的类型为 function<void(int, int, int)> 与func1类型一样
    //表示绑定函数 plus 的第一, 二为: 1, 2
    // 需要绑定的参数, 直接绑定值, 不需要绑定的参数给 placeholders::_1 、
    placeholders::_2.... 进行占位
    function<int()> func2 = bind(Plus, 1, 2);

    cout << func1(1, 2) << endl;
    cout << func2() << endl;
    Sub s;
    // 绑定成员函数
    std::function<int(int, int)> func3 = std::bind(&Sub::sub, s, placeholders::_1,
                                                    placeholders::_2);
    // 参数调换顺序
    std::function<int(int, int)> func4 = std::bind(&Sub::sub, s, placeholders::_2,
                                                    placeholders::_1);
    cout << func3(1, 2) << endl;
    cout << func4(1, 2) << endl;
    return 0;
}

```

## 线程库

C++11之前, 涉及到多线程时, 都是和平台相关的, 比如 windows 和 linux 下各有 自己的接口。代码的可移植性比较差。

C++11中 对线程进行了 支持。 是的 C++ 在并行编程时 不需要 依赖第三方库。  
还引入了 原子类。

<https://cplusplus.com/reference/thread/thread/>

头文件<thread>

thread()	构造一个线程对象, 没有关联任何线程函数, 即没有启动任何线程
thread(fn, arg1, arg2, ...)	构造一个线程对象, 并关联线程函数fn, arg1, arg2 是线程函数的 参数
get_id()	获得线程id
joinable()	线程是否还在执行, joinable代表一个 正在执行中的线程
join()	该函数调用后 会阻塞主线程, 该线程结束后, 主线程继续
detach()	在创建线程对象后 马上调用, 用于把 被创建的线程 和 线程对象 分离开, 分离的线程变为 后台线程, 创建的线程的 生命周期 和 主线程无

```

int main()
{
    int n = 2;
    //cin >> n;
    vector<thread> works(n);
    std::mutex mtx;
    size_t x = 0;
    size_t total_time = 0;
    for (auto& thd : works)
    {
        thd = thread([&mtx, &x, &total_time] () {
            size_t begin = clock();
            mtx.lock();
            for (int i = 0; i < 100000; ++i)
            {
                ++x;
            }
            size_t end = clock();
            total_time += (end - begin);
            mtx.unlock();
        });

        cout << this_thread::get_id() << ":" << end - begin << endl;
    });

    for (auto& thd : works)
    {
        thd.join();
    }

    cout << x << endl;
    cout << total_time << endl;
}

return 0;
}

```

当创建一个线程对象后，并且给线程关联线程函数，该线程就被启动，与主线程一起运行。  
线程函数一般可以通过下面4种方式提供

函数指针

lambda

函数对象

包装器

```

void ThreadFunc(int a) {
    cout << "Thread1" << a << endl;
}

void T()
{
    cout << "hello" << endl;
}

class TF
{
public:
    void TT()
    {
        cout << "NI" << endl;
    }
}

```

```

    }
    void operator()()
    {
        cout << "Thread3" << endl;
    }
};

int main()
{
    // 线程函数为函数指针
    thread t1(ThreadFunc, 10);

    // 线程函数为lambda表达式
    thread t2([] {cout << "Thread2" << endl; });

    // 线程函数为函数对象
    TF tf;
    thread t3(tf);
    thread t5(&TF::TT, TF());

    //线程函数为包装器
    function<void()> t = T;
    thread t4(t);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
    cout << "Main thread!" << endl;
    return 0;
}

```

thread 是 防 拷贝的，不允许拷贝构造以及赋值，但是可以 移动构造 和 移动赋值，即将一个线程对象 关联线程 的 状态转移给其他线程对象，转移期间 不影响 线程的执行。

可以通过 `joinable()` 来判断 线程是否有效，如果是以下任意情况，则线程无效

采用无参构造器构造的线程对象

线程对象的状态已经转移给其他线程对象

线程已经调用 `join` 或 `detach` 结束。

线程函数的参数 是以值拷贝的方式 拷贝到线程栈空间的，因此：即使线程参数 为引用类型，在线程中 修改后 也不能修改 外部实参，因为 其实际引用的是 线程中的 拷贝，而不是 外部实参。

如果是 类成员函数 作为 线程参数时， 必须将 `this` 作为线程函数参数。

## 原子操作

```

atomic<size_t> x = 0;
atomic<T> t;      // 声明一个类型为T的原子类型变量t

```

原子类型通常属于 资源型数据，多个线程 只能访问 单个原子类型的 拷贝，因此在 C++11 中，原子类型只能从 其模板参数中 进行构造，不允许 原子类型 进行 拷贝构造，移动构造，`operator=` 等。为了防止意外，标准库 已经将 `atomic` 模板类中的 拷贝构造，移动构造，赋值运算符重载 默认删除了。

## lock\_guard 与 unique\_lock

多线程环境下，如果要保证 某个变量的 安全性，只要将其设置为 对应的原子类型 即可，高效且不容易出现死锁。

但有些情况下，需要确保 一段代码的安全性，只能通过 锁 来进行控制。

C++11使用 RAI<sup>I</sup> 的方式 对锁进行了 封装，即 `lock_guard` 和 `unique_lock`。

C++11中，Mutex 总共有4 种互斥量

```
std::mutex  
std::recursive_mutex  
std::timed_mutex  
std::recursive_timed_mutex
```

lock\_guard 通过RAII 的方式，对其管理的 互斥量 进行封装。

缺点，太单一，用户没有办法对该锁进行控制，因此C++11 有了 unique\_lock

=====

=====

=====

=====

C++ 14

<https://www.cnblogs.com/lvdongjie/p/16320487.html>

## 1. 变量模板

在C++11及之前，我们只有针对类和函数的模板，C++14中，新增了 变量模板

```
template<class T>  
constexpr T pi = T(3.1415926535897932385L);  
  
template<class T>  
T circular_area(T r)  
{  
    return pi<T> *r * r;  
}
```

变量模板同样可以在 类变量中使用:

```
template<class T>
class X {
    static T s;
};

template<class T>
T X<T>::s = 0;
```

类似函数和类模板，当变量模板被引用时，则会发生实例化

## 2. lambda新增功能

支持在 lambda 中使用auto

```
auto glambda = [](auto& a) { cout << a << " "; };
int a[] = { 4, 2, 6, 3, 7, 5 };
for_each(a, a + sizeof(a) / sizeof(int), glambda);
cout << endl;
```

对捕获的变量和 引用 进行初始化

```
int main()
{
    int x = 4;
    auto y = [&r = x, x = x + 1]()>int
    {
        r += 2;
        return x * x;
    }();
    cout << "x = " << x << " y = " << y << endl;
}
```

## 3. constexpr 函数可以包含多个语句

C++11中，如果要使用 constexpr，则只能包含一个返回语句。

C++14中，允许 constexpr 中声明 变量，使用循环和条件语句 等。

```
#include <iostream>
#include <cmath>
using namespace std;
constexpr bool isPrimitive(int number)
{
    if (number <= 0)
    {
        return false;
    }
    for (int i = 2; i <= sqrt(number) + 1; ++i)
    {
        if (number % i == 0)
        {
            return false;
        }
    }
    return true;
}
int main()
{
    cout << boolalpha << isPrimitive(102) << " " << isPrimitive(103);
    return 0;
}
```

C++11中，我们一般需要通过 递归来实现 相同的功能

```
constexpr bool isPrimitive(int number, int currentFactor, int maxFactor)
```

```

{
    return currentFactor == maxFactor ? true : (number % currentFactor == 0 ?
        false : isPrimitive(number, currentFactor + 1, maxFactor));
}
constexpr bool isPrimitive(int number)
{
    return number <= 0 ? false : isPrimitive(number, 2, sqrt(number) + 1);
}

```

#### 4. 整型字面量

二进制字面量

支持**0b**开头的一串数组 作为二进制表示的 整型

```
int a = 0b10101001110; // 1358
```

数字分隔符

使用单引号进行分割。编译时，这些单引号会被忽略

```
int a = 123'456'789; // 123456789
```

#### 5. 返回类型自动推导

C++14中，可以使用 auto 作为函数返回值 并且不需要 指明其返回类型的 推导表达式

```
int x = 1;
auto f() { return x; }
/* c++11
auto f() -> decltype(x) { return x; }
*/
```

这种类型推导有一些限制：

一个函数中所有返回子句 必须推导出相同的类型。

使用 {} 包裹 的数据 作为返回值时， 无法推导类型

**虚函数** 和 **coroutine** 不能被推导

函数模板中可以 使用类型推导，但是 其 显式 实例化 和 特化版本 必须使用相同的  
返回类型描述符。

#### 6. exchange

用于移动语义，可以使用指定的新值替换 原值，并返回原值。

其定义在C++20中被 简单修改如下：

```
template<class T, class U = T>
constexpr // since C++20
T exchange(T& obj, U&& new_value)
{
    T old_value = std::move(obj);
    obj = std::forward<U>(new_value);
    return old_value;
}
```

使用如下：

```
#include <iostream>
#include <vector>
#include <utility>
using namespace std;

int main()
{
    vector<int> v = {5, 6, 7};
    std::exchange(v, {1, 2, 3, 4});
    std::copy(begin(v), end(v), ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

## 7. quoted

该类用于字符串转义的处理。使用 `out << quoted(s, delim, escape)` 的形式，可以将字符串 `s` 的转义格式写入输出流中，使用 `in >> quoted(s, delim, escape)` 可以将输入流去除转义格式后写入字符串 `s` 中。其中 `delim` 指明了需要转义的字符，`escape` 指明了修饰该转义字符的字符。

```
#include <iostream>
#include <iomanip>
#include <sstream>
using namespace std;

int main()
{
    stringstream ss;
    string in = "String with spaces, and embedded \"quotes\" too";
    string out;

    auto show = [&](const auto& what)
    {
        &what == &in
            ? cout << "read in [" << in << "]\n" << "stored as [" << ss.str()
              << "]\n"
            : cout << "written out [" << out << "]\n\n";
    };

    ss << quoted(in);
    show(in);
    ss >> quoted(out);
    show(out);

    ss.str("");
}

in = "String with spaces, and embedded $quotes$ too";
const char delim{ '$' };
const char escape{ '%' };

ss << quoted(in, delim, escape);
show(in);
ss >> quoted(out, delim, escape);
show(out);
}
```

---

<https://baike.baidu.com/item/c++14/12424852>

## lambda

C++11中，`lambda` 函数参数需要被声明为具体的类型。

C++14中，放宽了这一要求，允许`lambda` 函数参数类型 使用类型说明符 `auto`。

```
auto lambda = [] (auto x, auto y) { return (x + y); };
```

尽管使用了C++11的关键字`auto`，但是泛型`lambda` 函数并不遵循 `auto` 类型推导的句法，而是遵循 模板参数推导的 规则（它们相似，但并不是所有情况下都是相同的）。上面的代码 和下面的 等价：

```
struct unnamed_lambda {
    template <typename T, typename U>
    auto operator()(T x, U y) const {
```

```
    return x + y;
}
};

auto lambda = unnamed_lambda();
```

### lambda 捕获表达式

C++11 lambda 通过 值拷贝 或 引用 捕获在外层作用域 声明的变量。这意味着 lambda 的 值成员 不可以是 只能 move 的类型。

C++14 允许 被捕获的 成员 用任意表达式初始化。这既允许了 通过 move 捕获，也允许了 任意声明 lambda 的成员，而不需要 外层作用域 有一个 具有相应名字的变量。

这是通过 使用一个 初始化表达式 完成的

```
auto lambda=[value{1}] { return value; }
```

lambda函数返回 1。因为value 被初始化为1。被声明的捕获变量的类型 以和auto相同的方式，根据 初始化表达式 推导。

可以使用 std::move 使之 被用以通过 move 捕获

```
auto ptr = std::make_unique(10);
auto lambda = [ptr{std::move(ptr)}] { return (*ptr); };
```

声明 ptr{std::move(ptr)} 使用了2次ptr，第一次 声明了一个新变量，但是由于 C++的作用域规则，在初始化表达式 求职 完毕之前，该变量不在 作用域内。所以 第二个 ptr 代表了 在 lambda 外声明的 变量。

### 函数的返回值类型推导

C++11 允许lambda 函数 根据return 语句的 表达式类型推断返回类型。

C++14 为所有函数 提供这个能力。C++14 还扩展了 原有规则，使得 函数体不是 return expression; 形式的函数 也可以使用 返回类型推导。

为了诱发 返回类型推导，函数声明必须将 auto 作为其返回类型。但没有C++11 的后置返回类型说明符：

```
auto deduceReturnType() //Return type to be determined.
```

如果函数实现中有多个return表达式，这些表达式必须可以推导为 相同的类型。

使用返回类型推导的 函数 可以被 前向声明，但是在 定义之前不能使用。

这样的函数可能存在递归，但递归调用必须在 函数定义中的 至少一个return 语句之后，例如

```
auto correct(int i) {
    if (i == 1)
        return i;
    else
        return correct(i - 1) + i;
}
```

下面会引发一个error:

```
auto wrong(int i) {
    if(i != 1)
        return wrong(i - 1) + i; // Too soon to call this. No prior return statement.
    else
        return i;
}
```

### 其他类型推断

C++11 中增加了 2种 推断类型的方法。

auto 根据给出的表达式产生 具有合适类型的变量

decltype 可以计算给出的表达式的类型。

但是 decltype 和 auto 推断类型的方式 是不同的。

特别是，auto 总是推断 非引用类型，就好像使用了 std::remove\_reference 一样。而

`auto&&` 总是推导出 引用类型。

然而 `decltype` 可以根据表达式 的结果类别 和表达式的 性质推断出 引用或 非引用类型。

```
int i;
int &&f();
auto x3a = i; // decltype(x3a) is int
decltype(i) x3d = i; // decltype(x3d) is int
auto x4a = i; // decltype(x4a) is int
decltype((i)) x4d = i; // decltype(x4d) is int &
auto x5a = f(); // decltype(x5a) is int
decltype(f()) x5d = f(); // decltype(x5d) is int &&
```

C++14 增加了 `decltype(auto)` 的语法。这将允许 `auto` 声明 使用 `decltype` 对于 给定表达式的 推断规则。

`decltype(auto)` 的语法也可以用于 返回类型推导， 只需 使用 `decltype(auto)` 代替 `auto` 来 诱发。

### constexpr 限制

C++11 引入了 `constexpr` 的概念。这样的函数可以在 编译期 执行。

它们的返回值 可以通过 计算所需的 常量表达式 得出。

然而C++11要求 `constexpr` 函数 只含有一个将被 返回的 表达式

C++14放宽了这些限制。`constexpr` 函数将可以 含有以下内容：

任何声明，除了：

`static` 或 `thread_local` 变量

    没有初始化的 变量声明

    条件分支 `if` 和 `switch`。 但不允许`goto`

    所有的循环基于， 包括 基于range 的 `for`循环

    表达式可以改变一个对象的值，如果该对象的生命周期 在常量表达式函数内开始。包括 对有 `constexpr` 声明的 任何 非`const` 非静态 成员函数 的调用。

C++11 指出，所有被声明为 `constexpr` 的非静态成员 函数 也隐含声明为 `const` (即函数 不能修改 `*this` 的值)。这点已经 被删除。 非静态成员函数可以为 非`const`。然而，只有 当对象的生命周期 在常量表达式 求职中开始，非`const` 的 `constexpr` 成员函数才可以 修 改类成员。

### 变量模板

之前的版本中，模板 可以是 函数模板 或 类模板。

C++14 还可以创建 变量模板。 模板的一般规则，包括 特化 都适用于 变量模板的 声明 和 定义。

### 聚合初始化

C++11 新增 类内成员 默认初始化，这是一个表达式， 被应用到 类作用域 的 成员上，如果 构造器没有初始化这个成员。

聚合体的定义 被改为 明确排除任何含有类内成员默认初始值的类。因此，他们不允许使用 聚合初始化。

C++14 放松了这一限制，这种类型也允许 聚合初始化。 如果花括号初始化列表 不提供该参 数的值，类内成员默认 初始值 会初始化它。

### 二进制字面量

C++14 的数字可以 用 二进制形式指定。前缀`0B` 或 `0b`

### 新标准库特性

共享和互斥

C++14 增加了一类 共享的互斥体 和 相应的共享锁。

## 异构查询

C++标准库 定义了 4个关联容器类。

set 和 multiset 允许使用者 根据一个值 在容器中查找 对应的 同类型的值。

map 和 multimap 允许使用者 指定 键key 和 值value 的类型，根据 key 进行查找 并返回 对应的值。

然而查询只能接受指定类型的 参数，在 map 和 multimap 中就是 key 的类型， set 和 multiset 就是 值本身的类型。

C++14 允许通过 其他类型进行查找。只要 这个类型 和 实际key类型 之间可以进行比较操作。

在 小于 运算符被重载的 情况下， 允许 key 为 std::string 的map 和 一个 key 为 const char\* 的map 进行比较。

为了保证向后兼容，这种 异构查询 只在相应 关联容器的 比较运算符 允许的情况下 有效。 标准库类 std::less (set 和 map 的默认比较算符) 和 std::greater 也可以用来 进行 异构查询。

## 标准字面值

C++11 定义了 自定义字面值的语法，但是标准库没有使用 这些语法。

C++14增加了下面的标准字面值

"s"，用于创建各种 std::basic\_string 类型

"h", "min", "s", "ms", "us", "ns"，用于创建 相应的 std::chrono::duration 时间间隔 2个"s" 互不干扰，因为 表示字符串的 只能对 字符串字面值操作， 表示秒的 只针对数字。

。。。？这个是说 以前没有 string s = "asd"; ？ 还是什么？ 不。 使用 前缀或后缀 来 表明 字面量的 类型。

## 寻址多元组

C++11 引入了 std::tuple 类型，允许不同类型的值 的聚合体 用 编译器整型常数 索引。

C++14 还允许使用 类型 代替 常数 索引，从多元组中 获取 对象。

如果多元组 含有 多于一个 这个类型的 对象， 会产生一个 编译错误。

```
tuple t ("foo", "bar", 7);
int i = get(t); // i == 7
int j = get<2>(t); // Same as before: j == 7
string s = get(t); // Compiler error due to ambiguity
```

## 其他函数特性

std::make\_unique 可以像 std::make\_shared 一样使用， 用于产生 std::unique\_ptr 对象。

std::integral\_constant 会有一个返回常量值的 调用 运算符重载

增加了 std::cbegin, cend 。 常量迭代器

---

<https://blog.csdn.net/Devoteechen/article/details/103470419>

C++14

## 泛型lambda

允许lambda 的形参声明 中使用 类型说明符 auto。

```
auto lambda = [](auto x, auto y) {return x + y;}
```

泛型lambda 遵循 模板参数推导。上面等价于下面

```
struct unnamed_lambda {
```

```

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

auto lambda = unnamed_lambda{};

```

### lambda 捕获参数中使用 表达式

C++11 的lambda 通过 值拷贝 或 引用 捕获 已经在外层作用域声明的 变量。 这意味着 lambda 捕获的变量 不可以是 move-only 的类型

C++14 允许lambda 成员用任意的 被捕获表达式 初始化。 这既允许 capture by value-move， 也允许了 任意声明的 lambda 的成员， 而不需要 外层作用域 有一个 具有相应名字的变量。 这称为 广义捕获(Generalized capture)。 即在 捕获子句 中 增加并初始化新的变量， 该变量不需要在lambda 表达式所处 的闭包域(enclosing scope) 中存在； 即使再闭包域中存在， 也会被 新变量覆盖。 新变量类型 由它的 初始化表达式推导。 用途是 从外层作用域中捕获 move-only 的变量 并使用它。

```
auto lambda = [value = 1] {return value;}
```

通过 move 可以使之被用以通过move 捕获

```
auto ptr = std::make_unique<int>(10); //See below for std::make_unique
auto lambda = [value = std::move(ptr)] {return *value;}
```

### 函数返回类型推导

C++11 允许 lambda 根据return 的表达式类型 推导返回类型。

C++14 为一般函数也提供了 这个能力。 还扩展了 原有规则， 是的函数体 并不是 return expression; 形式的 函数也可以使用 返回类型推导。

为了启用返回类型推导， 函数声明必须将 auto 作为返回类型。 但 没有C++11 的后置返回类型说明符：

```
auto DeduceReturnType(); //返回类型由编译器推断
```

如果函数中有多个return 语句， 这些表达式必须可以推断为 相同的类型。

使用返回类型 推导 的函数 可以前向声明， 但在 定义之前不可以使用。 它们的 定义在 使用它们的 翻译单元(translation unit) 之中 必须是可用的。

这样的函数中可以存在 递归， 但递归调用必须在函数定义中的 至少一个 return 语句之后

```

auto Correct(int i) {
    if (i == 1)
        return i;           // 返回类型被推断为int
    else
        return Correct(i-1)+i; // 正确， 可以调用
}

auto Wrong(int i) {
    if(i != 1)
        return Wrong(i-1)+i; // 不能调用， 之前没有return语句
    else
        return i;           // 返回类型被推断为int
}

```

C++11有2种推断类型的方法。

auto 根据 表达式， 产生具有合适类型的变量。

decltype 可以计算出 表达式的 类型。

auto 和 decltype 推断类型的方式 是不同的。

auto总是 推断出 非ref类型， 就好像使用了 std::remove\_reference 一样， 而 auto&& 总是推导出 ref类型。

decltype 可以根据表达式的值类别(value category) 和表达式的性质 推断出 ref 或 非ref 类型。

```

int i;
int&& f();
auto x3a = i;           // x3a的类型是int
decltype(i) x3d = i;    // x3d的类型是int
auto x4a = (i);         // x4a的类型是int
decltype((i)) x4d = (i); // x4d的类型是int&
auto x5a = f();          // x5a的类型是int
decltype(f()) x5d = f(); // x5d的类型是int&&

```

C++14 增加了 `decltype(auto)` 语法，用来声明变量以及指示 函数返回类型。

当`decltype(auto)` 被用于声明变量时， 该变量必须 立即初始化。 假设该变量的 初始化表达式为 e，那么 该变量的 类型将被推导为 `decltype(e)`。 也就是说，在 推断变量类型时，先用 初始化表达式 替换 `decltype(auto)` 中的 `auto`，然后再根据 `decltype` 的语法规则 来确定 变量的 类型。

`decltype(auto)` 的语法 也可以用于 返回类型推导，只需要用 `decltype(auto)` 代替`auto`。

`decltype(auto)` 也可以被用于 指示函数的 返回值类型。 假设函数返回表达式 e，那么 该函数的返回值类型 将被推导为 `decltype(e)`。 也就是说 在推导函数返回值类型时，先用该返回值 替换 `decltype(auto)` 中的`auto`，然后根据`decltype` 的规则来推导 函数返回值的类型。

```

#include <iostream>

template<typename T>
T f();

struct S {int a;};

int a = 0;
S s;
decltype(auto) g1() {return s.a;}
decltype(auto) g2() {return std::move(a);}
decltype(auto) g3() {return (a);}
decltype(auto) g4() {return 0;}

int main() {
    decltype(auto) i1 = a;
    decltype(auto) i2 = std::move(a);
    decltype(auto) i3 = (s.a);
    decltype(auto) i4 = 0;
    f<decltype(i1)>();      // i1 为 int 型
    f<decltype(i2)>();      // i2 为 int&& 型
    f<decltype(i3)>();      // i3 为 int& 型
    f<decltype(i4)>();      // i4 为 int 型
    f<decltype(g1())>();   // g1() 返回 int 型
    f<decltype(g2())>();   // g2() 返回 int&& 型
    f<decltype(g3())>();   // g3() 返回 int& 型
    f<decltype(g4())>();   // g4() 返回 int 型
}

```

### 放松 `constexpr` 函数限制

C++11 引入了 声明为 `constexpr` 的函数 的概念。 声明为 `constexpr` 函数的意义是：如果 其参数 均为合适的 编译器常量，则对这个 `constexpr` 函数的 调用 就可以用于 期望 常量表达式 的场合 (如模板的 非类型 参数，枚举常量的值)。 如果参数的值 在运行期才能 确定，或者 虽然参数的值 是编译器常量，但不符合这个函数的要求，则对这个函数 调用的 求职 只能在 运行期进行。 然而C++11 要求 `constexpr` 函数 只含有 一个 将被返回的 表达式 (也可以还含有 `static_assert` 声明等其他语句，但允许的 语句类型很少。)

C++14 放松了这些限制。声明为 `constexpr` 的函数 可以具有 下面的内容：

任何声明，除了

`static` 或 `thread_local` 变量

    没有初始化的变量声明

    条件分支语句 `if` `switch`, 不允许 `goto`

    所有的循环语句，包括基于返回的 循环

    表达式可以改变一个对象的值，只需该对象的生命周期 在声明为 `constexpr` 的函数内部开始。包括对 有`constexpr` 声明的任何 非`const` 非静态 成员函数的 调用。

C++11 指出，所有被声明为 `constexpr` 的非静态成员函数 也隐式声明为 `const` (即函数不能修改 `*this` 的值)，这点已经被删除，非静态成员函数 可以是 非`const`。

## 变量模板

之前版本中，模板可以是 函数模板 或 类模板 (C++11引入类型别名模板)。

C++14 可以创建 变量模板。

在提案中给出的示例是 变量pi， 其可以被 读取 以 获得各种 类型的 pi 的值 (例如，当被读取为 整型时 值为3，被读取为 `float`, `double`, `long double`时， 可以是 近似 `float`, `double`, `long double` 精度的值) 包括特化在内，通常的模板的规则 都适用于 变量模板的 声明 和定义。

```
template<typename T>
constexpr T pi = T(3.141592653589793238462643383);

// 适用于特化规则：
template<>
constexpr const char* pi<const char*> = "pi";
```

## 聚合类成员初始化

C++11 增加了 `defualt member initializer`， 如果构造器 没有初始化某个成员，并且这个成员拥有 `default member initializer`， 就会使用 `default member initializer` 来初始化成员。 聚合类(aggregate type) 的定义被改为 明确排除 任何含有 `default member initializer` 的 类 类型，因此，如果一个类 含有 `default member initializer`, **就不允许使用 聚合初始化**。

```
struct CXX14_aggregate {
    int x;
    int y = 42;
};

CXX14_aggregate a = {1}; // C++14允许。a.y被初始化为42
```

## 二进制字面量

C++14， 以`0b` 或 `0B` 开头 的二进制

## 数字分位符

C++14， 单引号 作为 数字分位符。 为了更好的可读性。

Ada, D, java, perl, ruby 等使用 下划线 作为数字分位符。 C++不使用 下划线，是因为 下划线已经被 用在 用户自定义的 字面量的语法中。

```
auto integer_literal = 100'0000;
auto floating_point_literal = 1.797'693'134'862'315'7E+308;
auto binary_literal = 0b0100'1100'0110;
auto silly_example = 1'0'0'000'00;
```

## `deprecated` 属性

标记 不推荐使用的 实体。 可能在 编译时 出现 警告。

可以有一个 可选的 字符串文字 作为参数，以解释 原因，替代者。

```
[[deprecated]] int f();
```

```

[[deprecated("g() is thread-unsafe. Use h() instead")]]
void g( int& x );

void h( int& x );

void test() {
    int a = f(); // 警告: 'f' 已弃用
    g(a); // 警告: 'g' 已弃用: g() is thread-unsafe. Use h() instead
}

```

## 新的标准库特性

### 共享的互斥体 和 锁

C++14 增加了一类 共享的 互斥体 和 相对应的共享锁。 起初选择的名字 是 std::shared\_mutex， 但是由于 后来 增加了 与 std::timed\_mutex 相似的 特性， std::shared\_timed\_mutex 成了更合适的名字。

### 元函数的别名

C++11 定义了一组元函数，用于查询一个 给定类型 是否具有 某种特征，或者 转换给定类型的 某种特征，从而得到 另一个类型。 后一种 元函数通过成员类型 type 来返回 转换后的类型，当它们用在 模板中时，必须使用 typename 关键字。

```

template <class T>
type_object<
    typename std::remove_cv<
        typename std::remove_reference<T>::type
    >::type
>get_type_object(T&);

```

利用类型别名模板，C++14提供了 更便捷的写法。

命名规则是：如果 标准库的某个类模板（假设为 std::some\_class）只含有唯一的成员，即 成员类型type， 那么标准库提供 std::some\_class\_t 作为 tpyename std::some\_class::type 的别名。

C++14中，拥有类型别名的 原函数包括：

remove\_const、remove\_volatile、remove\_cv、add\_const、add\_volatile、add\_cv、remove\_reference、add\_lvalue\_reference、add\_rvalue\_reference、make\_signed、make\_unsigned、remove\_extent、remove\_all\_extents、remove\_pointer、add\_pointer、aligned\_storage、aligned\_union、decay、enable\_if、conditional、common\_type、underlying\_type、result\_of、tuple\_element。

```

template <class T>
type_object<std::remove_cv_t<std::remove_reference_t<T>>>
get_type_object(T&);

```

### 关联容器中的 异构查询

C++11 定义了4个 关联容器类， set， multiset， map， multimap， 查询只能接受 指定类型的参数， map， multimap中是 key 的类型， set和multiset 是容器中值本身的类型。

C++14 允许通过 其他类型 进行查找，只需要这个类型 和实际的 键类型之间可以进行 比较操作。

这允许 std::set<std::string> 使用 const cahr\* 或任何可以通过 operator< 与 std::string 进行比较的 类型 作为查询 参数。

为了向后兼容，这种异构查询 只在 提供给 关联容器的 比较器 允许的 情况下 有效。 标准库 的泛型 比较器，如 std::less<>， std::greater<> 允许异构查找。

## 标准自定义字面量

C++11 增加了 自定义字面量(user-defined literals) 的特性，使用户 能够定义新的 字面量后缀， 但标准库并没有对这一特性 加一利用。

C++14 标准库定义了 下面的 字面量后缀：

- "s": 创建 std::basic\_string 类型
- "h", min, s, ms, us, ns, 用于创建对应的 std::chrono::duration 时间间隔
- "if", i, il , 用于创建对应的 std::complex

```
auto str = "hello world"s; // 自动推导为 std::string
auto dur = 60s;           // 自动推导为 chrono::seconds
auto z   = 1i;            // 自动推导为 complex<double>
```

2个"s" 互补干扰，因为 表示字符串的 只能对 字符串字面量操作， 表示秒的 只针对数字。

通过类型寻址多元组

C++11 引入的 std::tuple 允许不同类型的值 的聚合体 在编译器用 常数 索引。

C++14 还允许 使用类型 代替 常数索引，从 多元组 中获取对象。

如果多元组中 含有 多于一个 这个类型的 对象，将会产生一个 编译错误

```
tuple<string, string, int> t("foo", "bar", 7);
int i = get<int>(t);          // i == 7
int j = get<2>(t);           // Same as before: j == 7
string s = get<string>(t);    //Compiler error due to ambiguity
```

std::make\_unique 可以像 std::make\_shared 一样使用，用于产生 std::unique\_ptr对象

std::is\_final, 用于识别一个class 类型是否 禁止被继承

std::integral\_constant, 增加了一个 返回常量值的 operator()

std::cbegin, cend, 常量迭代器

=====

=====

=====

C++17

<https://en.cppreference.com/w/cpp/17>

[https://blog.csdn.net/weixin\\_42482896/article/details/118943564](https://blog.csdn.net/weixin_42482896/article/details/118943564)

C++17新特性总结

语言特性

### 1. 折叠表达式

为了方便 模板编程， 分为 左右折叠。下图是解包形式

The instantiation of a *fold expression* expands the expression *e* as follows:

- 1) Unary right fold (*E op ...*) becomes (*E<sub>1</sub> op ... op E<sub>N-1</sub> op E<sub>N</sub>*)
- 2) Unary left fold (*... op E*) becomes ((*E<sub>1</sub> op E<sub>2</sub>*) op ...) op *E<sub>N</sub>*)
- 3) Binary right fold (*E op ... op I*) becomes (*E<sub>1</sub> op ... op (E<sub>N-1</sub> op (E<sub>N</sub> op I))*)
- 4) Binary left fold (*I op ... op E*) becomes (((*I op E<sub>1</sub>*) op *E<sub>2</sub>*) op ...) op *E<sub>N</sub>*)

```
template <typename... Args> auto sub_right(Args... args) {  
    return (args - ...);  
}  
  
template <typename... Args> auto sub_left(Args... args) { return (... - args); }  
  
template <typename... Args> auto sum_right(Args... args) {  
    return (args + ...);  
}  
  
int main() {  
    std::cout << sub_right(8, 4, 2) << std::endl; // (8 - (4 - 2)) = 6  
    std::cout << sub_left(8, 4, 2) << std::endl; // ((8 - 4) - 2) = 2  
  
    std::cout << sum_right(std::string{"hello "}, std::string{"world"})  
        << std::endl; // hello world  
  
    return 0;  
}  
  
. . . ? ? ?  
。数组 操作符 3个点  
.。一下子感觉 起飞了， 虽然感觉没有什么用，但是 好6。  
.。感觉 op 满足 结合律 还好。不满足 直接起飞。  
.。数组所在的 地方 就是最后 操作的地方
```

### 2. 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：

```
template <typename T> struct A { A(T, T); };  
int main() {  
    std::pair p(2, 4.5); // std::pair<int, double> p  
    std::tuple t(4, 3, 2.5); // std::tuple<int, int, double> t  
    std::less l; // std::less<void> l  
    auto y = new A{1, 2}; // A<int>::A(int, int)  
  
    return 0;  
}
```

### 3. auto 占位的 非类型模板形参

```
template <auto T> void foo() { std::cout << T << std::endl; }  
int main() {  
    foo<100>();  
    // foo<int>();
```

```

    return 0;
}

4. 编译器 constexpr if 语句
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;
    foo<false>(); //输出not ok, 并且汇编代码中只有 std::cout << "not ok"
    << std::endl;
    return 0;
}

```

。。。

## 5. constexpr 的lambda 表达式

lambda 表达式可以在 编译期 进行计算, 且 函数体 不能包含 汇编语句, goto, label, try, 静态变量, 线程局部存储, 没有初始化的 普通变量, 不能动态分配内存, 不能 new delete, 不能为 虚函数

```

int main() {
    constexpr auto foo = [] (int a, int b) { return a + b; };
    static_assert(5 == foo(2, 3), "not compile-time");
    return 0;
}

```

## 6. inline 变量

扩展了, 使之 可以在 头文件 或类内 初始化 静态 成员变量

```

// test.h
inline int value = 1;
// test.cpp
struct A {
    inline static int val = 1;
};

。。发现忘了, 应该是 类/结构体内 默认 非static, 需要 显式 static, 类/结构体外, 默认 static, 而且只能static 。吧 ?

```

## 7. 结构化绑定

C++11中, 要获得tuple 元素, 要使用 get<>() 或 tie<>() 。 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

```

auto student = std::make_tuple(std::string{"YongDu"}, 26, std::string{"man"});
std::string name;
size_t age;
std::string gender;
std::tie(name, age, gender) = student;
std::cout << name << ", " << age << ", " << gender << std::endl;
// YongDu, 26, man

```

C++17 中的结构化 绑定，大大方便了类似操作，而且 使用 ref 捕获时，还可以 修改 捕获对象里面的 值，代码也会简洁很多。

```
struct Student {
    std::string name;
    size_t age;
};

Student getStudent() { return {"dycc", 26}; }

int main() {
    auto student = std::make_tuple(std::string{"YongDu"}, 26, std::string{"man"});
    auto [name, age, gender] = student;
    std::cout << name << ", " << age << ", " << gender << std::endl;
    // YongDu, 26, man

    std::unordered_map<std::string, size_t> students;
    students.emplace(std::make_pair("DuYong", 26));
    students.emplace(std::make_pair("YongDu", 26));

    for (auto &[name, age] : students) {
        std::cout << name << ", " << age << std::endl;
    }

    auto [_name, _age] = getStudent();
    std::cout << _name << ", " << _age << std::endl;

    return 0;
}
```

#### 8. if, switch 初始化

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

#### 9. u8-char

[https://zh.cppreference.com/w/cpp/language/character\\_literal](https://zh.cppreference.com/w/cpp/language/character_literal)

#### 10. 简化的嵌套命名空间

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

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

## 11. using声明语句 可以声明 多个名称

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

## 12. 新的求值顺序规则

```
int main() {
    std::unordered_map<int, int> hashMap;
    hashMap[0] = hashMap.size(); // 此处不确定插入{0, 0}， 还是{0, 1}

    return 0;
}
```

C++17 优化了求值顺序：

后缀表达式 从左到右 求值，包括 函数调用 和 成员选择表达式

赋值表达式 从右向左。 包括 复合赋值。

从左 到右 计算位移操作符的 操作数。

## 13. 强制的复制消除

[https://en.cppreference.com/w/cpp/language/copy\\_elision](https://en.cppreference.com/w/cpp/language/copy_elision)

## 14. 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 ref，在多线程情况下，有可能 lambda 的 生命周期 超过 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 增加了 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。

```
struct A {
    int m_val;
    void foo() {
        auto lamfoo = [*this]() { cout << m_val << endl; };
        lamfoo();
    }
};

int main() {
    A a;
    a.m_val = 1;
    a.foo();

    return 0;
}
```

## 15. 简化重复命名空间的属性列表

```
[[gnu::always_inline]] [[gnu::hot]] [[gnu::const]] [[nodiscard]]
inline int f(); // declare f with four attributes
```

```
[[gnu::always_inline, gnu::const, gnu::hot, nodiscard]]
int f(); // same as above, but uses a single attr specifier that contains four
         attributes
```

```
// C++17:
[[using gnu:: const, always_inline, hot]] [[nodiscard]]
int f[[gnu::always_inline]](); // an attribute may appear in multiple specifiers

int f() { return 0; }

int main(){}
... . 感觉 和 注解 类似了？ 主要是 deprecated 也是 2个方括号的。
```

。。不过 不知道 这个 是什么作用。

#### 16. \_\_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() {
#ifndef __has_include
    cout << "iostream exist." << endl;
#endif

#ifndef __has_include(<cmath>)
    cout << "<cmath> exist." << endl;
#endif

    return 0;
}
```

#### 17. 新增属性

[[fallthrough]]

switch语句中 跳到下一条语句，不需要break，让 编译器 忽略警告。

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

[[nodiscard]]

所修饰的内容 不可被忽略，主要用于 修饰函数返回值

```
[[nodiscard]] auto foo(int a, int b) { return a + b; }
int main() {
    foo(2, 3); // 放弃具有 "nodiscard" 属性的函数的返回值
    return 0;
}
```

[[maybe\_unused]]

```
void foo1() {}
```

```
[[maybe_unused]] void func2() {} // no warning
```

```
int main() {
    int x = 1;
    [[maybe_unused]] int y = 2; // no warning
    return 0;
}
```

库相关

特性较多，更多信息参考 <https://en.cppreference.com/w/cpp/17>

## 1. 字符串转换函数

更方便地处理 字符串和 数字之间的 转换

```
#include <charconv>
#include <iostream>
using std::cout, std::endl;

int main() {
    std::string str("123456789");
    int val = 0;
    auto res = std::from_chars(str.data(), str.data() + 4, val);
    if (res.ec == std::errc{}) {
        cout << "val: " << val << ", distance: " << res.ptr - str.data() << endl;
        // val: 1234, distance: 4
    } else if (res.ec == std::errc::invalid_argument) {
        cout << "invalid" << endl;
    }

    str = std::string("12.34");
    double value = 0;
    auto format = std::chars_format::general;
    res = std::from_chars(str.data(), str.data() + str.size(), value, format);
    cout << "value: " << value << endl;
    // value: 12.34

    str = std::string("xxxxxxx");
    int v = 1234;
    auto result = std::to_chars(str.data(), str.data() + str.size(), v);
    cout << "str: " << str << ", filled: " << result.ptr - str.data()
        << " characters." << endl;
    // str: 1234xxxx, filled: 4 characters.

    return 0;
}
```

## 2. std::variant

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 更方便。

```
int main() {
    std::variant<int, std::string> var("hello");
    cout << var.index() << endl;

    var = 123;
    cout << var.index() << endl;

    try {
        var = "world";
        std::string str = std::get<std::string>(var); // 通过类型获取
        var = 3;
        int i = std::get<0>(var); // 通过索引获取
        cout << str << ", " << i << endl;
    } catch (...) {
    }

    return 0;
}。。? tuple ? union是什么
```

### 3. std::optional

简化 函数返回值的判断

```
std::optional<int> StoI(const std::string &str) {
    try {
        return std::stoi(str);
    } catch (...) {
        return std::nullopt;
    }
}

int main() {
    std::string str{"1234"};
    std::optional<int> result = StoI(str);
    if (result) {
        cout << *result << endl;
    } else {
        cout << "StoI() error." << endl;
    }

    return 0;
}
```

### 4. std::any

C++11 的 auto 太方便了变成，但是 auto 变量 一旦声明， 该变量类型不可再改变。

C++17 引入了 std::any 类型， 该类型 可以存储 任何类型的 值，也可以 时刻改变 它的类 型，类似于 py中的变量。

```
int main() {
    std::any a = 1;
    cout << a.type().name() << ", " << std::any_cast<int>(a) << endl;

    a = 2.2f;
    cout << a.type().name() << ", " << std::any_cast<float>(a) << endl;

    if (a.has_value()) {
        cout << a.type().name() << endl;
    }
    a.reset();
    if (a.has_value()) {
        cout << a.type().name();
    }
    a = std::string("hello");
    cout << a.type().name() << ", " << std::any_cast<std::string>(a) << endl;

    return 0;
}
```

### 5. std::apply

将tuple 元组 解包，并作为 函数的传入参数

```
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 << std::apply(add, std::pair(2, 3)) << endl;
    cout << std::apply(add_lambda, std::tuple(2, 3, 4)) << endl;
    return 0;
}
```

6. std::make\_from\_tuple

解包tuple 作为构造器 参数 构造对象

```

struct A {
    std::string _name;
    size_t _age;

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

int main() {
    auto param = std::make_tuple("kai", 18);
    std::make_from_tuple<A>(std::move(param));

    return 0;
}

```

7. std::string\_view

C++ 字符串有2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。

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。

```

void foo(std::string_view str_v) {
    cout << str_v << endl;
    return;
}

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

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

    return 0;
}

```

8. std::as\_count

将左值 转化为const 类型

```

int main() {
    std::string str{"hello world"};
    cout << std::is_const<decltype(str)>::value << endl;

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

```

9. std::filesystem

方便处理文件

```

int main() {
    fs::path str("./data");
}

```

```

if (!fs::exists(str)) {
    cout << "path not exist." << endl;
    return -1;
}
fs::directory_entry entry(str);
if (entry.status().type() == fs::file_type::directory) {
    cout << "it's a directory." << endl;
}
fs::directory_iterator list(str);
for (auto &file : list) {
    cout << file.path().filename() << endl;
}
return 0;
}
// it's a directory.
// "text.md"
// "test.cpp"
// "test.txt"

```

#### 10. std::shared\_mutex

[https://en.cppreference.com/w/cpp/thread/shared\\_mutex](https://en.cppreference.com/w/cpp/thread/shared_mutex)

。。看网页，有独占锁，有共享锁

独占锁

- lock
- try\_lock
- unlock

共享锁

- lock\_shared
- try\_lock\_shared
- unlock\_shared

<https://www.jianshu.com/p/0d96abf1dc88>

C++17部分特性整理

#### 1. 使 static\_assert 的文本信息可选

static\_assert ( 布尔常量表达式 , 消息 ) (C++11 起)

static\_assert ( 布尔常量表达式 ) (C++17 起)

#### 2. 删除trigraphs

删除三元转移字符

最初因为 iso646 的标准，部分国家 打不过 # ~ ^ 等字符，所以使用 ?? 加一个字符的方式 替代，C++11 不建议使用 trigraphs，C++17中废弃。

#### 3. 在模板参数中 允许使用 typename (作为替代类)

在模板声明中，typename 可以用作 class 的替代品，以 声明 类型模板形参 和 模板模板形参

template<typename T> class my\_array {};

// 两个类型模板形参和一个模板模板形参:

```

template<typename K, typename V, template<typename> typename C = my_array>
class Map
{
    C<K> key;
    C<V> value;
};

```

#### 4. braced-init-list 自动推导的新规则

采用大括号的形式初始化 auto 变量，可以自动完成类型的 推导。

规则 braced-init-list 左侧只能是 单个元素。

```
auto x1 = { 1, 2 }; // decltype(x1) is std::initializer_list<int>
```

【initializer\_list用于表示特定类型值的数组】

```
auto x2 = { 1, 2.0 }; // error: cannot deduce element type, 会对右侧常量进行类型  
检查
```

```
auto x3{ 1, 2 }; // error: not a single element
```

```
auto x4 = { 3 }; // decltype(x4) is std::initializer_list<int>
```

```
auto x5{ 3 }; // decltype(x5) is int
```

**std::initializer\_list**的简单介绍(列表初始化)

```
std::vector<int>a{1, 2, 3, 4, 5};
```

```
std::vector<int>a = {1, 2, 3, 4, 5};
```

//vecotr可以使用列表进行初始化，因为{1, 2, 3, 4, 5}被推导为**std::initializer\_list<int>**类型，vector的构造函数对**std::initializer\_list<int>**类型进行特化和对赋值运算符的重载进行了特化。

#### 5. 嵌套命名空间

嵌套命名空间定义： namespace A::B::C{ .. } 等价于 namespace A{ namespace B  
{namespace C { .. }}}}

namespace A::B::inline C { .. } 等价于 namespace A::B { inline namespace C  
{ .. }}

C++20 inline 可以出现于 除 首个意外的 每个命名空间 之前。

#### 6. 允许命名空间 和 枚举器的 属性

<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4266.html>

。。。

```
[[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; }
```

```
int main() {}
```

#### 7. 新的标准属性

```
[[fallthrough]]
```

```
switch
```

```
[[nodiscard]]
```

出现在 函数声明，枚举声明，类声明中

如果从 非转型 到 void 的 弃值 表达式中， 调用 声明为 nodiscard 的函数，或 调用 按值 返回 声明 为 nodiscard 的枚举 或 类的函数，则 鼓励 编译器发出警告。

```
struct [[nodiscard]] error_info { };  
error_info enable_missile_safety_mode();  
void launch_missiles();  
void test_missiles() {  
    enable_missile_safety_mode(); // 编译器可在舍弃 nodiscard 值时发布警告
```

```

    launch_missiles();
}

error_info& foo();
void f1() {
    foo(); // 并非按值返回 nodiscard 类型，无警告
}

```

[[maybe\_unused]]  
抑制 针对 未使用实体 的警告。

## 9. 允许所有非类型模板实参的 常量求值

```

//队列某个元素或者非静态数据成员的地址，对于非类型模板参数是不合法的;
template<int* p> class X { };

int a[10];
struct S { int m; static int s; } s;

X<&a[2]> x3; // error: address of array element
X<&s.m> x4; // error: address of non-static member
X<&s.s> x5; // error: &S::s must be used OK: address of static member
X<&S::s> x6; // OK: address of static member

```

## 10. 折叠表达式 及 在可变长参数模板中的使用

折叠表达式

- 一元左折叠(pack op ...)
- 一元右折叠(... op pack)
- 二元右折叠(pack op ... op I)
- 二元左折叠(I op ... op pack)

一元左折叠展开: ((E1 op E2) op ...) op EN  
 一元右折叠展开: E1 op (... op (EN-1 op EN))  
 二元右折叠展开: E1 op (... op (EN-1 op (EN op I)))  
 二元左折叠展开: (((I op E1) op E2) op ...) op EN

如果用作 init 或 pack 的表达式 在顶层 优先级 低于 转型，则它可以加 括号

```

template<typename ...Args>
int sum(Args&&... args) {
//    return (args + ... + 1 * 2); // 错误: 优先级低于转型的运算符
    return (args + ... + (1 * 2)); // OK
}

```

二元折叠表达式两边 的操作数 只能有一个 未展开的 参数包

```

template<typename... Args>
bool f(Args... args) {
    return (true + ... + args); // OK
}

template<typename... Args>
bool f(Args... args) {
    return (args && ... && args); // error: both operands contain unexpanded
parameter packs
}

```

变长参数模板参数解包方法

之前的解包方法

```

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

```

```

    std :: cout << value << std :: endl ;
}

template < typename T, typename ... Args >
void printf (T value , Args ... args ) {
    std :: cout << value << std :: endl ;
    printf ( args ... ) ;
}

int main () {
    printf (1, 2, "123 ", 1.1) ;
    return 0;
}

```

### C++17 基于 constexpr 的解包方法

```

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

```

### 11. if constexpr 关键字的使用

C++11 引入 constexpr 关键字，将 表达式 或 函数 编译为 常量。

C++17 将 constexpr 这个关键字 引入到 if 语句中。允许在 代码中 声明 常量，把这一特性引入到 条件判断中去，让代码 在编译时 完成 分支判断

```

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 ;
}

```

### 12. 结构化绑定声明

格式

```

attr(可选) cv-auto ref-运算符(可选) [ 标识符列表 ] = 表达式 ;
attr(可选) cv-auto ref-运算符(可选) [ 标识符列表 ] {表达式} ;
attr(可选) cv-auto ref-运算符(可选) [ 标识符列表 ] (表达式) ;

```

名称	解释
attr	任意数量的属性的序列
cv-auto	可有 cv 限定的 auto 类型说明符
ref-运算符	& 或 && 之一
标识符列表	此声明所引入的各标识符的逗号分隔的列表
表达式	顶层没有逗号运算符的表达式（文法上为赋值表达式），且具有数组或非联合类之一的类型。

#### 绑定到数组

```

int a[2] = {1,2};
auto [x,y] = a; // 创建 e[2]，复制 a 到 e，然后 x 指代 e[0], y 指代 e[1]

```

```

auto& [xr, yr] = a; // xr 指代 a[0], yr 指代 a[1]

绑定到元组式类型
float x{};
char y{};
int z{};
std::tuple<float&, char&&, int> tpl(x, std::move(y), z);
const auto& [a, b, c] = tpl;
// a 指名指代 x 的结构化绑定; decltype(a) 为 float&
// b 指名指代 y 的结构化绑定; decltype(b) 为 char&&
// c 指名指代 tpl 的第 3 元素的结构化绑定; decltype(c) 为 const int

```

绑定到结构体成员

```

struct S {
    int x1 : 2;
    volatile double y1;
};

S f();
auto [x, y] = f(); // x 是标识 2 位位域的 int 左值
                    // y 是 volatile double 左值
//根据测试结果 auto [x, y] 前不能加cv操作符, 否则编译不过: 由于tuple_size无法识别

```

注意

标识符的数量必须等于 数组元素 或 元素元素 或 结构体中非静态数据成员的 数量。

### 13. if 和 switch 语句中的 变量初始化

```

if ( init-statement <u>opt</u> condition ) statement
if ( init-statement <u>opt</u> condition ) statement else statement
switch ( init-statement <u>opt</u> condition ) statement
...

```

之前

```

status_code foo() {
{
    status_code c = bar();
    if (c != SUCCESS) {
        return c;
    }
}
// ...
}

```

现在

```

status_code foo() {
    if (status_code c = bar(); c != SUCCESS) {
        return c;
    }
}
// ...
}

```

之前

```

void safe_init() {
{
    std::lock_guard<std::mutex> lk(mx_);
    if (v.empty()){
        v.push_back(kInitialValue);
    }
}
// ...
}

```

}

现在

```
void safe_init() {
    if (std::lock_guard<std::mutex> lk(mx_); v.empty()) {
        v.push_back(kInitialValue);
    }
    // ...
}
```

---

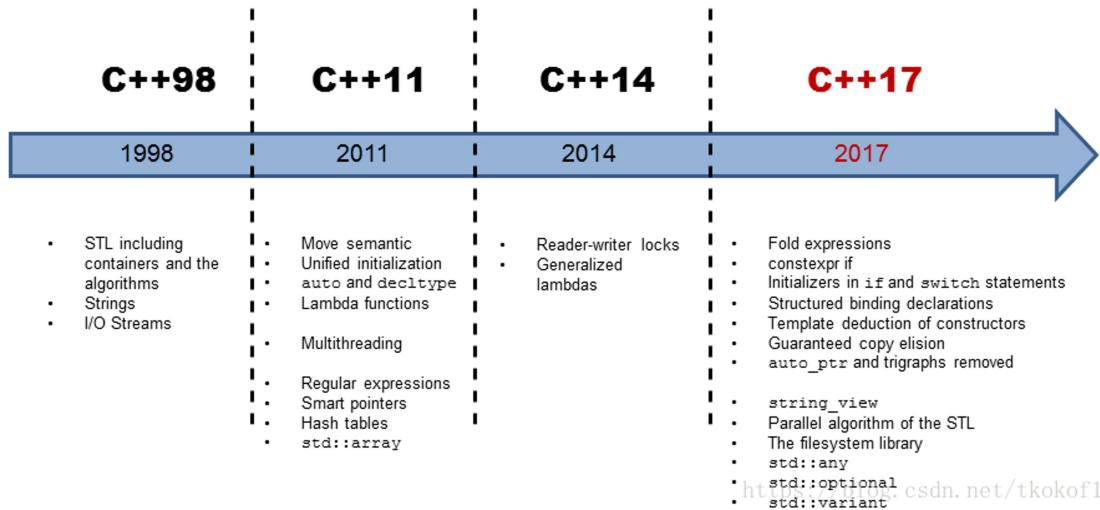
<https://blog.csdn.net/tkokof1/article/details/82713700>

[译]C++17, 标准库新引入的并行算法

原文www.modernescpp.com/index.php/c-17-new-algorithm-of-the-standard-template-library  
。。。不过访问不了了。。

C++17 之前标准库 有超过 100个算法，包括 搜索，计数，区间，元素操作等。

C++17 重载了 其中的 69个算法，并新增 7个算法。重载的算法 和 新增的算法 都支持 指定一个 所谓的 执行策略(execution policy)。通过调整这个参数，你可以指定 算法是 串行， 并行 或者 矢量并行 的方式 来运行



。。。不知道怎么回事，OneNote 的边框 向 右侧延伸了 一段，但是 没有看到 有什么东西 顶过去的。。

C++17 新引入的 7个算法

```
std::for_each_n  
std::exclusive_scan  
std::inclusive_scan  
std::transform_exclusive_scan  
std::transform_inclusive_scan  
std::parallel::reduce  
std::parallel::transform_reduce
```

C++17引入的 算法 在 纯函数式语言 Haskell 中都有 对应的方法

for\_each\_n 对应 map

exclusive\_scan 和 inclusive\_scan 对应的方法为 scanl 和 scanl1

transform\_exclusive\_scan 等同于组合使用 map 和 scanl，而 transform\_inclusive\_scan 等同于组合或者 map 和 scanl1.

reduce 对应 foldl 或者 foldl1.

transform\_reduce 对应 map 和 foldl 的组合或者 map 和 foldl1 的组合.

先说下 Haskell 中 这些方法的 功能

map 可以对一个列表应用一个函数

foldl 和 foldl1 可以对一个列表应用一个二元运算并将结果归纳为一个数值. foldl 与 foldl1 相比额外需要一个初始值.

scanl 和 scanl1 的操作与 foldl 和 foldl1 基本一致,但是他们会产 生所有的中间结果,所以最终你会获得一个列表,而不是一个数值.

foldl, foldl1, scanl 和 scanl1 的操作都是从列表的左侧开始.

Haskell 的例子

The screenshot shows a GHCi session window with the following code and numbered annotations:

```
Prelude> let ints = [1..9] (1)
Prelude> let strings= ["Only","for","testing","purpose"] (2)
Prelude> map (\a -> a * a) ints (3)
[1,4,9,16,25,36,49,64,81]
Prelude> scanl (*) 1 ints (4)
[1,1,2,6,24,120,720,5040,40320,362880]
Prelude> scanl (+) 0 ints (5)
[0,1,3,6,10,15,21,28,36,45]
Prelude> scanl (+) 0 . map(\a -> a * a) $ ints (6)
[0,1,5,14,30,55,91,140,204,285]
Prelude> scanl1 (+) . map(\a -> length a) $ strings (7)
[4,7,14,21]
Prelude> foldl1 (\l r -> l ++ ":" ++ r) strings (8)
"Only:for:testing:purpose"
Prelude> foldl (+) 0 . map (\a -> length a) $ strings (9)
21
Prelude> ■
```

The annotations correspond to the following steps:

- (1) 定义整数列表 ints
- (2) 定义字符串列表 strings
- (3) 使用 map 将 ints 中的每个元素乘以自身
- (4) 使用 scanl 从 1 开始将 ints 中的元素逐个相乘
- (5) 使用 scanl 从 0 开始将 ints 中的元素逐个相加
- (6) 使用 scanl1 将 strings 中的元素逐个相加, 并使用 map 将每个元素转换为其长度
- (7) 使用 scanl1 将 strings 中的元素逐个相加, 并使用 map 将每个元素转换为其长度
- (8) 使用 foldl1 将 strings 中的元素逐个连接, 使用 ":" 作为分隔符
- (9) 使用 foldl 将 strings 中的元素逐个相加, 并使用 map 将每个元素转换为其长度

(1) 和 (2) 处的代码分别定义了一个整数列表(ints)和一个字符串列表(strings).

在 (3) 中, 我给整数列表(ints)应用了一个 lambda 函数(\a -> a \* a).

(4) 和 (5) 则更加复杂些:(4) 中我将整数列表中的所有整数对相乘(乘法单位元素1作为初始元素). (5) 中则做了所有整数对相加的操作.

(6), (7), 和 (9) 中的操作可能有些难以理解, 你必须从右往左来阅读这几个表达式.

scanl1 (+) . map(\a -> length a) (即(7)) 是一个函数组合, 其中的点号(.)用以组合左右两个函数. 第一个函数将列表中的元素映射为元素的长度, 第二个函数则将这些映射的长度相加.

(9) 中的操作和 (7) 很相似, 不同之处在于 foldl 只产生一个数值(而不是列表)并且需要一个初始元素(我指定初始元素为0),

现在, 表达式(8)看起来应该比较简单了, 他以":"作为分隔符连接了列表中的各个字符串元素.

C++示例

```
#include <iostream>
#include <string>
#include <vector>
#include <execution>

int main()
{
    std::cout << std::endl;

    // for_each_n

    std::vector<int> intVec{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    // 1
    std::for_each_n(std::execution::par, // 2
                   intVec.begin(), 5, [](int& arg) { arg *= arg; });

    std::cout << "for_each_n: ";
```

```

for (auto v : intVec) std::cout << v << " ";
std::cout << "\n\n";

// exclusive_scan and inclusive_scan
std::vector<int> resVec{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::exclusive_scan(std::execution::par,           // 3
    resVec.begin(), resVec.end(), resVec.begin(), 1,
    [](int fir, int sec) { return fir * sec; });

std::cout << "exclusive_scan: ";
for (auto v : resVec) std::cout << v << " ";
std::cout << std::endl;

std::vector<int> resVec2{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };

std::inclusive_scan(std::execution::par,           // 5
    resVec2.begin(), resVec2.end(), resVec2.begin(),
    [](int fir, int sec) { return fir * sec; }, 1);

std::cout << "inclusive_scan: ";
for (auto v : resVec2) std::cout << v << " ";
std::cout << "\n\n";

// transform_exclusive_scan and transform_inclusive_scan
std::vector<int> resVec3{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::vector<int> resVec4(resVec3.size());
std::transform_exclusive_scan(std::execution::par,      // 6
    resVec3.begin(), resVec3.end(),
    resVec4.begin(), 0,
    [](int fir, int sec) { return fir + sec; },
    [](int arg) { return arg *= arg; });

std::cout << "transform_exclusive_scan: ";
for (auto v : resVec4) std::cout << v << " ";
std::cout << std::endl;

std::vector<std::string> strVec{ "Only", "for", "testing", "purpose" };
// 7
std::vector<int> resVec5(strVec.size());

std::transform_inclusive_scan(std::execution::par,        // 8
    strVec.begin(), strVec.end(),
    resVec5.begin(),
    [](auto fir, auto sec) { return fir + sec; },
    [](auto s) { return s.length(); });

std::cout << "transform_inclusive_scan: ";
for (auto v : resVec5) std::cout << v << " ";
std::cout << "\n\n";

// reduce and transform_reduce
std::vector<std::string> strVec2{ "Only", "for", "testing", "purpose" };

std::string res = std::reduce(std::execution::par,          // 9
    strVec2.begin() + 1, strVec2.end(), strVec2[0],
    [](auto fir, auto sec) { return fir + ":" + sec; });

std::cout << "reduce: " << res << std::endl;

// 11
std::size_t res7 = std::transform_reduce(std::execution::par,
    strVec2.begin(), strVec2.end(), 0u,

```

```

[] (std::size_t a, std::size_t b) { return a + b; },
[] (std::string s) { return s.length(); }
);

std::cout << "transform_reduce: " << res7 << std::endl;
std::cout << std::endl;

return 0;
}

```

与 Haskell 中的示例对应, 我使用 `std::vector` 创建了整数列表 (1) 和字符串列表 (7).

在代码 (2) 处, 我使用 `for_each_n` 将(整数)列表的前5个整数映射成了整数自身的平方.

`exclusive_scan` (3) 和 `inclusive_scan` (5) 非常相似, 都是对操作的元素应用一个二元运算, 区别在于 `exclusive_scan` 的迭代操作并不包含列表的最后一个元素, Haskell 中对应的表达式为: `scanl (*) 1 ints.` (译注: 结果并不完全等同, Haskell 的 `scanl` 操作包含列表最后一个元素, 后面提到的相关 Haskell 对应也是如此, 注意区别)

`transform_exclusive_scan` (6) 执行的操作有些复杂, 他首先将 `lambda` 函数 `function [] (int arg) { return arg *= arg; }` 应用到列表 `resVec3` 的每一个元素上, 接着再对中间结果(由上一步 `lambda` 函数产生的临时列表)应用二元运算 `[] (int fir, int sec) { return fir + sec; }` (以 0 作为初始元素), 最终结果存储于 `resVec4.begin()` 开始的内存处. Haskell 中对应表达式为:

`scanl (+) 0 . map(\a -> a * a) $ ints.`

(8) 中的 `transform_inclusive_scan` 和 `transform_exclusive_scan` 所执行的操作很类似, 其中第一步的 `lambda` 函数将元素映射为了元素的长度, 对应的 Haskell 表达式为:

`scanl1 (+) . map(\a -> length a) $ strings.`

现在, 代码中的 `reduce` 函数 (9) 看起来就比较简单了, 他需要在各个(字符串)元素之间放置 “:” 字符. 因为结果的开头不能带有 “:” 字符, `reduce` 的迭代是从第二个元素开始的 (`strVec2.begin() + 1`), 并以第一个元素作为初始元素(`strVec2[0]`). Haskell 中对应表达式为: `foldl1 (\l r -> l ++ ":" ++ r) strings.`

如果你想深入了解一下 (11) 中的 `transform_reduce`, 可以看看我之前的文章, 这里同样给出 Haskell 中对应的表达式: `foldl (+) 0 . map (\a -> length a) $ strings.`

。。。。直接复制的。

<https://en.cppreference.com/w/cpp/experimental/parallelism>

---

[https://blog.csdn.net/qq\\_55125921/article/details/126531996](https://blog.csdn.net/qq_55125921/article/details/126531996)

总结归纳: C++17新特性

关键字

`constexpr`

扩展了使用范围, 可以用于 `if` 语句中, 也可以用于 `lambda` 表达式。

`#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;
}

```

static\_assert  
扩展了用法, 显示文本可选  
static\_assert(true, "");  
static\_assert(true); //c++17支持

typename  
扩展, 允许出现在 模板的 模板的参数中。  
回顾typename 的用法:  
    用于模板中, 表示模板参数为类型  
    用于声明某名字是变量名

```

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;
}

```

新特性的typename用法

```

#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;
}

```

### inline

可以让遍历有多于一次的定义。C++17之前，我们定义全局变量，总需要将变量定义在cpp文件中，然后通过 **extern** 关键字 告知编译器 这个变量已经在其他地方定义过了。

inline变量出现后，我们可以直接将全局变量定义在头文件中，而不用担心出现redefine 的错误信息。

扩展用法，可以用于 定义内联变量，功能与内联函数相似。inline可以避免函数 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，下面的代码 不会发生错误

```
// 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;
}
```

### auto

C++11中，auto 能够通过初始化器推导出变量的 类型。

C++14中，auto 能力进一步提升，能通过 return语句 推导出 函数的返回类型。

使用 auto 能提升编码效率，同时能够简化重构流程。但是 C++11 的auto推导，往往结果和预期的不同。

C++11为了能支持 统一初始化，引入了新的 统一初始化语法：

```
// c++11
auto x3{ 1, 2 }; // std::initializer_list<int>
auto x4 = { 3 }; // decltype(x4) is std::initializer_list<int>
auto x5{ 3 };   // std::initializer_list<int>
```

这3种方法初始化的变量，最终类型 推导的结果 都是 std::initializer\_list，而不是我们认为的 int。

这是因为：当auto 声明变量 的 表达式 是 {} 包围的，推导的类型就会使 std::initializer\_list

c++17中，对 auto 表达式 推导的规则进行了改变。

```
// c++17
auto x3{ 1, 2 }; // error: not a single element
```

```
auto x4 = { 3 }; // decltype(x4) is std::initializer_list<int>
auto x5{ 3 }; // decltype(x5) is int
```

更加的直观

语法

lambda表达式

C++11中，lambda 只能捕获 this， this是当前对象的一个只读的引用。

C++17，捕获的this，是当前对象的一个拷贝，捕获当前对象的拷贝，能够确保当前对象释放后，lambda 能安全地调用this 中的方法和变量。

条件表达式中支持初始化语句

C++17 支持在 if 或 switch 语句中进行初始化。

```
// c++17之前
map<int, string> c = { {1,"a"} };
{
    auto res = c.insert(make_pair(2, "b"));
    if(!res.second) {
        cout << "key 1 exist" << endl;
    } else {
        cout << "insert success, value:" << res.first->second << endl;
    }
}
```

上面的代码中，由于 res 是一个临时变量，不希望影响后面的代码，所以用 {} 限制它的作用域。

在 if 条件中初始化 res，更简洁

```
// c++17
map<int, string> c = { {1,"a"} };
if(auto res = c.insert(make_pair(2, "b")); !res.second ) {
    cout << "key 1 exist" << endl;
} else {
    cout << "insert success, value:" << res.first->second << endl;
}
```

```
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;
}
```

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

```
#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;
}

```

## 结构化绑定

用包含一个或多个变量的 中括号，表示结构化绑定，但 使用结构化绑定时，必须使用 auto 关键字。即绑定时声明变量。

```

/*
 * 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;
}

```

允许非类型模板参数进行常量计算  
非类型模板参数可以传入类的静态成员。

```
class MyClass
{
public:
    static int a;
};

template<int *arg>
void foo() {}

int main()
{
    foo<&MyClass::a>();
    return 0;
}
```

### 聚合初始化

初始化对象时，用 花括号 对其成员 进行赋值。

```
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;
}
```

### 嵌套命名空间

```
//传统写法
namespace A
{
    namespace B
    {
        namespace C
        {

        };
    };
};

//新写法
namespace A::B::C
{
};
```

lambda 捕获 \*this  
可以捕获 \*this，但是 this 及其成员 只读。

```
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
        }();
    }
};

枚举[类]对象的构造
可以给枚举[类]对象赋值

```

```

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

```

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。

```

int main()
{
    float value = 0x1111f;
    return 0;
}

```

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);

```

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。

下面是带有争议的代码段

```
#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;
}

```

为了解决这个情况，新计算顺序规则是

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。

赋值表达式从右向左求值。这包括复合赋值。

从左到右计算移位操作符的操作数。

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。

```

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

```

简化重复命名空间的属性列表

```

[[ using CC: opt(1), debug ]] void f() {}
//作用相当于 [[ CC::opt(1), CC::debug ]] void f() {}
。。using CC 使得 CC 作用于 后续的全部方法。

```

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。

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 和 改写构造器的用法。

```

#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)
```

宏

\_has\_include

判断有没有包含某文件

```
int main()
{
#if __has_include(<cstdio>)
    printf("hehe");
#endif
#if __has_include("iostream")
    std::cout << "hehe" << std::endl;
#endif
return 0;
}
```

属性

fallthrough

用于switch语句块内，表示会执行下一个case 或 default

```
switch (0)
{
    case 0:
        ok1 = 0;
    [[fallthrough]];
    case 1:
        ok2 = 1;
    [[fallthrough]];
}
```

nodiscard

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。

```
[[nodiscard]] class A {};
[[nodiscard]] enum class B {};
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;
}
```

maybe\_unused

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 编译器会没用实体的警告。即加上该属性后，对某一实体不会发出"未使用" 的警告。

```
[[maybe_unused]] class A {};
[[maybe_unused]] enum B {};
[[maybe_unused]] int C;
[[maybe_unused]] void fun();
```

## 数据类型

C++17 新增以下的数据类型：

`std::variant`

是类型安全的 联合体，加强版的union，variant支持更复杂的数据类型，例如 map, string 等。

`std::optional`

表示一个可能存在的值。当通过 `函数` 创建对象时，通常是 通过函数 来返回错误码， 通过 `参数` 返回 对象本身。

如果通过optional 返回创建的对象，就会变得更加直观

`std::optional` 提供下面的几个方法：

`has_value()`， 检查对象是否有值

`value()`， 返回对象的值，值不存在，则抛出`std::bad_optional_access` 异常

`value_or()`， 值存在时返回值，不存在时 返回默认值

`std::any`

一个类型安全的 可以保存任何值的 容器

`std::string_view`

可以理解为 原始字符串的 一个只读引用。

`string_view` 本身没有申请额外的 内存来存储 原始字符串的 数据，仅保存了 原始字符串 的 地址 和 长度 等信息。

很多情况下，我们只是 临时处理 字符串，本就不需要 原始字符串的 拷贝。

`string_view` 可以减少不必要的 内存拷贝，可以提高 程序性能。 相比使用字符串指针，`string_view` 做了更好的封装。

要注意的是，`string_view` 由于没有 原始字符串的所有权，所以要注意 原始字符串的 生命 周期。

当原始的字符串已经销毁，则不能再调用 `string_view`。

=====

=====

=====

[https://blog.csdn.net/qq\\_38289815/article/details/126447913](https://blog.csdn.net/qq_38289815/article/details/126447913)

## C++ 20 新特性简介

本篇介绍的内容源自于 C++ 之父 Bjarne Stroustrup 的论文——HOPL4。HOPL 是 History of Programming Languages（编程语言历史）的缩写，是 ACM (Association of Computing Machines, 国际计算机协会) 旗下的一次会议，Bjarne 的这篇论文是他为 2021 年 HOPL IV 会议准备的。这篇 HOPL4 论文尤其重要，因为它涵盖了 C++98 之后的所有 C++ 版本，从 C++11 直到 C++20。论文中有许多涉及提案的细节，也包含了一些“故事”，感兴趣的可以阅读原文。

### Concept

对于C++ 来说，泛型编程 和 使用模板的元编程 已经取得了巨大的成功。  
但是，对泛型组件的接口 却迟迟未能以 一种令人满意的方式 进行合适的规范。

例如，C++98中，标准库算法大致是如下规定的

```
template<typename Forward_iterator, typename Value>
ForwardIterator find(Forward_iterator first,
                     Forward_iterator last,
                     const Value& val) {
    while (first != last && *first != val) {
        ++first;
    }
    return first;
}
```

### C++ 标准规定

第一个模板参数必须是 前向迭代器

第二个模板参数必须是 能够使用 == 与 该迭代器的 值类型 进行比较

前2个 函数参数必须 标示出 一个序列(sequence)

这些要求是隐含在代码中的：编译器所要做的就是在函数体中使用模板参数。

结果是：极大的灵活性，对正确调用生成出色的 代码，以及对不正确的调用 有 非常糟糕的 错误信息。

解决方案 显而易见：将前2项条件作为 模板接口的一部分来指定：

```
template<forward_iterator Iter, typename Value>
    requires equality_comparable<Value, Iter::value_type>
forward_iterator find(Iter first, Iter last, const Value& val);
```

equity\_comparable 标识了 2个模板参数之间必需有的关系。这样的 多参数 概念非常 常见。表达第三个要求([first, last) 是一个 序列) 需要一个库扩展。

C++20 在 Ranges 标准库组件中 提供了 该特性

```
template<range R, typename Value>
    requires equality_comparable<Value, Range::value_type>
forward_iterator find(R r, const Value& val) {
    auto first = begin(r);
    auto last = end(r);
    while (first != last && *first != val)
```

```

    ++first;
    return first;
}

```

concept 实际上是一个 语法糖，本质可以认为是一个模板类型的 bool 变量。定义一个 concept 本质上是在 定义一个 bool 类型的 编译器的变量。使用一个 concept 实际上是利用 SFINAE 机制 来约束 模板类型。

约束模板类型，用传统的 C++11 编写代码如下：

```

template<class T,
         class = typename std::enable_if<std::is_integral<T>::value>::type>
T add_test(T& a, T& b) {
    return a + b;
}

```

标准库已经定义了一些常用的 concepts，位于 头文件 <concepts> 中。

有了concept 后，做类型约束后，代码清晰很多

```

template<typename T>
concept Integral = std::is_integral<T>::value;

```

C++ 20 引入 requires 关键字，在 template 声明以后 紧接着使用 requires 关键字 说明 模板参数 需要 满足的 concept。

```

template <typename T>
requires Integral<T>
T add(T& a, T& b) {
    return a + b;
}

template <typename T>
T add(T& a, T& b) requires Integral<T> {
    return a + b;
}

template <Integral T>
T add(T& a, T& b) {
    return a + b;
}

int main() {
    add(1, -2);
    add(1.0f, 2.0f); // 匹配失败
}

```

.....

## 模板

在C++程序中改进模块化 是一个迫切的需求。从C语言中， C++ 继承了 #include 机制，依赖从头文件使用 文本形式 包含 C++ 源代码，这些头文件中 包含了 接口的 文本定义。 一个流行的 头文件 可以在 大型程序的 各个单独编译 的部分中 被 #include 数百次。

基本问题是：

不够卫生：	一个头文件中的 代码可能会 影响 同一个编译单元中 包含的 另一个 #include 中的 代码的含义，因此 #include 并不是 顺序无关的。 宏是 这里 的 一个主要问题，但不是唯一的问题。
分离编译的 不一致性：	2个编译单元中 同一个实体的 声明可能不一致，但并非所有此类错误 都 被 编译器 或 链接器 捕获。
编译次数过 多：	从源代码文本 编译接口比较慢。从源代码文本 反复地编译 同一份接口 非常慢

```

#include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
}

```

这段标准代码有 70个字符，但是在 #include 之后，它会产生 419909 个字符 需要编译器

来消化。尽管现代C++ 编译器有 傲人的处理速度，但是 模块化问题 已经迫在眉睫。

模块化是什么意思？

顺序独立性： import X; import Y; 应该与 import Y; import X; 相同。换句话说，任何东西都不能隐式 地 从一个 模块泄露到另一个模块。这是 #include 文件的一个关键问题。#include 中的任何内容都会影响所有后续的 #include

顺序独立性 是 代码卫生 和性能的 关键。通过坚持这种做法，Gabriel Dos Reis 的模块实现 也比 使用 头文件 在编译时间上得到了 10倍量级 的性能提升 --- 即使在旧式 编译中，使用了 预编译头文件 也是如此。

考虑下面的 C++20 模块化的 简单示例

```
export module map_printer; // 定义一个模版
import iostream;           // 使用 iostream . . . 。没有<> 不是头文件。
import containers;         // 使用自己的 containers
using namespace std;

export                     // 让 print_map() 对 map_printer 的用户可见
template<Sequence S>
    requires Printable<Key_type<S>> && Printable<Value_type<S>>
void print_map(const S& m) {
    for (const auto& [key, val] : m) {
        cout << key << " -> " << val << endl;
    }
}
```

这段代码定义了一个模块 map\_printer，该模块提供函数 print\_map 作为其用户接口，并使用了 从 模块 iostream 和 containers 导入的功能来实现该函数。

为了强调与 旧的C++ 风格 的区别，我使用了 概念和 结构化绑定。关键思想：

export 指令 使实体 可以被 import 到另一个模块中。

import 指令 使得从另一个 模块 export 出来的实体 能够被使用。

import 的实体 不会被隐式地 再 export 出去。

import 不会将实体 添加到 上下文中，它只会使实体能被使用（因此，未使用的import 基本上是 无开销的）

最后2点不同于 #include，并且它们对于模块化 和 编译期性能 至关重要。

头文件和 #include 不会 瞬间被淘汰，可能再过几十年都不会。

好几个人和组织提出，我们需要一些过渡机制，使得 头文件 和 模块 可以在程序中 共存，并让库 为 不同代码成熟度的用户 同时提供 头文件和 模块的 接口。

考虑在无法修改 iostream 和 container 头文件的 约束下 实现 map\_printer

```
export module map_printer;

import <iostream>;
import "containers";
using namespace std;

export
template<Sequence S>
    requires Printable<Key_type<S>> && Printable<Value_type<S>>
void print_map(const S& m) {
    for (const auto& [key, val] : m) {
        cout << key << " -> " << val << endl;
    }
}
```

指明某个头文件的import 指令工作起来 几乎 和 #include 一样 --- 宏，实现细节 及 递归地 #include 到的头文件。

但是，编译器确保 import 导入的 “旧头文件” 不具有 相互依赖关系。也就是说，头文件的 import 是顺序无关的，因此提供了 部分，但并非全部的 模块化的好处。

例如，像 import <iostream> 这样导入 单个头文件（粒度更粗的模块），程序员就需要 决定 该导入哪些头文件，也因为 与 文件系统进行不必要的多次 交互 而降低 编译速度，还限制了 来自不同头文件的 标准库组件的 预编译。

## 协程

提供了一种协作式多任务模型，比使用线程 或进程 要高效得多。

协程曾是 早期 C++ 的重要组成部分。如果没有提供协程的库，C++将胎死腹中，但是由于多种原因，协程并没有进入 C++98标准。

C++20 的协程的历史 始于 Niklas Gustafsson(微软) 关于 “可恢复函数”的提案。其主要目的是 支持 异步IO，“能够处理成千上万或以百万计客户的服务器应用程序”。它相当于引入 C#(2015年的6.0版) 的 `async/await` 功能。

Niklas 的提案引发了 来自 Oliver Kowalke 和 Nat Goodspeed 的基于 Boost.Coroutine 的竞争提案。

`await` 设计 无栈，不对称 且 需要语言支持，而 Boost 的设计 则使用 栈，具有 对称控制原语 且 基于库。无栈协程 只能在 其自身函数体中 挂起，而不能从其 调用的 函数中 挂起。这样，挂起 仅涉及 保存单个栈帧 (“协程状态”)，而不是保存整个栈。对于性能而言，是一个巨大优势。

协程的设计空间很大，因此很难达成共识。

委员会 希望结合这2个方式的优点， 经过讨论分析，结论时， 有可能 同时 利用这2种方式的优点，但需要 研究， 然后 花费了 数年研究，但没有得出明确的 结果。 与此同时， 出现了更多提案。

在提案中， 3种想法反复出现

1. 将协程的状态及其操作表示为 `lambda`，从而 使协程 优雅地适配 C++ 类型系统，而不需要 `await` 式 协程 所使用的 某些 “编译器魔法”
2. 为无栈 和 有栈 协程提供通用接口 --- 也可能为 其他类型的 并发机制，如 线程 和 纤程，提供通用接口。
3. 为了在 最简单 和 最关键 的用途 (生成器 和 管道) 上获得最佳性能 (运行时间 和 空间)，无栈协程 需要编译器支持，并且 一定不能为了 支持 更高级的用例 而在 接口上 做妥协。

作者非常喜欢 通用接口的 想法，因为 可以最大限度 减少学习需要的努力，并使得 实现大为便捷。类似地，使用完全普通的对象 来表示 协程 将开放 整个语言 来支持协程。然而，最终 性能论胜出。

TS协程的一个 重要 且 可能致命的问题是， 它依赖于 自由存储区(动态内存，堆) 上的分配。在某些应用程序中，这是很大的开销。更糟糕的是，对于许多关键的 实时 和 嵌入式 应用程序，不能 使用 自由存储区，因为它 可能导致 不可预测的 响应时间 和 内存碎片。核心协程 没有这个问题。然而，Gor Nishanov 和 Richard Smith 论证了，TS协程 可以通过 多种方式 来保证 几乎所有用途都没有 (并检测 和防止其他用途) 自由存储区的 使用。特别是，对于几乎所有的 关键用例，都可以将自由存储区 使用 优化为 栈分配 (所谓的 Halo优化)

C++20 协程的 简单例子

```
generator<int> fibonacci() {
    int a = 0;
    int b = 1;

    while (true) {
        int next = a + b;
        co_yield a;           // 返回下一个斐波那契数
        a = b;                // 更新值
        b = next;
    }
}

int main() {
    for (auto v : fibonacci())
        cout << v << endl;
}
```

使用 `co_yield` 使 `fibonacci()` 成为 一个协程。

`generator<int>` 返回值 将保存生成 的下一个 `int` 和 `fibonacci()`等待 下一个调用 所需的 最小状态。对于异步使用，我们将用 `future<int>` 而不是 `generator<int>`。

对协程返回类型的 标准库支持 仍然不完整，但 库 就应该 在 生产环境的使用 中 成熟。

## 编译器计算支持

多年以来，在C++中，编译期求值 的重要性一直在稳步提高。

STL严重依赖 编译期分发，而 模板元编程 主要 旨在 将 计算从 运行期 转移到 编译期。甚至在 早期C++ 中，对重载的 依赖 以及 虚函数表 的使用 都可以看作 是通过 将计算从 运行期 转移到 编译期 来获得 性能。因此， 编译期一直是 C++的关键部分

C++从 C 继承了 只限于 整型 且 不能 调用函数的 常量表达式。

曾经有段时间， 宏对于 任何稍微复杂点的事情 都必不可少。

但这些 都不好 规模化。

一旦引入 模板 并 发现了 模板元编程，模板元编程 就广泛用于 在 编译器 计算 值 和 类型上。

2010年，Gabriel Dos Reis 和 Bjarne Stroustrup 发表了一篇论文，指出，编译期 的 值 计算 可以（也应该）像 其他 计算一样表达，一样地 依赖于 表达式 和 函数的 常规 规则，包括 使用 用户定义的类型。这称为 C++11 的 constexpr 函数，它是 现代 编译期 编程的 基础。C++14 推广了 constexpr 函数，C++20 增加了 好几个相关特性：

constexpr --- 保证在 编译期 进行求值的 constexpr 函数

constexpr --- 保证 在 编译期 初始化的 声明修饰符

允许在 constexpr 函数中 使用 成对的 new 和 delete

constexpr string 和 constexpr vector

使用 virtual 函数

使用 unions， 异常， dynamic\_cast， typeid

使用 用户定义类型 作为 值模板参数--- 最终允许在 任何可以用 内置类型的 地方 使用 用户定义类型。

is\_constant\_evaluated() 谓词 --- 使库实现者 能够在 优化代码时 大大减少 平台相关的内部函数的 使用

随着这一努力，标准库 正在变得 对编译期 求值 更加友好，最终也是 为了 让 C++23 或更高版本 支持 静态反射。

## <=> 三向比较运算符

在 <=> 投票进入 C++20 后，很明显，在语言规则 及其 标准库的 集成方面 都需要 进一步地 认真工作。

出于 对解决 跟 比较 相关的 棘手问题 的 过度热情 和 渴望， 委员会 成了 意外后果定律 的受害者。

一些委员(包括我) 担心引入<=> 过于仓促。然而，在我们的 担忧 坐实 的时候，早有很多工作 在 假设 <=> 可用的前提下 完成了。

此外，三向比较 可能带来的 性能优势 让 许多委员会成员 和 其他更广泛的 C++社区成员 感到兴奋。

因此，当发现 <=> 在 重要用例 中 导致了 显著 低效时， 那就是 一个相当 令人不愉快的 意外了。

类型有了<=> 之后， == 是从 <=> 生成的。对于字符串，==通常 通过 首先比较size 来优化： 如果字符串长度不同，则 字符串不相等。 从<=> 生成 == 则必须 读取 足够的 字符串 以确定它们的 词典顺序，那开销 就会大得多了。

经过长时间的讨论，我们决定 不从 <=> 生成 ==。这一点 和 其他一些修正，解决了 手头的问题，但 损害了 <=> 的根本承诺： 所有 比较 运算符 都可以 从一行简单的 代码中生成。此外，由于 <=> 的 引入， == 和 < 现在有了 许多不同于 其他运算符 的规则 (例如， == 被假定为 对称的)。无论好坏，大多数 与运算符 重载 相关的规则 都将 <=> 作 为 特例来对待。

## 范围

范围库始于 Eric Niebler 对 STL 序列观念 的推广 和 现代化的工作。

它提供了更易于使用，更通用 及 性能更好的 标准库 算法。

例如C++20 标准库 为 整个容器 提供了 期待已久的 更简单的 表示方法

```
void test(vector<string>& vs) {
```

```
    sort(vs);           // 而不是 sort(vs.begin(), vs.end());  
}
```

C++98 采用的 原始STL 将序列 定义为 一对迭代器。这 遗漏了 指定序列 的 2种重要方  
式。范围库提供了 3种 主要的 替代方法 (现在称为 ranges)：

1. (首项, 尾项过一): 用于当我们知道序列的 开始 和结束位置时 (例如, 对 vector 的  
开始 到结束 位置进行 排序)
2. (首项, 元素个数): 用于当我们 实际上 不需要计算序列的 结尾时 (例如, 查看列表的  
前10个元素)
3. (首项, 结束判断): 用于当我们使用谓词 (例如, 一个哨位) 来定义序列 的结尾时 (例  
如, 读取到输入结束)

range 本身是一种 concept。所有C++20 标准库算法 现在 都使用 concept 进行了 精确规  
定。这本身 是一个 重大 改进，并使得我们在 算法里 可以推广到 使用 range，而不仅仅是  
是 迭代器。这种推广，允许我们 将算法 如管道般 连接起来：

```
vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 10};  
auto even = [](int i) { return i%2 == 0; }  
for (int i : vec | view::filter(even)  
    | view::transform([](int i) { return i*i; })  
    | view::take(5))  
    cout << i << endl;      // 打印前 5 个偶整数的平方
```

。。。 ? ? ?

像在Unix中一样， 管道运算符 | 将其 左操作数的 输出 作为 输入 传递到 右操作数 (例如  
A|B 表示 B(A))。

一旦人们开始 使用协程 来编写 管道过滤器，这就会变得有趣得多。

## 日期和时区

日期库是 多年工作和 实际使用的 结果，它基于 chrono 标准库的 时间支持。  
2018年，通过投票 进入了C++20，并和 旧的 时间工具一起 放在 <chrono> 中。

考虑如何 表达时间点(time\_point)：

```
constexpr auto tp = 2016y/May/29d + 7h + 30min + 6s + 153ms;  
cout << tp << endl;      // 2016-05-29 07:30:06.153
```

该表示法 很传统 ( 使用用户定义的字面量)，日期表示为 年 月 日 结构。但是，当需要  
时，日期 会在 编译期 映射到 标准时间线 (system\_time) 上某个点 (使用 constexpr 函  
数)， 因此 它极其快速，也可以在 常量表达式中 使用。例如

```
static_assert(2016y/May/29==Thursday); // 编译期检查
```

默认情况下，时区是 UTC (又称 Unix 时间)，但转换为 其他时区 很容易

```
zoned_time zt = {"Asia/Tokyo", tp};  
cout << zt << '\n';      // 2016-05-29 16:30:06.153 JST
```

日期库 还可以处理 星期几 (例如, Monday 和 Friday)， 多个日历(如 格里历 和 儒略  
历)，以及 更深奥 (但必要) 的概念，如 闰秒

除了有用和快速之外，日期库 还有趣在 它提供了 非常细粒度 和 静态类型 检查。常见错  
误 会在 编译器 捕获。例如：

```
auto d1 = 2019y/5/4;      // 错误，是 5 月 4 日，还是 4 月 5 日  
auto d2 = 2019y/May/4;    // 正确  
auto d2 = May/4/2019;     // 错误  
auto d3 = d2 + 10;        // 错误，是加 10 天，还是加 10 年？
```

日期库 是标准组件中 一个少见的例子，它直接服务于 某应用领域，而非 仅仅 提供 支持  
性的 计算机科学 抽象。我希望在将来 的标准中 看到更多这样的例子。

## 格式化

iostream 库提供了 类型安全的 IO 的扩展，但它的格式化工具 较弱。

另外，还有的人 不喜欢使用 << 分隔输出值的方式。

格式化库 提供了一种 类 printf 的方式 去组装字符串 和 格式化输出值。同时这种方法 类型安全，快捷，并能和 iostream 协同工作。类型中带有 << 运算符 的可以在一个格式化的字符串中输出：

```
string s = "foo";
cout << format("The string '{}' has {} characters", s, s.size());
```

输出结果是 The string 'foo' has 3 characters。

这是“类型安全的printf” 变参模板思想的一个变体。大括号 {} 简单地表示了 插入参数值的 默认表示形式。参数值 可以按照 任意顺序 被使用

```
// s 在 s.size() 前
cout << format("The string '{0}' has {1} characters", s, s.size());
// s 在 s.size() 后
cout << format("The string '{1}' has {0} characters", s.size(), s);
```

像printf 一样，format() 为展现格式化细节 提供了一门小而完整的 编程语言，比如，字段宽度，浮点数精度，整数基 和 字段内对齐。不同于 printf()，format() 是可扩展的，可以处理 用户定义类型。

下面是<chrono> 库中 一个打印日期的例子

```
string s1 = format("{} ", birthday);
string s2 = format("{0:>15%Y-%m-%d} ", birthday);
```

年-月-日 是默认格式。>15 意味着 使用 15个字符 和 右对齐文本。日期库中还包含了 另一门小的 格式化语言 可以和 format() 一起使用。它甚至可以用来 处理 时区 和 区域  
std::format(std::locale{"fi\_FI"}, "{}", zt);

这段代码会 给出 芬兰的 当地时间。默认情况下， 格式化 不依赖于 区域， 但是你可以选择 是否根据区域来格式化。

相比 传统的 iostream， 默认区域无关的 格式化 大大提高了 性能，尤其是 当你不需要区域信息的时候。 输入(iostream) 没有等价 的format 支持。

## span

越界访问，有时也称为 缓冲区溢出，从C 的时代 以来 就一直 是一个严重的问题。

考虑下面的例子

```
void f(int* p, int n) { // n 是什么?
    for (int i = 0; i < n; ++i) {
        p[i] = 7;           // 是否可行?
    }
}
```

试问 一个工具，比如编译器 要如何知道 n 代表着 所指向的数组 中元素个数？ 一个开发者 如何 在一个 大型程序中 对此 使用保持正确？

```
int x = 100;
int a[100];
f(a, x);      // 可以
f(a, x / 2); // 可以： a 的前半部分
f(a, x + 1); // 灾难
```

编译器不能捕获 范围错误。运行时 检查 所有下标 则 普遍被 认为 对于 生产 来说 代价过于高昂。

显而易见的解决方案 是提供一种 抽象机制，带有一个 指针 在加上一个大小。举例来说，1990年，Dennis Ritchie 向 C 标准委员会 提议：“胖指针”，它的表示中 包含 内存空间以及存放运行期间可调整 的边界。但由于种种原因，C 标准委员会 没有通过这个提案。

2015年，Neil MacIntosh 在C++ 核心指南 里恢复了这一想法， 那里 我们需要 一种机制 来鼓励 和选择性地 强制 使用 高效 编程风格。span<T> 类模板 就这样 被放到 C++ 核心 指南的 支持库中，并立刻被 移植到 微软，Clang，GCC 的 C++编译器中。

2018年，投票进入 C++20，使用span 的一个例子如下：

```
void f(span<int> a) {    // span 包含一个指针和一条大小信息
    for (int& x : a) {
        x = 7;           // 可以
    }
}
```

range-for 从 span 中提取范围，并准确地遍历 正确数量的元素（无需 代价高昂的 范围检查）。这个例子说明了 一个适当的抽象 可以 同时 简化 写法 并提示 性能。

如果有必要的话，你可以 显式 指定 一个大小（比如 操作一个 子范围）。但这样的话，你需要 承担风险，并且这种写法比较扎眼，也易于让人警觉。

```
int x = 100;
int a[100];
f(a, x);      // 模版参数推导: f(span<int>{a, 100})
f(a, x / 2); // 可以: a 的前半部分
f(a, x + 1); // 灾难
```

自然，简单的元素 访问也可以，如  $a[7] = 9$ ，同时运行期 也能进行检查。span 的范围 检查 是 C++ 核心指南 支持库 (GSL) 的默认行为。

事实证明，将span 纳入 C++20 的 最具争议的 部分 在于 下标 和 大小的 类型。C++核心 指南中，span::size() 被定义为 返回 一个 有符号 整数，而不是标准库 容器所使用的 无符号整数。下标也是类似情况。这导致了一个古老争议的重演：

1. 一组人 认为 显然下标 作为 非负数 应该使用 无符号整数
2. 一组人 认为 与 标准库容器保持一致性更重要，这点使得 使用无符号整数 是不是一个 过去的失误变得 无关紧要。
3. 一组人 认为 使用 无符号整数 去表示一个 非负数 是一种误导（给人一种虚假的安全感），并且 是错误的主要来源之一。

不顾 span 最初的设计者（包括我在内）和实现者的强烈反对，第二组赢得了投票，并受到第一组热情地支持。就这样，std::span 拥有无符号的范围大小和下标。我个人认为那是一个令人悲伤的失败，即未能利用一个难得的机会来弥补一个令人讨厌的老错误。C++ 委员会选择了与问题兼容而不是消除一个重大的错误来源，这在某种程度上是可以预见的，也算不无道理吧。

使用无符号整数作为下标 会出什么问题呢？这似乎是一个 相当情绪化的话题。

我曾受到很多封 与之相关的 仇恨邮件。存在 2个 基本问题

1. 无符号数 并不以 自然数 为模型：无符号数 是用 模算术，包括减法，比如 ch 是一个 unsigned char，ch + 100 永远不会溢出。
2. 整数 和 无符号数 彼此相互交换，稍不留意 负值就会变成 巨大的无符号数，反之亦然。比如， $-2 < 2u$  为假，2u是 unsigned，因此，-2在比较前 会被转换为 一个 巨大的正整数。

一个真实情况下，偶尔可见的 无限循环的例子

```
for (size_t i = n-1; i >= 0; --i) { /* ... */ } // “反向循环”
```

。。。还好，我只用 int。

总的来说，作为 C++ 继承自 C 的特性，有符号 和 无符号 类型 之间的 转换规则 几十年来 都是 那种 难以 发现的错误 的 一个主要来源。

并发

尽管做出了努力，并正在形成广泛的共识，但是 人们所期望的 通用并发模型（“执行器”）在 C++20 中 还没有准备好。不过，有几个 不那么剧烈的 有用 改进 还是 及时完成了，包

括：

1. jthread 和 停止令牌
2. atomic<shared\_ptr<T>>
3. 经典的信号量
4. 屏障 和 锁存器
5. 小的内存模型的修复和改进

jthread( joining thread)，是一个 遵循 RAI<sup>I</sup> 的线程， 也就是说，如果 jthread 超出了作用域，它的析构函数 将合并线程 而不是 终止程序。

```
void some_fct() {  
    thread t1;  
    jthread t2;  
    // ...  
}
```

在作用域的最后， t1 的析构函数 会终止程序，除非 t1 的任务已经完成，已经join 或 detach，而 t2 的析构函数 会等待 其任务完成。

一开始的时候 (C++11 之前)，很多人 (包括我在内) 都希望 thread 可以拥有如今 jthread 的行为，但是根植于传统操作系统线程的人坚持认为终止一个程序要远比造成死锁好得多。2012 年和 2013 年，Herb Sutter 曾经提出过合并线程。这引发了一系列讨论，但最终却没有作出任何决定。2016 年，Ville Voutilainen 总结了这些问题，并为将合并线程纳入 C++17 发起了投票。投票支持者众多以至于我 (只是半开玩笑地) 建议我们甚至可以把合并线程作为一个错误修复提交给 C++14。但是不知何故，进展又再次停滞。到了 2017 年，Nico Josuttis 又一次提出了这个问题。最终，在八次修订和加入了停止令牌之后，这个提案才成功进入了 C++20。

"停止令牌" 解决了一个 老问题，即 如何在 我们对线程 的 结果不再 感兴趣后 停止它。基本思想是 使用协作式的线程取消方式。假如 我想要一个 jthread 停止，我就设置 它的停止令牌。线程 有义务 不时地去检查 停止令牌 是否被设置了，并在 设置时 进行 清理 和退出。这个技巧由来已久，对于 几乎每个有 主循环 的线程 都能 完好 高效地工作，在这个 主循环里 就可以 对停止令牌 进行检查。  
。。。或者 高消耗的 动作(SQL，访问其他 接口)前 查询下。

像往常一样，命名成了问题：safe\_thread, ithread(i代表可中断)，raii\_thread, joining\_thread，最终成了 jthread。  
C++核心指南支持库(GSL) 中称其为 gsl::thread。说实话，最合适的名字是 thread，但是不幸，那个名字已经被一个 不太有用的 线程 占用了。

## 次要特性

C++20提供了许多次要的 新特性，如：

- (1) C99 风格的指派 初始化器
- (2) 对 lambda 捕获的改进
- (3) 泛型 lambda 表达式的 模板参数列表
- (4) 范围 for 中初始化一个额外的变量
- (5) 不求值语境中的 lambda 表达式
- (6) lambda 捕获中的包展开
- (7) 在一些情况下 移除对 typename 的需要
- (8) 更多属性： [[likely]] 和 [[unlikely]]
- (9) 在不使用宏的情况下，source\_location 给出一段代码中的 源码位置
- (10) 功能测试宏
- (11) 条件 explicit
- (12) 有符号整数保证是 2 的补码
- (13) 数学上的常数，比如 pi 和 sqrt2
- (14) 位的操作，比如轮转和统计 1 的个数

其中有些属于改进，但是我担心的是晦涩难懂的新特性的数量之大会造成危害。对于非专家来说，它们使得语言变得更加难以学习，代码更加难以理解。我反对一些利弊参半的特性 (比如，使用指派初始化器的地方原本可以使用构造函数，那会产生更易于维护的代码)。

很多特性具有特殊用途，有些是“专家专用”。不过，有的人总是领会不到，一个对某些人有某种好处的特性，对于 C++ 整体可能是个净负债。当然，那些增加写法和语义上的通用性和一致性的小特性，则总是受欢迎的。从标准化的角度来看，即使最小的特性也需要花时间去处理、记录和实现。这些时间是省不掉的。

## 进行中的工作

### 网络库

2003年，Christopher M. Kohlhoff 开始开发一个名叫 asio 的库，以提供网络支持：Asio 是用于网络和底层 I/O 编程的一个跨平台 C++ 库，它采用现代化 C++ 的方式，为开发者提供了一致的异步模型

2005年，它成为 Boost 的一部分，并在 2006 年 被提案 进入标准。

2018年，它成为了 TS。

尽管经过了 13年的 重度生产环境使用，它还是未能进入 C++17 标准。更糟糕的是，让 网络库进入 C++20 的工作 也停滞不前。

延误的原因在于，我们仍在严肃地讨论，如何最好地将 asio 中 和 其他场合中 处理并发的方式 一般化。为此提出的“执行器(executors)”提案 得到了 广泛的支持。

## 契约

在最后一刻被移除出 C++20

为了避免引入新的关键字，我们使用 属性语法。例如 `[[assert: x+y>0]]`。 一个契约 对一个有效的程序 不起任何作用，因此 这种方式 满足属性的 原来概念。

有3种契约：

- assert: 可执行代码中的断言
- expects: 函数声明中的前置条件
- ensure: 函数声明中的后置条件

有3种不同等级的契约检查

- audit: 代价高昂的谓词，仅在某些 “调试模式” 检查
- default: 代价低廉的谓词，在生产中 检查 也是可行的
- axiom: 主要给 静态分析器 看的 谓词，在运行期 不会检查

在违反契约时，将执行（可能是用户安装的）契约违反处理程序。默认行为是 程序 立即终止。

基于Bloomberg 代码的 相关经验，当你把 契约 加入到一个 大型的 古老 代码仓库， 契约 总是会被 违反：

- 某些代码会违反契约，而实际上并没有 做 任何 该契约 所要防止的 事情。
- 某些 新契约本身 就包含错误
- 某些新契约 具有 意料之外的效果

人们发现工作文件文本中允许编译器基于所有契约（无论检查与否）进行优化。那并非有意而为之。从所有的契约在正确的程序中都有效角度看，这是合理的，但是这么做，对于那些带有特别为捕获“不可能的错误”而写的契约的程序来说却是灾难性的。考虑下面的例子：

```
[[assert: p!=nullptr]]  
p->m = 7;
```

假如 `p==nullptr`，那么 `p->m` 将是未定义行为。编译器被允许假设未定义行为不会发生；由此编译器优化掉那些导致未定义行为的代码。这样做的结果可能让人大吃一惊。在这样的情况下，如果违反契约之后程序能够继续执行，编译器将被允许假定 `p->m` 是有效的，因此 `p!=nullptr`；然后编译器会消除契约关于 `p==nullptr` 的检查。这种被称为“时间旅行优化”的做法当然是与契约的初衷大相径庭，还好若干补救方案被及时提出。

契约被从 C++20 中移除

## 静态反射

2013年，一个研究“反射”的研究组成立，并征集意见。有一个广泛的共识，那就是 C++ 需要静态反射机制。

更确切地说，需要一种方法来写出能够检查它自己是属于哪个程序一部分代码，并基于此往该程序中注入代码。那样，就可以使用简洁的代码替换冗长而棘手的样板代码，宏，语言外的生成器。比如，可以为下面的场景自动生成函数，如 I/O 流，日志记录，比较，用于存储和网络的封送处理(marshaling)，构造和使用对象映射，枚举的字符串化，测试支持，及更多可能。

反射研究组的目的 是为了 C++20 或 23 做好准备。

大家普遍认同，依赖在运行期间遍历一个始终存在的数据结构的反射/内省方式不适合 C++，因为这种数据的大小，语言构造的完整表示的复杂性和运行期遍历的成本都是问题。

在静态反射长时间的酝酿期内，基于 `constexpr` 函数的编译期计算稳步发展，最终出现了基于函数而不是类层次结构的静态反射的提案。设计焦点转移的主要论据，一部分是由于分析和生成代码这些事天生是函数式的，而且基于 `constexpr` 函数的编译期计算已经发展到元编程和反射相结合的地步。这种方法的另一个优点是从编译器的内部的数据结构看，为函数服务的天生就比类层次服务的更小，生命周期更短，因此它们使用的内存明显更少，编译速度也明显更快。

2019年2月，David Sankel 和 Michael Park 展示了一个结合了这两个方法优点的设计。在最根本的层面上仅有一个单一的类型存在。这达到了最大的灵活性，并且编译器开销也最小。

最重要的是，静态类型的接口可以通过一种类型安全的转换来实现（从底层的单一类型 `meta::info` 到更具体的类型，如 `meta::type_` 和 `meta::class_`）。通过concept重载，它实现了从 `meta::info` 到更具体类型的转换。

考虑下面的例子

```
namespace meta {
    consteval std::span<type_> get_member_types(class_ c) const;
}

struct baz {
    enum E { /*...*/ };
    class Buz { /*...*/ };
    using Biz = int;
}

void print(meta::enum_);
void print(meta::class_);
void print(meta::type_);

void f() {
    constexpr meta::class_ metaBaz = reflexpr(baz);
    template for (constexpr meta::type_ member : get_member_types(metaBaz))
        print(meta::most_derived(member));
}

. . . . .

```

这里关键的新语言特性是 `reflexpr` 运算符，它返回一个(元)对象，该对象描述了它的参数，还有 `template for`，根据一个异质结构中的元素的类型扩展每个元素，从而遍历该结构的各元素。此外，我们也有机制可以将代码注入正在编译的程序中。类似这种东西很可能在 C++23 或 26 中成为标准。

作为一个副作用，在反射方案上的雄心勃勃的工作也刺激了编译器求值功能的改进

标准中的类型特征集

源代码位置的宏（如 `_FILE_` 和 `_LINE_`）被内在机制所替代

编译期计算的功能（例如，用于确保编译期求值的 `consteval`）

展开语句 (template for——到 C++23 就可以用来遍历元组中的元素。

---

[https://blog.csdn.net/qq\\_35423154/article/details/124450695](https://blog.csdn.net/qq_35423154/article/details/124450695)

C++20 的这些新特性，你都知道吗？

语言特性

三路比较运算符

左操作数  $\leqslant$  右操作数

表达式返回一个对象，使得

如果 左 < 右，则  $(a \leqslant b) < 0$

如果 左 > 右，则  $(a \leqslant b) > 0$

如果 左 和 右 相等/等价，则  $(a \leqslant b) == 0$

```
int main() {
    double foo = -0.0;
    double bar = 0.0;

    auto res = foo <= bar;

    if (res < 0)
        std::cout << "-0 小于 0";
    else if (res > 0)
        std::cout << "-0 大于 0";
    else // (res == 0)
        std::cout << "-0 与 0 相等";
}
```

range for 中的初始化语句 和 初始化器

C++17 在 if, switch 添加 初始化器，

C++20 在 range for 添加了这个功能

```
for (auto n = v.size(); auto i : v) // 初始化语句 (C++20)
    std::cout << -n + i << ' ';
```

consteval

consteval 指定函数 是立即函数 (immediate function)，即每次调用该函数 必须产生 编译时常量。如果不能在 编译期间 执行，则编译失败。

```
consteval int sqr(int n) {
    return n*n;
}
constexpr int r = sqr(100); // OK

int x = 100;
int r2 = sqr(x); // 错误：调用不产生常量

consteval int sqrsqr(int n) {
    return sqr(sqr(n)); // 在此点非常量表达式，但是 OK
}

constexpr int dblsqr(int n) {
```

```
    return 2*sqr(n); // 错误: 外围函数并非 consteval 且 sqr(n) 不是常量
}
```

### constinit

它断言变量拥有静态初始化，即零初始化与常量初始化，否则程序非良构。

```
const char *g() { return "dynamic initialization"; }
constexpr const char *f(bool p) { return p ? "constant initializer" : g(); }

constinit const char *c = f(true); // OK
// constinit const char *d = f(false); // 错误
```

### concept

就是一种编译时谓词，指出一个或多个类型应该如何使用，其能用于进行模板实参的编译时校验，以及基于类型属性的函数派发。

例如在老版本的C++中，如果我们想要定义一个只针对某个类型的函数模板，就只能通过类型萃取机制如enable\_if\_t写一些又臭又长的代码

例如，想声明一个只针对整数的函数模板

```
template <typename T>
auto mod(std::enable_if_t<std::is_integral_v<T>, T> d)
{
    return d % 10;
}
```

约束条件简单还行，但是如果条件复杂，则代码会又臭又长，且难以复用。

在C++20中引入了concepts，我们可以用来指定函数类型：

```
template <class T>
concept integral = std::is_integral_v<T>;
```

```
template <integral T>
auto mod(T d)
{
    return d % 10;
}
```

### 约束

约束是逻辑操作和操作数的序列，它指定了对模板实参的要求。它们可以用于requires表达式中，也可以直接作为concept的主题。

下面使用requires约束表达式写一个针对utf-8的string的约束类型u8string\_t。

```
template <typename T>
concept u8string_t = requires (T t)
{
    t += u8"";
};
```

接着以这个约束类型声明一个模板函数print，此时只能能够满足u8string\_t约束的类型才能够匹配当前模板。

```
template <u8string_t T>
auto print(T t)
{
```

```
    cout << t << endl;
}
```

此时我们以不同类型的string来尝试调用，此时只有 u8string 调用成功。

```
int main()
{
    string str;
    u8string str_u8;
    u16string str_u16;
    u32string str_u32;

    print(str);           //调用失败
    print(str_u8);        //调用成功
    print(str_u16);       //调用失败
    print(str_u32);       //调用失败
}
```

## 协程

协程 是能暂停执行 以 在 之后恢复的函数。

协程 是无栈的：它们 通过返回到 调用方 暂停执行，并且 从栈中分离 存储 恢复执行 所需的数据。这样就可以 编写 异步执行的 顺序代码（例如，不使用显示的回调 来处理 非阻塞IO）， 还支持 对 惰性计算的 无限序列上的 算法 和 其他用途。

如果函数的 定义 进行了下列操作 之一，那么它是 协程：

1. co\_await 暂停执行，直到恢复

```
task<> tcp_echo_server() {
    char data[1024];
    while (true) {
        std::size_t n = co_await socket.async_read_some(buffer(data));
        co_await asyanc_write(socket, buffer(data, n));
    }
}
```

2. co\_yield 暂停执行并返回一个值（协程无法return）

```
generator<int> iota(int n = 0) {
    while(true)
        co_yield n++;
}
```

3. co\_return 完成执行 并返回一个值

```
lazy<int> f() {
    co_return 7;
}
```

注意，协程不能使用 变长实参，普通的return 语句，或占位符 返回类型（auto 或 concept）。constexpr 函数，构造器，析构器 及 main 函数 不能是协程。

## 模块

它是一个用于 在 编译单元间 分享 声明 和定义的 语言特性。 它们可以在某些地方 替代 使用头文件。

## 主要优点

没有头文件

声明实现 仍然可以分离，但非必要

可以显式指定 导出哪些 类或函数

不需要 头文件 重复引入 宏（include guards）

模块之间名称可以相同，并且不会冲突

模块只处理一次，编译更快（头文件每次引入都需要处理，需要通过 `pragma once` 约束）

预处理宏 只在模块内 有效

模块的引入 和 引入顺序 无关

创建模块

```
// helloworld.cpp
export module helloworld; // 模块声明
import <iostream>; // 导入声明

export void hello() { // 导出声明
    std::cout << "Hello world!\n";
}
```

导入模块

```
// main.cpp
import helloworld; // 导入声明

int main() {
    hello();
}
```

库特性

`format`

文本格式化库 提供了 `printf` 函数族 的 安全 且 可扩展的 替用品。

补充 现有的 C++ IO 流库 并复用 其 基础设施。

例如，对用户定义类型 重载 的流 插入运算符：

```
std::string message = std::format("The answer is {}.", 42);
```

`osyncstream`

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>,
    class Allocator = std::allocator<CharT>
> class basic_osyncstream; public std::basic_ostream<CharT, Traits>
```

类模板 `std::basic_osyncstream` 是 `std::basic_syncbuf` 的便利包装。提供机制 来 同步写入 同一个流 的线程（主要用于解决 `std::cout` 线程不安全问题）

用法如下

```
{
    std::osyncstream sync_out(std::cout); // std::cout 的同步包装
    sync_out << "Hello, ";
    sync_out << "World!";
    sync_out << std::endl; // 注意有冲入，但仍未进行
    sync_out << "and more!\n";
} // 转移字符并冲入 std::cout
```

`span`

是对象的 连续序列上的 无所有权视图。其所描述的 对象 能指代 对象的 相接序列，序列的首元素 在 0位置。 `span` 能拥有静态长度，该情况下 序列中的元素数 已经 并编码于 类型中，或拥有 动态长度。

```
#include <algorithm>
#include <cstddef>
```

```

#include <iostream>
#include <span>

template<class T, std::size_t N> [[nodiscard]]
constexpr auto slide(std::span<T,N> s, std::size_t offset, std::size_t width) {
    return s.subspan(offset, offset + width <= s.size() ? width : 0U);
}

template<class T, std::size_t N, std::size_t M> [[nodiscard]]
constexpr bool starts_with(std::span<T,N> data, std::span<T,M> prefix) {
    return data.size() >= prefix.size()
        && std::equal(prefix.begin(), prefix.end(), data.begin());
}

template<class T, std::size_t N, std::size_t M> [[nodiscard]]
constexpr bool ends_with(std::span<T,N> data, std::span<T,M> suffix) {
    return data.size() >= suffix.size()
        && std::equal(data.end() - suffix.size(), data.end(),
                      suffix.end() - suffix.size());
}

template<class T, std::size_t N, std::size_t M> [[nodiscard]]
constexpr bool contains(std::span<T,N> span, std::span<T,M> sub) {
    return std::search(span.begin(), span.end(), sub.begin(), sub.end()) !=
span.end();
//    return std::ranges::search(span, sub).begin() != span.end();
}

void print(const auto& seq) {
    for (const auto& elem : seq) std::cout << elem << ',';
    std::cout << '\n';
}

int main()
{
    constexpr int a[] { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
    constexpr int b[] { 8, 7, 6 };

    for (std::size_t offset{}; ; ++offset) {
        constexpr std::size_t width{6};
        auto s = slide(std::span{a}, offset, width);
        if (s.empty())
            break;
        print(s);
    }

    static_assert(starts_with(std::span{a}, std::span{a, 4})
        && starts_with(std::span{a+1, 4}, std::span{a+1, 3})
        && !starts_with(std::span{a}, std::span{b})
        && !starts_with(std::span{a, 8}, std::span{a+1, 3})
        && ends_with(std::span{a}, std::span{a+6, 3})
        && !ends_with(std::span{a}, std::span{a+6, 2})
        && contains(std::span{a}, std::span{a+1, 4})
        && !contains(std::span{a, 8}, std::span{a, 9}));
}

```

## Endian

主要用于判断 当前机器是 大端 还是小端。(之前只能通过 整型截断 或 union 判断)

如果所有标量类型 都是小端，则 std::endian::native 等于 std::endian::little	
如果所有标量类型 都是大端，则	等于 std::endian::big

如果所有标量类型 拥有等于1 的sizeof，则 端序无影响，且 std::endian::little, std::endian::big 及 std::endian::native 三个值相同。  
如果平台 使用 **混合端序**，则 std::endian::native 既不等于 big 也不等于 little。

```
#include <bit>
#include <iostream>

int main() {

    if constexpr (std::endian::native == std::endian::big)
        std::cout << "big-endian\n";
    else if constexpr (std::endian::native == std::endian::little)
        std::cout << "little-endian\n";
    else std::cout << "mixed-endian\n";
}
```

### jthread

就是 通过 RAI<sup>I</sup> 机制封装的 thread，其会在 析构时 自动调用 join 防止线程 crash。  
同时 它也是可中断的，可以搭配 这些 中断线程 执行的 相关类使用：

stop\_token: 查询线程 是否 中断  
stop\_source: 请求线程 停止 运行  
stop\_callback: 执行时，可以触发的 回调函数

### semaphore

是一个轻量级 的同步原语，可用来 实现 任何其他同步概念，如 mutex, shared\_mutex, latches, barriers 等。

根据 LeastMaxValue 不同，主要分为2种：

counting\_semaphore (多元信号量)：允许同一资源 有多于一个 同时访问，至少允许 LeastMaxValue 个 同时的 访问者。  
binary\_semaphore (二元信号量)：是 counting\_semaphore 的 特化的别名，其 LeastMaxValue 为1。 实现可能将 binary\_semaphore 实现得 比 counting\_semaphore 的默认实现 更高效。

用法如下：

```
// 全局二元信号量实例
// 设置对象计数为零
// 对象在未被发信状态
std::binary_semaphore smphSignal(0);
void ThreadProc()
{
    // 通过尝试减少信号量的计数等待来自主程序的信号
    smphSignal.acquire();
    // 此调用阻塞直至信号量的计数被从主程序增加
    std::cout << "[thread] Got the signal" << std::endl; // 回应消息
    // 等待 3 秒以模仿某种线程正在进行的工作
    std::this_thread::sleep_for(3s);
    std::cout << "[thread] Send the signal\n"; // 消息
    // 对主程序回复发信
    smphSignal.release();
}

int main()
{
    // 创建某个背景工作线程，它将长期存在
```

```

std::jthread thrWorker(ThreadProc);
std::cout << "[main] Send the signal\n"; // 消息
// 通过增加信号量的计数对工作线程发信以开始工作
smpSignal.release();
// release() 后随 acquire() 可以阻止工作线程获取信号量，所以添加延迟：
std::this_thread::sleep_for(50ms);
// 通过试图减少信号量的计数等待直至工作线程完成工作
smpSignal.acquire();
std::cout << "[main] Got the signal\n"; // 回应消息
}

```

## latch

是 std::ptrdiff\_t 类型的 向下 计数器，能用于同步线程。在创建时 初始化计数器的值。主要有以下特点：

- 线程可能在 latch 上阻塞 直到 计数器 减少到 0. 不可能 增加 或 重置 计数器，这使得 latch 是 单次使用的 屏障。
- 同时调用 latch 的成员函数，除了析构函数，不引入 数据竞争。
- 不同于 std::barrier，参与线程 能减少 std::latch 多于一次

。。CountDownLatch and CyclicBarrier

## barrier

类模板 barrier 提供允许 至多 为 期待数量的 线程 阻塞 直到 期待数量的 线程 到达 该屏障。不同于 latch，屏障可重用：一旦到达 的线程 从屏障阶段 的同步点 除阻，则可重用 同一屏障。

屏障对象的 生存期 由 屏障阶段的 序列组成。每个阶段 定义 一个阶段 同步点。在 阶段 中 到达 屏障的 线程 能通过 调用 wait 在 阶段同步点上阻塞，而且 将保持阻塞 直到 运行阶段完成 此步骤。

屏障阶段由下列步骤组成：

1. 每次调用 arrive 或 arrive\_and\_drop 减少期待数
2. 期待计数 抵达0时， 运行阶段完成步骤。完成步骤 调用 完成函数对象，并 除阻 所有 在 阶段同步点上阻塞的 线程。完成步骤的 结束 先发生于 所有 从完成步骤 所 除阻 的 调用的返回。
  - a. 对于特化 std::barrier<> (使用默认模板实参)，完成步骤 作为 对 arrive 或 arrive\_and\_drop 的 导致期待计数 抵达0 的调用的一部分运行。
  - b. 对于其他特化，完成步骤 在该阶段 期间 到达屏障的 线程 之一上运行。而 若 在完成步骤中 调用 屏障对象的 wait 以外的成员函数，则行为 未定义。
3. 完成步骤结束后，重置 期待计数 为构造时 指定的 值，可能为 arrive\_and\_drop 调用 所调整，并开始下一阶段。

## 位运算库

bit 库封装了一些常用的位操作，包括：

- bit\_cast: 将一个类型的对象表示重解释为另一类型的对象表示。
- byteswap: 反转给定整数值中的字节。
- has\_single\_bit: 检查一个数 是否为二的整数次幂。
- bit\_ceil: 寻找 不小于 给定值的 最小的 二的 整数次幂。
- bit\_floor: 寻找 不大于 给定值的 最大的 二的 整数次幂。
- bit\_width: 寻找 表示给定值所需的 最小位数。
- rotl: 计算逐位 左旋转 的结果。
- rotr: 计算逐位 右旋转 的结果。
- countl\_zero: 从最高位起计量连续的 0 位的数量。
- countl\_one: 从最高位起计量连续的 1 位的数量。
- countr\_zero: 从最低位起计量连续的 0 位的数量。
- countr\_one: 从最低位起计量连续的 1 位的数量。
- popcount: 计量无符号整数中为 1 的位的数量。

ranges

ranges 提供处理元素范围的组件，包括各种 视图适配器。最大的作用就是 让我们可以像组装 函数一样 组装 算法，使代码更高效，可读。

提供命名空间别名 std::views， 作为 std::ranges::views 的缩写。

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

int main()
{
    auto const ints = {0, 1, 2, 3, 4, 5};
    auto even = [] (int i) { return 0 == i % 2; };
    auto square = [] (int i) { return i * i; };

    // 组合视图的“管道”语法:
    for (int i : ints | std::views::filter(even) | std::views::transform(square))
    {
        std::cout << i << ',';
    }

    std::cout << '\n';

    // 传统的“函数式”组合语法:
    for (int i : std::views::transform(std::views::filter(ints, even), square)) {
        std::cout << i << ',';
    }
}
```

---

<https://zhuanlan.zhihu.com/p/137646370>

10分钟速览 C++20 新增特性

新增关键字

concept  
requires  
constinit  
consteval  
co\_await  
co\_return  
co\_yield  
char8\_t

新增标识符

import  
module

模块 Module

优点

没有头文件  
声明实现 仍然可以分离，但非必要  
可以显式 指定 导出什么 类，函数 等  
不需要 头文件 重入引入宏 (include guards)  
模块之间 名称可以相同 不会冲突  
模块只处理一次，编译更快(头文件每次引入 都需要处理)  
预处理宏 只在模块内 有效  
模块引入顺序 无关紧要

创建模块

```
// cppcon.cpp
export module cppcon;
namespace CppCon {
    auto GetWelcomeHelper() { return "Welcome to CppCon 2019!"; }
    export auto GetWelcome() { return GetWelcomeHelper(); }
}
```

引用模块

```
// main.cpp
import cppcon;
int main() {
    std::cout << CppCon::GetWelcome();
}
```

import 头文件
import <iostream>
**隐式地将 iostream 转换为 模块**
加速构建，因为 iostream 只会处理一次
和预编译头(PCH) 具有相似的效果

ranges

代表一串元素，或者一串元素中的一段  
类似 begin end 对

好处

简化语法，方便使用

```
vector<int> data{11, 22, 33};
sort(begin(data), end(data));
sort(data); // 使用 Ranges
防止 begin end 不匹配
使 变换/过滤 等 串联操作 成为可能。
```

相关功能

视图：延迟计算，不持有，不改写

Actions：即时处理(eagerly evaluated)，改写

Algorithms：所有接受 begin end 对的 算法 都可用

Views 和 actions 使用 管道符 | 串联

例子

串联视图

```
vector<int> data {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto result = data |
    views::remove_if([](int i) { return i % 2 == 1;}) |
    views::transform([](int i) { return to_string(i);});
// result = {"2", "4", "6", "8", "10"};
// 注意 以上操作被延迟，当你遍历result的时候才触发
```

## 串联actions

```
vector<int> data{4, 3, 4, 1, 8, 0, 8};  
vector<int> result = data | actions::sort | actions::unique;
```

排序然后去重，操作会原地 对 data 进行修改，然后返回。

## 过滤和变换

```
int total = accumulate (  
    view::ints(1) |  
    view::transform([](int i) {return i * i;}) |  
    view::take(10),  
    0);
```

view::ints(1) 产生一个 无限的 整型数列

平方

取前10个元素，然后累加

所有的计算 延迟到 accumulate 累加遍历时 发生

## 协程 Coroutines

### 什么是协程

是一个函数

具有如下关键字之一：

```
co_wait: 挂起协程，等待 其他计算完成  
co_return: 从协程返回（协程不能使用 return）  
co_yield: (同py yield)，弹出一个值，挂起协程，下次调用继续 协程的运行。  
for co_await: 循环体  
    for co_await (for-range-declaration: expression) statement
```

### 用处

简化如下问题的实现

```
generator  
异步IO  
延迟计算  
事件驱动的程序
```

### 例子

```
experimental::generator<int> GetSequenceGenerator(  
    int startValue,  
    size_t numberofValues) {  
    for (int i = 0 startValue; i < startValue + numberofValues; ++i) {  
        time_t t = system_clock::to_time_t(system_clock::now());  
        cout << std:: ctime(&t); co_yield i;  
    }  
}  
int main() {  
    auto gen = GetSequenceGenerator(10, 5);  
    for (const auto& value : gen) {  
        cout << value << "(Press enter for next value)" << endl;  
        cin.ignore();  
    }  
}
```

## concept

对模板类 和 函数的模板参数的 约束

编译器 断言

可声明多个

定义

```
template<typename T> concept Incrementable = requires(T x) {x++; ++x;};

template <typename T> concept HasSize = requires (T x){
    {x.size()} -> std::convertible_to<std::size_t>;
};

template<typename T>
requires Incrementable<T> && Decrementable<T>
void Foo(T t);
// or
template<typename T>
concept Incr_Decrementable = Incrementable<T> && Decrementable<T>;

template<Incr_Decrementable T>
void Foo(T t);
```

lambda表达式

[=, this] 显式捕获 this 变量  
C++20之前, [=] 隐式捕获this  
C++20开始, 需要显示捕获 this

模板形式的 lambda 表达式

可以在 lambda表达式中 使用 模板语法

```
[]template<T>(T x) /* ... */;
[]template<T>(T* p) /* ... */;
[]template<T, int N>(T (&a)[N]) /* ... */;
```

C++20之前, 获取 vector 元素类型, 需要

```
auto func = [](auto vec){
    using T = typename decltype(vec)::value_type;
}
```

C++20可以

```
auto func = []<typename T>(vector<T> vec) {
    // ...
}
```

之前 访问静态函数

```
auto func = [](auto const& x) {
    using T = std::decay_t<decltype(x)>;
    T copy = x; T::static_function();
    using Iterator = typename T::iterator;
}
```

C++20

```
auto func = []<typename T>(const T& x) {
    T copy = x; T::static_function();
    using Iterator = typename T::iterator;
}
```

之前 完美转发

```
auto func = [](auto&& ...args) {
    return foo(std::forward<decltype(args)>(args)...);
}
```

C++20

```
auto func = []<typename ...T>(T&& ...args) {
```

```
    return foo(std::forward(args)...);  
}
```

**lambda** 捕获支持 pack expansion

之前

```
template<class F, class... Args>  
auto delay_invoke(F f, Args... args) {  
    return [f, args...] {  
        return std::invoke(f, args...);  
    }  
}
```

C++20

```
template<class F, class... Args>  
auto delay_invoke(F f, Args... args) {  
    // Pack Expansion: args = std::move(args)...  
    return [f = std::move(f), args = std::move(args)...]() {  
        return std::invoke(f, args...);  
    }  
}
```

常量表达式 (constexpr) 的更新

constexpr 虚函数

constexpr 的虚函数 可以重写 非 constexpr 的虚函数  
非 constexpr 的虚函数 可以 重写 constexpr 的虚函数

constexpr 函数可以:

使用 dynamic\_cast() 和 typeid  
动态内存分配  
更改union成员的值  
包含 try/catch  
但是不允许 throw 语句  
在触发常量求值的时候 try/catch 不发生作用  
需要开启 constexpr std::vector

```
constexpr string & vector  
std::string, std::vector 类型 现在可以作为 constexpr  
未来需要支持 constexpr 反射
```

原子智能指针

智能指针(shared\_ptr) 线程安全吗?

是: 引用计数控制单元线程安全, 保证对象只释放一次  
否: 对于数据的 读写 没有线程安全

如何将智能指针变成 线程安全?

使用 mutex 控制 智能指针的访问  
使用全局 非成员 原子操作函数 访问, 如 std::atomic\_load(), atomic\_store() 等  
缺点: 容易出错, 忘记这些操作  
C++20: atomic<shared\_ptr<T>>, atomic<weak\_ptr<T>>  
内部原理可能使用了 mutex  
全局 非成员 原子操作函数 标记位 不推荐使用

```
template<typename T>  
class concurrent_stack {  
    struct Node {  
        T t;
```

```

        shared_ptr<Node> next;
    };
    atomic_shared_ptr<Node> head;
    // C++11: 去掉 "atomic_" 并且在访问时, 需要用
    // 特殊的函数控制线程安全, 例如用std::atomic_load
public:
    class reference {
        shared_ptr<Node> p;
        <snip>
    };
    auto find(T t) const {
        auto p = head.load(); // C++11: atomic_load(&head)
        while (p && p->t != t)
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
        return reference(head);
    }
    void push_front(T t) {
        auto p = make_shared<Node>();
        p->t = t; p->next = head;
        while (!head.compare_exchange_weak(p->next, p)) {
    } // C++11: atomic_compare_exchange_weak(&head, &p->next, p); }
    void pop_front() {
        auto p = head.load();
        while (p && !head.compare_exchange_weak(p, p->next)) {
    } // C++11: atomic_compare_exchange_weak(&head, &p, p->next);
    }
};
```

自动合流(Joining), 可中断(Cancellable) 的线程

```

std::jthread
    头文件 <thread>
    支持中断
    析构函数中自动 join
        析构函数调用 stop_source.request_stop() 然后 join()
    中断线程执行
        头文件 <stop_token>
        std::stop_token
            用来查询线程是否中断
            可以和 condition_variable_any 配合使用
        std::stop_source
            用来请求线程停止运行
            stop_resources 和 stop_tokens 都可以查询到 停止请求
        std::stop_callback
            如果对应的 stop_token 被要求终止, 则会触发 回调函数
            用法: std::stop_callback myCallback(myStopToken, []{ /* ... */ });
```

例子

自动合流join

std::thread 在析构器中 如果线程 joinable() 则直接调用 std::terminate() 直接导致程序退出。

```

void DoWorkPreCpp20() {
    std::thread job([] { /* ... */ });
    try {
        // ... Do something else ...
    } catch (...) {
```

```

        job.join();
        throw; // rethrow
    }
    job.join();
}

void DoWork() {
    std::jthread job([] { /* ... */ });
    // ... Do something else ...
} // jthread destructor automatically calls join()

```

## 中断

```

std::jthread job([](std::stop_token token) {
    while (!token.stop_requested()) {
        //...
    });
//... job.request_stop();
// auto source = job.get_stop_source()
// auto token = job.get_stop_token()

```

## C++20 同步(synchronization)库

### 信号量semaphore

头文件<semaphore>

轻量级的 同步原语

可以用来实现 任何其他 同步概念: mutex, latches, barriers 等

两种类型:

多元信号量(counting semaphore)

二元信号量(binary semaphore)

### std::atomic 等待 和 通知接口

等待/阻塞 在原子对象 直到 其值 发生改变, 通过通知函数发送通知  
比 轮询(polling) 更高效

方法:

```

wait()
notify_one()
notify_all()

```

### 锁存器和屏障 (latch and barrier)

辅助线程 条件同步

### 锁存器 latch

头文件<latch>

线程的同步点

线程将阻塞在这个位置, 直到 到达的线程 个数 达标 才放行, 放行之后 不再关闭

锁存器 只会作用一次

### 屏障(barrier)

头文件<barrier>

多个阶段

每个阶段中

一个参与者 运行至 屏障点时 被阻塞, 需要等待 其他参与者 都到达屏障点, 当  
到达线程数达标之后

阶段完成 的回调 将被执行

线程计数器 被重置

开启下一阶段

线程得以继续执行

```
std::atomic_ref
头文件<atomic>
通过 引用访问 变为 原子操作，被引用对象 可以为 非原子类型。
```

其他更新

指定初始化(Designated Initializers)

```
struct Data {
    int anInt = 0;
    std::string aString;
};

Data d{ .aString = "Hello" };
```

<=>操作符

三路比较运算符

结果如下

```
(a <= b) < 0 // 如果 a < b 则为 true
(a <= b) > 0 // 如果 a > b 则为 true
(a <= b) == 0 // 如果 a 与 b 相等或者等价 则为 true
```

一般情况：自动生成 所有的比较操作符，如果对象是 结构体 则逐个比较，可以用下面代码代替 所有的 比较运算符

```
auto X::operator<=>(const Y&) = default;
```

高级情况：指定返回类型 (支持6种 所有的比较运算符)

		<b>a &lt; b supported</b>	
		Yes	No
<b>a==b implies f(a) == f(b)?</b>	Yes	<b>strong_ordering</b>	<b>strong_equality</b>
	No	<b>weak_ordering or partial_ordering</b>	<b>weak_equality</b>

**partial\_ordering:** allows incomparable values,  
i.e.  $x < y$ ,  $x > y$ , and  $x == y$  could all be false.

例子

```
class Point {
    int x; int y;
public:
    friend bool operator==(const Point& a, const Point& b) {
        return a.x==b.x && a.y==b.y;
    }
    friend bool operator< (const Point& a, const Point& b) {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    }
    friend bool operator!=(const Point& a, const Point& b) {
        return !(a==b);
    }
    friend bool operator<=(const Point& a, const Point& b) {
        return !(b<a);
    }
    friend bool operator> (const Point& a, const Point& b) {
        return b<a;
    }
    friend bool operator>=(const Point& a, const Point& b) {
        return !(a<b);
    }
}
```

```
// ... 其他非比较函数 ...
};

#include <compare>
class Point {
    int x; int y;
public:
    auto operator<=(const Point&) const = default; // 比较操作符自动生成
    // ... 其他非比较函数 ...
};


```

标准库类型支持 `<=>`  
vector, string, map, set, sub\_match 等

range for 循环语句支持 初始化语句  
for (auto data = GetData(); auto& value : data.values) {  
 // Use 'data'  
}

非类型模板形参支持字符串

```
template<auto& s> void DoSomething() {
    std::cout << s << std::endl;
}
int main() {
    DoSomething<"CppCon">();
}
```

[[likely]], [[unlikely]]  
先验概率 指导 编译器优化  
switch (value) {
 case 1: break;
 [[likely]] case 2: break;
 [[unlikely]] case 3: break;
}

日历(Calendar)和时区(Timezone)  
<chrono> 增加日历 和时区的支持  
只支持公历

其他日历 有可以通过 扩展加入，并能和 <chrono> 进行交互  
初始化 年月日的方法

```
// creating a year
auto y1 = year{ 2019 };
auto y2 = 2019y;
// creating a month
auto m1 = month{ 9 };
auto m2 = September;
// creating a day
auto d1 = day{ 18 };
auto d2 = 18d;
```

创建完整的日期

```
year_mouth_day fulldate1{2019y, September, 18d};
auto fulldate2 = 2019y / September / 18d;
year_mouth_day fulldate3{Monday[3]/September/2019}; // Monday[3] 表示第三个星期一
```

新的时间间隔单位，类似于 秒，分钟

```
using days = duration<signed integer type of at least 25bits,
                    ratio_multiply<ratio<24>, hours::period>>;
using weeks = ...; using months = ...;
```

```
using years = ...;
```

例子

```
weeks w{1}; // 1 周  
days d{w}; // 将 1 周 转换成天数
```

新的时钟类型 (之前有 system\_clock, steady\_clock, high\_resolution\_clock)

```
utc_clock:  
tai_clock:  
gps_clock:  
file_clock:
```

新增 system\_clock 相关的别名

```
template<class Duration>  
using sys_time = std::chrono::time_point<std::chrono::system_clock, Duration>;  
using sys_seconds = sys_time<std::chrono::seconds>;  
using sys_days = sys_time<std::chrono::days>;  
// 用例:  
system_clock::time_point t = sys_days{ 2019y / September / 18d }; // date ->  
time_point  
auto yearmonthday = year_month_day{ floor<days>(t) }; // time_point -> date
```

日期+时间

```
auto t = sys_days{2019y/September/18d} + 9h + 35min + 10s; // 2019-09-18 09:35:10  
UTC
```

时区转换

```
// Convert UTC to Denver  
time::zoned_time denver = { "America/Denver", t };  
// Construct a local time in Denver:  
auto t = zoned_time{  
    "America/Denver", local_days{Wednesday[3] / September / 2019} + 9h  
};  
// Get current local time:  
auto t = zoned_time{ current_zone(), system_clock::now() };
```

std::span

头文件<span>

某段连续数据的“视图”

不持有数据，不分配 和销毁数据

拷贝非常快，推荐复制的方式传参(类似 string\_view)

不支持数据跨步(stride)

可以通过 运行期 确定 长度 也可以编译期 确定长度

```
int data[42]; span<int, 42> a{data}; // fixed-size: 42 ints  
span<int> b{data}; // dynamic-size: 42 ints  
span<int, 50> c{data}; // compilation error  
span<int> d{ptr, len}; // dynamic-size: len ints
```

特性测试宏

可以判断 编译期是否支持 某个功能，例如

语言特性

```
_has_cpp_attribute(fallthrough)  
_cpp_binary_literals  
_cpp_char8_t  
_cpp_coroutines
```

标准库特性

```
__cpp_lib_concepts  
__cpp_lib_ranges  
__cpp_lib_scoped_lock
```

<version>

包含 C++ 标准版本，发布日期，版权证书，特性宏等

consteval 函数

constexpr 函数可能编译期执行，也可能运行期执行， consteval 只能在 编译期执行，如果不能满足要求 则编译失败。

constexpr

强制 指定 以常量方式 初始化

```
const char* GetStringDyn() {  
    return "dynamic init";  
}  
  
constexpr const char* GetString(bool constInit) {  
    return constInit ?  
        "constant init" :  
        GetStringDyn();  
}  
  
constexpr const char* a = GetString(true); // ✓  
constexpr const char* b = GetString(false); // ✗
```

用 using 引用 enum 类型

```
enum class CardTypeSuit {  
    Clubs,  
    Diamonds,  
    Hearts,  
    Spades  
};  
  
std::string_view GetString(const CardTypeSuit cardTypeSuit) {  
    switch (cardTypeSuit) {  
        case CardTypeSuit::Clubs:  
            return "Clubs";  
        case CardTypeSuit::Diamonds:  
            return "Diamonds";  
        case CardTypeSuit::Hearts:  
            return "Hearts";  
        case CardTypeSuit::Spades:  
            return "Spades";  
    }  
}  
  
std::string_view GetString(const CardTypeSuit cardTypeSuit) {  
    switch (cardTypeSuit) {  
        using enum CardTypeSuit; // 这里  
        case Clubs: return "Clubs";  
        case Diamonds: return "Diamonds";  
        case Hearts: return "Hearts";  
        case Spades: return "Spades";  
    }  
}
```

格式化库 std::format

```
std::string s = std::format("Hello CppCon {}!", 2019);
```

增加数学常量  
头文件<numbers>  
包含 e, log2e, log10e pi, inv\_pi, inv\_sqrt pi ln2, ln10 sqrt2, sqrt3, inv\_sqrt3  
egamma  
。。感觉逗号少了。

std::source\_location  
用于获取 代码位置，对于 日志 和错误信息 尤其有用

[[nodiscard(reason)]]  
表明返回值不可抛弃，加入reason的支持

```
[[nodiscard("Ignoring the return value will result in memory leaks.")]]  
void* GetData() { /* ... */ }
```

位运算  
加入循环移位，计数0和1位 等功能

一些小更新  
字符串支持 starts\_with, ends\_with  
map支持 contains 查询是否 存在某个键  
list 和 forward list 的 remove, remove\_if, unique 操作 返回 size\_type 表明删除的个数  
<algorithm> 增加 shift\_left, shift\_right  
midpoint 计算中位数，可以避免溢出  
lerp 线性插值 lerp(float a, float b, float t) 返回 a + t(b-a)  
新的向量化策略 unsequenced\_policy(execution::unseq)

```
std::string str = "Hello world!";  
bool b = str.starts_with("Hello"); // starts_with, ends_with  
std::map myMap{ std::pair{1, "one"s}, {2, "two"s}, {3, "three"s} };  
bool result = myMap.contains(2); // contains, 再也不用 .find() == .end() 了
```

=====

=====

=====

聚合类  
聚合初始化

=====

自定义字面值

=====

noexcept

=====

constexpr

=====

coroutine

=====

quoted

=====

exchange

=====

SFINAE

=====

模板

## Concept

类型安全

C++ 核心指南 支持库 (GSL)

如果我们想要定义一个只针对某个类型的函数模板，就只能通过类型萃取机制如 `enable_if_t` 写一些又臭又长的代码