C++ 11 标准

1. long long 类型

新增了类型 long long 和 unsigned long long ，以支持 64 位（或更宽）的整型。

在 vs中， int 和 long 都是 4 字节，long long 是8字节。

在 linux中，Int 是 4字节，long 和 long long 是8字节。

2. char16_t 和 char32_t 类型

新增了类型 char16_t 和 char32_t ，以支持 16位和32位的字符。

3. 统一初始化列表

C++ 11 丰富了大括号的使用范围，用大括号括起来的列表（统一的初始化列表）可以用于所有内置类型和用户自定义类型。使用统一的初始化列表时，可以添加等号（=），也可以不添加。

```
int x={5};
double y{2.5};
short quar[5]{4,5,2,76,1};
```

统一的初始化列表也可以用于 new 表达式中：

```
int* arr = new int[4]{2,4,6,7};
```

创建对象时，也可以使用大括号（而不是圆括号）来调用构造函数：

```
class CGirl
{
private:
    int m_bh;
    string m_name;
public:
    CGilr(int bh,string name):m_bh(bh),m_name(name) { }
};

CGirl g1(3,"西施");    //C++98 的风格
CGirl g2 = {5,"米米"};     // C++11 的风格
CGirl g3{8,"蜜蜜"};    // C++11 的风格
```

STL 容器提供了 initializer_list 模板类作为参数的构造函数：

```
vector<int> v1(10);      // 把v1初始化为10个int类型的元素
vetcor<int> v2{10};      // 把v2初始化为一个元素，这个元素是10
vector<int> v3{3,5,8};   // 把v3初始化为3个元素，值分别为3, 5, 8
```

头文件<initializer_list> 提供了对模板类 initializer_list的支持，这个类包含了成员函数 begin() 和 end() 。除了用于构造函数外，还可以将 initializer_list 用于常规函数的参数：

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

double sum(std::initializer_list<double> ii)
{
    double total = 0;
    for (auto it = ii.begin(); it != ii.end(); it++) {
        total = total + *it;
    }
    return total;
}

int main(void)
{
    //double total = sum( 3.14,5.20,8 );    // 错误，没有大括号，这是三个参数。
    double total = sum({ 3.14,5.20,8 });    // 正确，有大括号，这是一个参数。
    std::cout << "total=" << total << std::endl;
}
```

4. 模板的别名

对于冗长或复杂的标识符，如果能够创建其别名将很方便。以前，C++为此提供了 `typedef`：

```
typedef std::vector<std::string>::iterator itType;
```

C++11 提供了另一个创建别名的方法。

```
using itType = std::vector<std::string>::iterator;
```

差别在于，using方法也可以用于模板部分具体化，但typedef不能。

```
template<typename T>
using arr12 = std::array<T,12>;
```

上述语句的部分具体化模板 array<T,int> （将参数int设置为12）。例如，对于下述声明：

```
std::array<double, 12> a1;
std::array<std::string,12> a2;
```

可将它们替换为如下声明：

```
arr12<double> a1;
arr12<std::string> a2;
```

5. 空指针 nullptr

空指针是不会指向有效数据的指针。以前，C/C++ 用0表示空指针，这带来了一些问题，0 既可以表示指针常量，又可以表示整型常量。

```
#define NULL 0       // C++11 之前
```

C++11 新增了关键字 `nullptr`，用于表示空指针；**它是指针类型，不是整型类型。**

为了向后兼容，C++11 仍允许用0表示空指针，因此表达式 **nullptr==0 为 true** 。

使用 `nullptr` 提供了更高的类型安全。例如，可以将 0传递给形参为 `int` 的函数，但是，如果将 `nullptr` 传递给形参为 int 的函数，编译器将视为错误。

因此，出于安全和清晰考虑，使用**nullptr**。

6. 强类型枚举

传统的 C++ 枚举提供了一种**创建常量**的方法，但类型检查比较低级。还有，**如果在同一作用域内定义的两个枚举，它们的成员不能同名。**

针对枚举的缺陷，C++11 标准引入了枚举，又称**强类型枚举**。

声明强类型枚举非常简单，只需要**在 enum 后加上关键字 class** 。

```
enum e1{res,green};
enum class e2{red,green,blue};
enum class e3{red,green,blue,yellow};
```

使用强类型枚举时，要在枚举成员前面加上 **枚举名和 ::** ，以免发生名称冲突，例如： e2::red ， e3::blue

强类型枚举默认的类型是int，也可以显示地指定类型，具体做法是在枚举名后面加上 **:type** ，type 可以是**除 wchar_t 以外的任何类型。**

```
enum class e2:char { red, green, blue };
```

7. explicit 关键字

C++支持对象自动转换，但是，自动类型转换可能导致意外。为了解决这种问题，C++ 11 引入了 `explicit` ，用于关闭自动转换特性。

8. 类内成员初始化

在类的定义中初始化成员变量。

```
class CGirl
{
private:
    int m_bh=20;
    string m_name="西施";
    char m_xb ='X';
public:
    CGirl(int bh,string name,char xb):m_bh(bh),m_name(name),m_xb(xb){ }
};
```

9. 新的STL容器

array（静态数组）：array的大小是固定的，不像其他的模板类，但array有 begin() 和 end() 成员函数，程序员可以 array对象使用STL算法。

forward_list（单向链表）

unordered_map，unordered_multimap，unordered_set，unordered_multiset（哈希表）

10. 新的STL方法（成员函数）

C++ 11 新增的方法 `cbegin()` ，`cend()` ，`crbegin()` ，`crend()` ，这些方法**将元素视为 const**。

`iterator emplace(iterator pos, ...) ;` // 在指定位置插入一个元素， ... 用于构造元素，返回指向插入元素的迭代器

更重要的是，除了传统的拷贝构造函数和赋值函数，C++11 新增了**移动构造函数和移动赋值函数**。

11. 摒弃 export

C++ 98 新增了 export 关键字，**C++11 不再使用**，但仍保留它作为关键字，供以后使用。

12. 嵌套模板的尖括号

为了避免与运算符 >> 混淆，C++ 要求在**声明嵌套模板时使用空格将尖括号分开**。

```
vector<list<int> > v1;      // 两个 > 之间必须加空格
```

C++11 不再这样要求：

```
vector<list<int>> v2;       // 两个 > 之间不必加空格
```

13. final 关键字

`final`关键字用于限制某个类不能被继承，或者某个虚函数不能被重写。

`final` 关键字放在类名或虚函数名的后面。

```
class AA
{
```

```
public:
    virtual void test()
    {
        cout << "AA class...";
    }
};

class BB : public AA
{
public:
    void test() final    // 如果有其它类继承BB，test()方法将不允许重写。
    {
        cout << "BB class...";
    }
};

class CC : public BB
{
public:
    void test()  // 错误，BB类中的test()后面有final，不允许重写。
    {
        cout << "CC class...";
    }
};
```

14. override 关键字

在**派生类**中，把 `override` **放在成员函数的后面，表示重写基类的虚函数**，提高代码可读性。

```
class AA
{
public:
    virtual void test()
    {
        cout << "AA class...";
    }
};

class BB : public AA
{
public:
    void test() override
    {
        cout << "BB class...";
    }
};
```

在派生类中，如果某成员函数不是重写基类的虚函数，随意加上 `override` 关键字，编译器会报错。

15. 数值类型和字符串之间的转换

传统方法用 `sprintf()` 和 `snprintf()` 函数把数值转换为 char* 字符串；用 `atoi()` , `atol()` , `atof()` 把 char* 字符串转换为数值。

C++11 提供了新方法，在**数值类型和string字符串之间的转换**。

15.1 数值转换为字符串

使用 **to_string()** 函数可以将**各种数值类型转换为 string 字符串类型**，这是一个重载函数，在头文件 `<string>` 中声明，函数原型如下：

```
string to_string (int val);
string to_string (long val);
string to_string (long long val);
string to_string (unsigned val);
string to_string (unsigned long val);
string to_string (unsigned long long val);
string to_string (float val);
string to_string (double val);
string to_string (long double val);
```

15.2 字符转换为数值类型

在C++中，数值类型包括整型和浮点型，针对于不同的数值类型提供了不同的函数在头文件 `<string>` 中声明，函数原型如下：

```
int         stoi( const string& str, size_t* pos = nullptr, int base = 10 );

long        stol( const string& str, size_t* pos = nullptr, int base = 10 );

long long      stoll( const string& str, size_t* pos = nullptr, int base = 10 );

unsigned long    stoul( const string& str, size_t* pos = nullptr, int base = 10 );

unsigned long long stoull( const string& str, size_t* pos = nullptr, int base = 10 );

float        stof( const string& str, size_t* pos = nullptr );

double       stod( const string& str, size_t* pos = nullptr );

long double     stold( const string& str, size_t* pos = nullptr );
```

形参说明：

> str：需要要转换的string字符串。
>
> pos：传出参数，存放从哪个字符开始无法继续解析的位置，例如：123a45，传出的位置将为3。
>
> base：若base为0，则自动检测数值进制：若前缀为0，则为八进制，若前缀为0x或0X，则为十六进制，否则为十进制。
>
> typedef unsigned long long size_t ;

注意：string字符串转换为数值的函数可能会抛出异常，

```
std::string str = "123a45";
size_t pos;
int val = stoi(str, &pos, 10);
std::cout <<"val=" << val << std::endl; // 输出123
std::cout << "pos=" << pos << std::endl;    // 输出3
```

16. 常量表达式 constexpr 关键字

`const`关键字从功能上来说有双重语义：**只读变量和修饰常量**

```
void func(const int len1)
{   // 给函数的形参加上const，表示只读变量，调用函数时，实参可能是变量，也可能是常量
    // len1是只读变量，不是常量
    int array1[len1] = { 0 };   // VS报错，Linux平台的数组长度支持变量，不会报错。
    // 在vs中，定义数组的时候，长度必须是常量表达式，不能是变量

    constexpr int len2 = 8;     // 定义变量的时候，加上const，len2才是真正的常量
    int array2[len2] = { 0 };   // 正确，len2是常量
}
```

C++11 标准为了解决const 关键字的双重语义问题，**保留了`const`表示“只读”的语义**，而**将“常量”的语义划分给了新添加的`constexpr`**

所以，C++ 11 标准中，建议将`const`（只读） 和`constexpr`（常量）的功能分开， **表达“只读”语义的场景用const，表达“常量”语义的场景用 constexpr**

```
constexpr int len2 = 8;
```

16.1 const与constexpr的区别

1. **const**：`const`用于定义一个只读的值。它可以用于变量、函数参数和函数返回值。例如：

```
const int x = 5; // x是一个只读的常量
void foo(const int y); // 函数参数y是只读的
const int bar(); // 函数返回值是只读的
```

2. **constexpr**：`constexpr`用于声明在编译时就可以计算出结果的表达式或函数。它通常用于在编译时进行优化，以便在运行时之前计算表达式的值。例如：

```
constexpr int square(int x) { return x * x; }
const int y = square(5); // 在编译时计算square(5)的值并将结果存储在y中
```

注意，`constexpr`要求表达式在编译时就能够确定其值，因此它通常用于简单的、不涉及运行时信息的情况。

总结起来，`const`表示只读性，`constexpr`表示在编译时计算的常量。虽然它们在一些情况下可以一起使用，但它们的主要目的和用途是不同的。

17. 默认函数控制 =default 和 =delete

在C++中自定义的类，编译器会默认生成一些成员函数：

> 无参构造函数
>
> 拷贝构造函数（浅拷贝）
>
> 拷贝赋值函数
>
> 移动构造函数
>
> 移动赋值函数

> 析构函数

=default表示启用默认函数。

=delete表示禁用默认函数。

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

class CGirl {
private:
    int m_bh=20;
    string m_name="西施";
    char m_xb = 'X';
public:
    CGirl() = default;       // 启用默认构造函数
    CGirl(int bh,string name,char xb):m_bh(bh),m_name(name),m_xb(xb) { }
    CGirl(const CGirl& g) = delete;      // 删除拷贝构造函数
    void show() { cout << "bh=" << m_bh << ",m_name=" << m_name << endl; }
    //~CGirl() = delete;    // 删除析构函数
};

// 如果我们提供了构造函数，那么编译器就不会提供默认构造函数


int main(void)
{
    CGirl g1;
    g1.show();
    CGirl g2 = g1;   // 错误，拷贝构造函数已删除。
    return 0;
}
```

18. 委托构造和继承构造

18.1 委托构造

在实际开发中，为了满足不同的需求，一个类可能会重载多个构造函数。多个构造函数之间可能会有重复的代码。例如变量初始化，如果在每个构造函数中都写一遍，这样的代码会显得臃肿。

委托构造就是在一个构造函数的初始化列表中调用另一个构造函数。

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

class AA {
private:
    int m_a;
    int m_b;
    double m_c;
public:
    // 有一个参数的构造函数，初始化m_c
    AA(double c)
    {
        m_c = c + 3;
        cout << "AA(double c)" << endl;
    }
    // 有两个参数的构造函数，初始化 m_a 和 m_b
    AA(int a, int b)
    {
        m_a = a + 1;
        m_b = b + 2;
        cout << "AA(int a, int b)" << endl;
    }
    // 构造函数委托AA(int a, int b)初始化m_a和 m_b
```

```
    AA(int a, int b, const string& str):AA(a,b)
    {
        cout << "m_a=" << m_a << ",m_b=" << m_b << ",str=" << str << endl;
    }

    // 构造函数委托AA(double c)初始化 m_c
    AA(double c, const string& str):AA(c)
    {
        cout << "m_c=" << m_c << ",str=" << str << endl;
    }
};

int main(void)
{
    AA a1{ 10,20,"我是一只傻傻鸟" };

    AA a2{ 3.8,"我是一只傻傻鸟" };
}
```

注意：

> 不要生成环状的构造函数。
>
> 一旦使用委托构造，就不能在初始化列表中再次初始化其他成员变量。
>
> ```
> AA(double c, const string& str):AA(c),m_a(0)
> ```
>
> 这里 构造函数委托AA(double c)初始化 m_c，就不能在初始化列表中初始化其他成员变量（可以在函数体内进行初始化）

这是因为委托构造函数的目的是为了将对象初始化的责任委托给其他构造函数。当你使用委托构造函数时，**它会在执行委托前，先执行默认的成员初始化操作，然后再执行你自己的构造逻辑**。如果允许在委托构造函数中使用初始化列表初始化其他成员变量，就会导致混乱，因为这样可能会出现一些不一致的初始化操作。

```
class MyClass {
public:
    MyClass(int value) : member1(value) { } // 委托构造函数
    MyClass() : MyClass(0), member2(0) { } // 错误，委托构造后再次在初始化列表中初始化其他成员
private:
    int member1;
    int member2;
};
```

在这个例子中，`MyClass(int value)` 是一个委托构造函数，它将成员 `member1` 初始化为指定的值。然后，我们试图在另一个构造函数 `MyClass()` 中使用委托构造函数，并在初始化列表中初始化了 `member2`。这是不允许的，因为在委托构造函数调用前，`member2` 已经通过默认构造函数初始化过了。

所以，为了保持构造过程的一致性和可预测性，C++ 标准规定了委托构造函数不能再次在初始化列表中初始化其他成员变量。

18.2 继承构造

在C++11之前，派生类如果要使用基类的构造函数，可以在派生类构造函数的初始化列表中指定。

C++11 推出了继承构造（Inheriting Constructor）,**在派生类中使用uisng 来声明继承基类的构造函数**。

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

class AA {
public:
    int m_a;
    int m_b;

    AA(int a) :m_a(a) { cout << "AA(int a)" << endl; }
```

```
    AA(int a, int b) :m_a(a), m_b(b) { cout << "AA(int a,int b)" << endl; }
};

class BB :public AA {
public:
    double m_c;
    using AA::AA;   // 使用基类的构造函数
    BB(int a, int b, double c) :AA(a, b), m_c(c) { cout << "BB(int a,int b,double c)" << endl;
}
    void show() { cout << "m_a=" << m_a << ",m_b=" << m_b << ",m_c=" << m_c << endl; }
};

int main(void)
{
    BB b1(10);  // 将使用基类有一个参数的构造函数，来初始化m_a
    b1.show();

    BB b2(10,20);   // 将使用基类有两个参数的构造函数，来初始化m_a,m_b
    b2.show();

    BB b3(10,20,3.56);  // 将使用派生类有三个参数的构造函数，来初始化m_a,m_b,m_c
    b3.show();
}
```

19. lambda函数

`lambda` 函数是 C++11 标准新增的语法糖，也称为 `lambda` 表达式或匿名对象。

`lambda` 函数的特点是：距离近，简洁，高效，功能强大。

例如： -> void {cout<<"亲爱的"<<no<<"号，我是一只傻傻鸟"<<endl;};

语法：

捕获列表	参数列表	函数选项	返回类型	函数体
[capture list]	(parameters)	mutable noexcept	->return type	{ statement }

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

// 表白函数
void zsshow(const int& no)
{
    cout << "亲爱的" << no << "号：我是一只傻傻鸟" << endl;
}

// 表白仿函数
class czs {
public:
    void operator()(const int& no)
    {
        cout << "亲爱的" << no << "号：我是一只傻傻鸟" << endl;
    }
};

int main(void)
{
    vector<int> vv = { 5,8,3 }; // 存放超女编号的容器

    // 第三个参数是普通函数
    for_each(vv.begin(), vv.end(), zsshow);

    // 第三个参数是仿函数
    for_each(vv.begin(), vv.end(), czs());
```

```
    // 第三个参数是Lambda表达式
    //for_each(vv.begin(), vv.end(),
    //  [](const int& no) {
    //      cout << "亲爱的" << no << "号：我是一只傻傻鸟" << endl;
    //  }
    //);
    // 采用lambda表达式不需要提前准备好函数或仿函数，用到的时候，临时写一段代码就好了

    auto f = [](const int& no) {
        cout << "亲爱的" << no << "号：我是一只傻傻鸟" << endl;
    };
    // 第三个参数是Lambda表达式
    for_each(vv.begin(), vv.end(),f);
    // 采用lambda表达式不需要提前准备好函数或仿函数，用到的时候，临时写一段代码
    f(333);      // 像使用普通函数一样，使用Lambda函数
}
```

19.1 参数列表

参数列表是可选的，类似普通函数的参数列表，如果没有参数列表，()可以省略不写。

与普通函数的不同：

> Lambda函数 不能有默认参数
>
> 所有的参数必须有参数名。
>
> 不支持可变参数

19.2 返回类型

用后置的方法书写返回类型，类似于普通函数的返回类型，如果不写返回类型，Lambda函数会根据函数体中的代码推断出来。

如果有返回类型，建议显示的指定，自动推断可能与预期不一样。

19.3 函数体

类似于普通函数的函数体。

19.4 捕获列表

通过捕获列表，`Lambda`函数可以访问父作用域中的非静态局部变量（静态局部变量可以直接访问）

捕获列表在书写在[]中，与函数参数的传递相似，捕获列表也可以是值或者引用。（本质就是向函数传递参数）

如果要捕获多个变量，在捕获列表中把要想捕获的变量名写出来，中间用逗号隔开。

> 在C++中，父作用域是指一个包含当前作用域的外部作用域。换句话说，父作用域是包围当前代码块的更大的作用域。这种嵌套的层级关系在代码的不同部分定义了标识符（如变量、函数名等）的可见性和范围。
>
> 在C++中，代码块、函数、类和命名空间都是作用域的例子。每个代码块都有一个父作用域，通常是包含该代码块的外部代码块。函数的父作用域可以是全局作用域，类的父作用域可以是命名空间，命名空间的父作用域可以是全局作用域。

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

int main(void)
{
    int ii = 3;
    double dd = 3.8;
```

```
    auto f = [ii,dd](const int& no) {
        cout << "ii=" << ii << endl;
        cout << "dd=" << dd << endl;
        cout << "亲爱的" << no << "号，我是一只小小鸟" << endl;
    };

    //! lambda 函数的函数体内，可以直接使用父作用域中的局部变量(将变量名写在捕获列表中)
    //! 如果要捕获多个变量，在捕获列表中把要想捕获的变量名写出来，中间用逗号隔开
    f(8);
    return 0;
}
```

以下列出了不同的捕获列表的方式：

lambda捕获方式	
[ ]	空捕获列表。lambda不能使用所在函数中的变量。
[names]	names是一个逗号分隔的名字列表，这些名字都是lambda所在函数的局部变量。默认情况下是值捕获，名字前加&指明是引用捕获。
[=]	隐式捕获列表，采用值捕获方式。lambda体将拷贝所使用的来自所在函数的实体都值。
[&]	隐式捕获列表，采用引用捕获方式。lambda体中所使用的来自所在函数的实体都采用引用方式使用。
[&, identifier_list]	identifier_list是一个逗号分隔的列表，包含0个或多个来自所在函数的变量。这些变量采用值捕获方式，而任何隐式捕获的变量都采用引用捕获。identifier_list中的名字前面不能使用&
[=, identifier_list]	identifier_list中的变量都采用引用方式捕获，而任何隐式捕获的变量都采用值捕获。identifier_list中的名字不能包括this，且这些名字之前必须使用&

注意：**如果是值捕获，在Lambda函数中不允许修改变量的值。如果是 [&name]引用捕获，可以在lambda函数体中修改变量。**

如果是 [=] 捕获（值捕获的隐式方式），如果lambda函数体内用到了ii和dd，那么这个 [=] 相当于 [ii,dd] ；如果lambda函数体内只用到了dd，那么 [=] 相当于 [dd]；如果lambda函数体内什么都没用到，那么 [=] 就相当于 [ ] 。

[=] 编译器根据lambda函数中的代码，自动判断需要捕获的变量，都是以值捕获的方式，在Lambda函数中不允许修改变量的值。

[&] 编译器自动判断需要捕获的变量，但是都是以引用捕获的方式，可以在lambda函数体中修改变量。

[=,&dd] 所有变量以值捕获的隐式方式，但变量dd是引用捕获

[&,dd] 所有变量以引用捕获的隐式方式，但变量dd是值捕获

1. 值捕获

与传递参数类似，**采用值捕获的前提是变量可以拷贝。**

与传递参数不同，**变量的值是在lambda函数创建时拷贝**，而不是调用时拷贝。

例如：

size_t v1 = 42; // size_t 是 unsigned long long

auto f = [v1] { return v1; }; // 使用了值捕获，将v1拷贝到名为f的可调用对象。

v1 = 0;

auto j = f(); // j为42，f保存了我们创建它是v1的拷贝。

由于被捕获的值是在lambda函数创建时拷贝，因此在随后对其修改不会影响到lambda内部的值。

默认情况下，如果以传值方式捕获变量，则在lambda函数中不能修改变量的值。

2. 引用捕获

和函数引用参数一样，引用变量的值在lambda函数体中改变时，将影响被引用的对象。

```
size_t v1 = 42;
auto f = [ &v1 ] { return v1; };  // 引用捕获，将v1拷贝到名为f的可调用对象。
v1 = 0;
auto j = f();  // j为0。
```

如果采用引用方式捕获变量，就必须保证被引用的对象在lambda执行的时候是存在的。

3. 隐式捕获

除了显式列出我们希望使用的父作域的变量之外，还可以让**编译器根据函数体中的代码来推断需要捕获哪些变量，这种方式称之为隐式捕获。**

隐式捕获有两种方式，分别是[=]和[&]。[=]表示以值捕获的方式捕获外部变量，[&]表示以引用捕获的方式捕获外部变量。

```
int a = 123;
auto f = [ = ] { cout << a << endl; };   //值捕获
f();   // 输出：123
auto f1 = [ & ] { cout << a++ << endl; };   //引用捕获
f1();  //输出：123（采用了后++）
cout << a << endl;     //输出 124
```

4. 混合方式捕获

lambda函数还支持混合方式捕获，**即同时使用显式捕获和隐式捕获。**

混合捕获时，**捕获列表中的第一个元素必须是 = 或 &**，此符号指定了**默认捕获的方式是值捕获或引用捕获**。

需要注意的是：**显式捕获的变量必须使用和默认捕获不同的方式捕获**。例如：

```
int i = 10;
int j = 20;
auto f1 = [ =, &i] () { return j + i; };   // 正确，默认值捕获，显式是引用捕获
auto f2 = [ =, i] () { return i + j; };    // 编译出错，默认值捕获，显式值捕获，冲突了
auto f3 = [ &, &i] () { return i +j; };   // 编译出错，默认引用捕获，显式引用捕获，冲突了
```

5. 修改值捕获变量的值

在lambda函数中，如果以传值方式捕获变量，则函数体中不能修改该变量，否则会引发编译错误。

在lambda函数中，**如果希望修改值捕获变量的值，可以加mutable选项**，但是**，在lambda函数的外部，变量的值不会被修改。**

```
int a = 123;
auto f = [a]() mutable ->void {cout << ++a << endl; }; // 不会报错
cout << a << endl;  // 输出：123
f();            // 输出：124
cout << a << endl;  // 输出：123
```

6. 异常说明

lambda可以抛出异常，用throw(…)指示异常的类型，用noexcept指示不抛出任何异常。

19.7 lambda函数的本质

当我们编写了一个lambda函数之后，编译器将它翻译成一个类，该类中有一个重载了()的函数。

1. 采用值捕获

采用值捕获时，lambda函数生成的类用捕获变量的值初始化自己的成员变量。

例如：

```
int a =10;

int b = 20;

auto addfun = [=] (const int c ) -> int { return a+c; };

int c = addfun(b);

cout << c << endl;
```

等同于：

```
class Myclass

{

  int m_a;     // 该成员变量对应通过值捕获的变量。

public:

  Myclass( int a ) : m_a(a){};  // 该形参对应捕获的变量。

  // 重载了()运算符的函数，返回类型、形参和函数体都与lambda函数一致。

  int operator()(const int c) const

  {

    return a + c;

  }

};
```

默认情况下，**由lambda函数生成的类是const成员函数，所以变量的值不能修改。如果加上mutable，相当于去掉const。这样上面的限制就能讲通了。**

2. 采用引用捕获

如果lambda函数采用引用捕获的方式，编译器直接引用就行了。

唯一需要注意的是，**lambda函数执行时，程序必须保证引用的对象有效。**

20. 右值引用

20.1 左值，右值

在C++中，所有的值不是左值，就是右值。**左值是指表达式结束后依然存在的持久化对象，右值是指表达式结束后就不再存在的临时对象。有名字的对象都是左值，右值没有名字。**

还有一个可以区分左值和右值的便捷方法：**看能不能对表达式取地址，如果能取地址，则为左值；否则为右值。**

C++ 11 扩展了右值的概念，将**右值分为了纯右值和将亡值。**

> 纯右值：非引用返回的临时对象　　运算表达式产生的结果　　字面常量（c风格字符串除外，它是地址）
>
> 将亡值：与右值引用相关的表达式，例如：将要移动的对象，T&&函数返回的值，std::move() 的返回值，转换成T&& 的类型的转换函数的返回值

```cpp
#include <iostream>
using namespace std;

class AA {
    int m_a;
};

AA getTemp()
{
    return AA();    //返回AA匿名对象
}

int main(void)
{
    int ii = 3; // ii是左值，3是右值

    int jj = ii + 8;    // jj是左值， ii+8是右值（ii+8没有名字）

    AA aa = getTemp();  // aa是左值， getTemp()返回的是右值（临时对象，没有名字）
};
```

20.2 左值引用，右值引用

C++ 98 中的引用很常见，就是给变量取一个别名，在C++11中，因为增加了右值引用(rvalue reference)的概念，所以**C++98中的引用都统称为左值引用（lvalue reference）。**

右值引用就是给右值取个别名，使用的符号是&&

```cpp
#include <iostream>
using namespace std;

class AA {
public:
    int m_a=9;
};

AA getTemp()
{
    return AA();    //返回AA匿名对象
}

int main(void)
{
    const int&& a = 3;  // 3是右值，给3取个名字，叫a（a是左值）

    int b = 8; // b是左值
    int&& c = b + 5;    // b+5是右值，给b+5取个名字，叫c（c是左值）

    AA&& aa = getTemp();    // getTemp()的返回值是右值（临时变量），给它取个名字，叫aa（aa是左值）
    // getTemp()函数返回值，本来在表达式语句结束后，其生命周期也就该终结了，
    //但是，通过右值引用获得了新生，它的生命周期将与右值引用类型变量aa的生命周期一样
    // （只要aa还活着，该右值临时变量就会一直存活下去）

    //! 右值有了名字之后，就成了普通变量，可以像使用左值一样使用它们,可以取地址...
    // 为什么要给临时变量续命？在某些场景中临时变量还有继续利用的价值。
    cout << "a=" << a << endl;
    cout << "c=" << c << endl;
    cout << "aa.m_a=" << aa.m_a << endl;
    cout << "&a=" << &a << endl;
}
```

`getTemp()` 函数返回值，本来在表达式语句结束后，其生命周期也就该终结了，但是，通过右值引用获得了新生，它的生命周期将与右值引用类型变量aa的生命周期一样。只要aa还活着，该右值临时变量就会一直存活下去。

右值有了名字之后，就成了普通变量，可以像使用左值一样使用它们,可以取地址...

引入右值引用的主要目的是实现移动语义。

左值引用只能绑定（关联，指向）左值，右值引用只能绑定右值，如果绑定的不对，编译就会失败。

但是，**常量左值引用却是个特例**，它可以算是一个**万能的引用类型**，它可以绑定**非常量左值，常量左值，右值**，而且在绑定右值的时候，常量左值引用还可以像右值引用一样将右值的生命周期延长，**缺点是只能读不能改**。

```
int a = 1;
const int& ra = a;  // a是非常量左值

constexpr int b = 1;    // 常量
const int& rb = b;      // b 是常量左值

const int& rc = 1;  // 1是右值
```

21. 移动语义

如果一个对象中有堆区资源，需要编写拷贝构造函数和赋值函数，实现深拷贝。

深拷贝把对象中的堆区资源复制了一份，如果源对象（被拷贝的对象）是临时对象，拷贝完就没什么用了，这样会造成没有意义的资源申请和释放操作。如果能够直接使用源对象所拥有的资源，可以节省资源申请和释放的时间。c++11 新增的移动语义就能够做到这一点。

实现移动语义要增加两个函数：移动构造函数和移动赋值函数。

移动构造函数的语法：

```
类名(类名&& 源对象){...}
```

移动赋值函数的语法：

```
类名 operator=(类名&& 源对象){...}
```

注意：

- 对于一个左值，会调用拷贝构造函数，但是有些左值是局部变量，生命周期也很短，能不能也移动而不是拷贝呢？

  - C++11为了解决这个问题，提供了 `std::move()` 方法来 **将左值转义为右值** ，从而方便使用移动语义。它其实 **告诉编译器，虽然我是一个左值，但不要对我使用拷贝构造函数，用移动构造函数吧。左值对象被转移资源后，不会立刻析构** ，只有在离开自己的作用域的时候才会析构，如果继续使用左值中的资源，可能会发生意想不到的错误。

```
AA a1;
a1.alloc(); // 分配堆区内存
*a1.m_date = 3; // 给堆区内存赋值
cout << "*a1.m_date=" << *a1.m_date << endl;

AA a2 = move(a1);   // 调用移动构造函数
cout << "*a2.m_date=" << *a2.m_date << endl;

AA a3;
a3 = move(a1);  // 调用移动赋值函数
cout << "*a3.m_date=" << *a3.m_date << endl;
// 结果会报错，程序奔溃掉了，为什么？因为a1这个资源被转移了两次，第一次是成功的，第二次不会成功。
```

- **如果没有提供移动构造函数/赋值函数，只提供了拷贝构造函数/赋值函数，编译器找不到移动构造函数/赋值函数，就会去寻找拷贝构造函数/赋值函数。**
- C++ 11中的所有容器都实现了移动语义，避免对含有资源的对象发生无谓的拷贝。
- 移动语义对于拥有资源（如堆区内存，文件句柄，网络连接）的对象有效，如果是基本类型，使用移动语义没有意义。
- 在c++11中，把拷贝数的操作叫拷贝语义。移动语义的意思是：不要拷贝，直接把资源转移过来（转移资源的操作叫移动语义）
  不同的类，转移资源的方法是不同的，移动构造函数和移动赋值函数的代码的写法也可能不一样，但是移动的语义是类似的

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

class AA {
public:
    int* m_date = nullptr;  // 数据成员，指向堆区资源的指针

    AA() = default; //启用默认构造函数

    void alloc()    // 给数据成员m_data分配内存
    {
        m_date = new(nothrow) int;  // 分配内存
        memset(m_date, 0, sizeof(int)); // 初始化已分配的内存
    }

    AA(const AA& a) // 拷贝构造函数
    {
        cout << "调用了拷贝构造函数" << endl;
        if (m_date == nullptr) {
            alloc();    // 如果没有分配内存，就分配
        }
        memcpy(m_date, a.m_date, sizeof(int));  //把数据从源对象中拷贝过来
    }

    AA(AA&& a)  // 移动构造函数，需要操作源对象（被转移对象）的指针，所以形参不能用const
    {
        cout << "调用了移动构造函数" << endl;
        if (m_date != nullptr) {
            delete m_date;  // 如果已分配内存，先释放掉，把指针空出来，准备指向新的资源
        }
        m_date = a.m_date;  //把资源从源对象中转移过来
        a.m_date = nullptr; // 把源对象中的指针置空，它不再拥有资源
    }

    AA& operator=(const AA& a)  // 赋值函数
    {
        cout << "调用了赋值函数" << endl;
        if (this == &a) {   // 避免自我赋值
            return *this;
        }
        if (m_date == nullptr) {    // 如果没有分配内存，就分配
            alloc();
        }
        memcpy(m_date, a.m_date, sizeof(int));  //把数据从源对象中拷贝过来
        return *this;
    }


    AA& operator=(AA&& a)   // 移动赋值函数
    {
        cout << "调用了移动赋值函数" << endl;
        if (this == &a) {   // 避免自我赋值
            return *this;
        }
        if (m_date != nullptr) {    // 如果已分配内存，先释放掉，把指针空出来，准备指向新的资源
            delete m_date;
```

```
        }
        m_date = a.m_date;  // 把资源从源对象中转移过来
        a.m_date = nullptr; // 把源对象的指针置空，它不再拥有资源
        return *this;
    }
    ~AA()   // 析构函数
    {
        if (m_date != nullptr){
            delete m_date;
            m_date = nullptr;
        }
    }
};

//在c++11中，把拷贝数的操作叫拷贝语义。
//移动语义的意思是：不要拷贝，直接把资源转移过来（转移资源的操作叫移动语义）

// 不同的类，转移资源的方法是不同的，，移动构造函数和移动赋值函数的代码的写法也可能不一样，但是移动的语义是类似的

int main(void)
{
    AA a1;
    a1.alloc(); // 分配堆区内存
    *a1.m_date = 3; // 给堆区内存赋值
    cout << "*a1.m_date=" << *a1.m_date << endl;

    AA a2 = a1; // 调用拷贝构造函数
    cout << "*a2.m_date=" << *a2.m_date << endl;

    AA a3;
    a3 = a1;    // 调用赋值函数
    cout << "*a3.m_date=" << *a3.m_date << endl;
    cout << "-----------------------------------" << endl;

    auto f = []()->AA
    {
        AA aa;
        aa.alloc();
        *aa.m_date = 8;
        return aa;
    };      // 返回AA类对象的lambda函数

    AA a4 = f();    // 函数以值的方式返回AA类的临时对象，就是一个右值，将调用移动构造函数
    cout << "*a4.m_date=" << *a4.m_date << endl;

    AA a5;
    a5 = f();   // 调用移动赋值函数
    cout << "*a5.m_date=" << *a5.m_date << endl;
}
```

```
调用了移动构造函数
*a4.m_date=8
调用了移动构造函数
调用了移动赋值函数
*a5.m_date=8
```

a5 = f(); 这里为什么会调用移动构造函数？（VS中）

```
&a5=0000006E97D7FA28
对象aa的地址是0000006E97D7F8C8
this=0000006E97D7FB28
调用了移动构造函数
&a=0000006E97D7FB28
调用了移动赋值函数
*a5.m_date=8
```

> AA& operator=(AA&& a)　移动赋值函数形参AA&& a ， lambda函数f()返回临时对象aa，AA&& a=aa，调用了移动构造函数。
>
> 之后 a5=f() 调用移动赋值函数

在linux中，只调用了移动赋值函数。

> linux中，g++做了优化， 移动赋值函数中的形参 a与 f()函数返回的临时对象aa的地址是一样的。

22. 完美转发

在函数模板中，可以将自己的参数“完美”地转发给其他函数。所谓完美，即**不仅能准确地转发参数的值，还能保证被转发参数的左，右值属性不变。**

C++11 标准引入了右值引用和移动语义，所以，能否实现完美转发，决定了该参数在传递过程使用的是拷贝语义（调用拷贝构造）还是移动语义（调用移动构造函数）。

右值有了名字之后，就成了左值

为了支持完美转发，c++11提供了以下方案：

1. **如果函数模板的参数类型是 `T&&` （这不是右值引用），既可以接受左值引用，又可以接受右值引用。**
2. **提供了模板函数 `std::forward<T>(参数)` ，用于转发参数，如果参数是一个右值，转发之后仍然是右值引用；如果参数是一个左值，转发之后仍然是左值引用。**

> // func2()把参数传给func1()的时候，也要有左值和右值之分
>
> 编译器可以区分左值和右值，调用模板的时候，它把参数的左，右值属性存起来。
> forward()模板函数再读取存起来的信息，然后转发出去

```
template<typename T>
void func2(T&& ii)
{
    func1(forward<T>(ii));
}
```

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

void func1(int& ii) // 如果参数是左值，调用此函数
{
    cout << "参数是左值=" << ii << endl;
}

void func1(int&& ii)    // 如果参数是右值，调用此函数
{
    cout << "参数是右值=" << ii << endl;
}

//void func2(int& ii)   // 左值版本
//{
//  func1(ii);
//}
```

```
//void func2(int&& ii)  // 右值版本
//{
//  func1(move(ii));
//}

// 我们希望的是：func2()把参数传给func1()的时候，也要有左值和右值之分

template<typename T>
void func2(T&& ii)
{
    func1(forward<T>(ii));
}

// 编译器可以区分左值和右值，调用模板的时候，它把参数的左，右值属性存起来。
// forward()模板函数再读取存起来的信息，然后转发出去

int main(void)
{
    int ii = 3;
    func2(ii);  // 将调用左值函数
    func2(8);   // 将调用右值函数
}
```

23. 可变参数模板

可变参数模板是C++11新增的最强大的特性之一，它对参数进行了泛化，能支持任意个数，任意数据类型的参数。

在C++中，可变参数函数的声明通常使用省略号 `...` 放在参数列表的前面，而在函数调用时，`...` 会放在参数列表的后面，用于表示函数可以接受不定数量的参数。这样的函数通常需要使用一些特殊的语法来处理参数的传递和访问，比如使用 `va_list` 或C++11引入的 `std::initializer_list` 等方式。感谢你的补充。

```
#include <iostream>
#include <thread>
using namespace std;

template <typename T>
void show(T girl)       // 向超女表白的函数，参数可能是超女编号，也可能是姓名，所以用T。
{
    cout << "亲爱的" << girl << "，我是一只傻傻鸟。\n";
}

// 递归终止时调用的非模板函数，函数名要与展开参数包的递归函数模板相同。
void print()
{
    cout << "递归终止。\n";
}

// 展开参数包的递归函数模板。（整体思路：用递归函数把参数包一次一次的展开）
template <typename T, typename ...Args> // 第一个是普通模板参数，第二个参数中间有三个圆点，表示可变参数
void print(T arg, Args... args)      //TODO:这是多个参数
{ // 第一个参数 arg用T来定义，表示已展开的参数，也就是这一次取出来的参数
    // 第二个参数args用Args来定义，表示尚未展开的参数包

    //cout << "参数:  " << arg << endl;          // 显示本次展开的参数。

    show(arg);          // 把参数用于表白。

    //cout << "还有" << sizeof...(args) << "个参数未展开。" << endl;  // 显示未展开变参的个数。

    print(args...);      // 继续展开参数。   //TODO: args... 这是多个参数
}
//! print()是递归函数，每调用一次，就从参数包中取一个参数
// 参数包展开之后，你想怎么用就怎么用

template <typename...Args>
void func(const string& str, Args...args)   // 除了可变参数，还可以有其它常规参数。
```

```
{
    cout << str << endl;    // 表白之前，喊句口号。

    print(args...);    // 展开可变参数包。

    cout << "表白完成。\n";
}

// 可变参数模板特殊的地方，递归终止条件，和普通递归终止不一样，是参数为0时，递归结束。
//需要定义一个非模板函数，这个函数没有参数，表示参数包里已经没有参数了（可以理解为print模板函数的没有参数的具体化版本）

int main(void)
{
    //print("金莲", 4, "西施");   // 这多个参数叫参数包，把多个参数打成一个包的意思
    //print("冰冰", 8, "西施", 3);
    func("我是绝世帅歌。", "冰冰", 8, "西施", 3);  // "我是绝世帅歌。"不是可变参数，其它的都是。
}
```

24. 时间操作chrono库

C++11提供了`chrono`模板库，实现了一系列时间相关的操作（**时间长度，系统时间和计时器**）

头文件：`#include <chrono>`

命名空间：`std::chrono`

1. 时间长度

`duration`模板类用于表示一段时间（**时间长度，时钟周期**），如：1小时，8分钟，5秒

`duration`的定义如下：

```
template<class Rep, class Period = std::ratio<1, 1>>
class duration
{
   ……
};
```

为了方便使用，定义了一些常用的时间长度，比如：时，分，秒，毫秒，微秒，纳秒，它们都位于`std::chrono`命名空间下，定义如下：

```
using hours         = duration<Rep, std::ratio<3600>>   // 小时
using minutes           = duration<Rep, std::ratio<60>>     // 分钟
using seconds           = duration<Rep>                     // 秒
using milliseconds      = duration<Rep, std::milli>         // 毫秒
using microseconds      = duration<Rep, std::micro>         // 微秒
using nanoseconds   = duration<Rep, std::nano>              // 纳秒
```

```
#include <iostream>
#include <chrono>
using namespace std;

int main(void)
{
    //chrono::hours t1(1);  // 1小时
    //chrono::minutes t2(60);   // 60分钟
    //chrono::seconds t3(60 * 60);  // 60*60秒
    //chrono::microseconds t4(60 * 60 * 1000);  // 60*60*1000毫秒
    //// duration时间长度类重载了各种算术运算符，+, -, *, /, 还支持时间的运算，1小时=60分钟
    //chrono::microseconds t5(60 * 60 * 1000 * 1000);   // 微秒，警告：整数溢出
```

```
    //chrono::nanoseconds t6(60 * 60 * 1000 * 1000 * 1000); // 纳秒，警告：整数溢出
    //
    //if (t1 == t2) cout << "t1==t2" << endl;
    //if (t1 == t3) cout << "t1==t3" << endl;
    //if (t1 == t4) cout << "t1==t4" << endl;

    //// 获取时钟周期的值，返回的是int整数
    //cout << "t1=" << t1.count() << endl;
    //cout << "t2=" << t2.count() << endl;
    //cout << "t3=" << t3.count() << endl;
    //cout << "t4=" << t4.count() << endl;

    chrono::seconds t7(1);  // 1秒
    chrono::milliseconds t8(1000);  // 1000毫秒
    chrono::microseconds t9(1000 * 1000);   // 1000*1000微秒
    chrono::nanoseconds t10(1000 * 1000 * 1000);    //1000*1000*1000纳秒

    if (t7 == t8) cout << "t7==t8" << endl;
    if (t7 == t9) cout << "t7==t9" << endl;
    if (t7 == t10) cout << "t7==t10" << endl;

    // 获取时钟周期的值
    cout << "t7=" << t7.count() << endl;
    cout << "t8=" << t8.count() << endl;
    cout << "t9=" << t9.count() << endl;
    cout << "t10=" << t10.count() << endl;
}
```

2. 系统时间

system_clock 类支持了对系统时钟的访问，提供了三个**静态成员函数**。

- `static std::chrono::time_point<std::chrono::system_clock> now() noexcept;`

> 返回当前时间的时间点。

- **static std::time_t to_time_t(const time_point& t) noexcept;**

> 将时间点time_point类型转换为std::time_t 类型。

- **static std::chrono::system_clock::time_point from_time_t(std::time_t t) noexcept;**

> 将std::time_t类型转换为时间点time_point类型。

```
#define _CRT_SECURE_NO_WARNINGS  // localtime()需要这个宏。
#include <iostream>
#include <chrono>
#include <iomanip>   // put_time()函数需要包含的头文件。
#include <sstream>
using namespace std;

int main(void)
{
    //(1) 静态成员函数chrono::system_clock::now()用于获取系统时间（C++时间）
    //chrono::time_point<chrono::system_clock> now = chrono::system_clock::now();
    auto now = chrono::system_clock::now();

    // (2) 静态成员函数chrono::system_clock::to_time_t()把时间转换为tiem_t。(UTC时间)
    //time_t t_now = chrono::system_clock::to_time_t(now);
    auto t_now = chrono::system_clock::to_time_t(now);

    //TODO: 时间的偏移(加上对应的秒)
    t_now = t_now + 24 * 60 * 60;   // 把当前的时间加1天
    //t_now = t_now + (-1 * 60 * 60);   // 把当前时间减1秒
    //t_now = t_now + 120;  // 当前时间减120秒
```

```
    // (3) std::localtime()函数把time_t转换为本地时间 (北京时间)
    // localtime()不是线程安全的, VS用localtime_s代替, Linux用localtime_r代替
    //tm* tm_now = std::localtime(&t_now);
    auto tm_now = std::localtime(&t_now);

    // (4) 格式化输出本地时间
    std::cout << std::put_time(tm_now, "%Y-%m-%d %H:%M:%S")<<std::endl;
    // Y-年 m-月 d-日 H-小时 M-分钟 S-秒
    std::cout << std::put_time(tm_now, "%Y-%m-%d") << std::endl;
    std::cout << std::put_time(tm_now, "%H:%M:%S") << std::endl;
    std::cout << std::put_time(tm_now, "%Y%m%d%H%M%S") << std::endl;

    // 实际开发中, 获取系统时间后, 不一定要显示出来。
    stringstream ss; //  创建stringstream对象, 需要包含头文件 <sstream>
    ss << std::put_time(tm_now, "%Y-%m-%d");    // 把时间输出到对象ss中
    string timestr = ss.str();  // 把ss转换成string对象
    cout << timestr << endl;
}
```

3. 计时器

steady_clock类相当于秒表，操作系统只要启动就会进行时间的累加，常用于耗时的统计（精确到纳秒）

（我们用计时器把程序执行任务时消耗的时间记录下来，就可以衡量程序的性能）

```
#include <iostream>
#include <chrono>
using namespace std;

int main(void)
{
    // 静态成员函数 chrono::steady_clock::now()获取开始的时间点
    //chrono::steady_clock::time_point start = chrono::steady_clock::now();
    auto start = chrono::steady_clock::now();

    // 执行一些代码, 让它消耗一些时间
    cout << "计时开始....." << endl;
    for (int ii = 0; ii < 1000000; ii++) {
        //cout << "我是一只傻傻鸟" << endl;
    }
    cout << "计时结束...." << endl;

    // 静态成员函数 chrono::steady_clock::now()获取结束的时间点
    //chrono::steady_clock::time_point end = chrono::steady_clock::now();
    auto end = chrono::steady_clock::now();

    // 计算消耗的时间
    auto dt = end - start;
    cout << "耗时: " << dt.count() << "纳秒 (" << (double)dt.count() / (1000 * 1000 * 1000) <<
"秒) " << endl;
}
```