



2022K+ 全球软件研发行业创新峰会

技术创新
无边界

剖玄析微 - C++新标准细节简析

主讲人：谢丙堃



天下难事，必作于易；天下大事，必作于细。

—— 老子《道德经》第六十三章

目录

CONTENTS

- 1 参数传递
- 2 返回对象
- 3 聚合类型扩展





1

参数传递



契机



- 传值还是传引用，似乎早有定论？
- 为何是现在讨论这个问题？
- 深入到代码细节，我们还能体会到哪些优化方法？



关于string_view



- C++17标准引入到标准库；
- 最早草案是2012年的string_ref ([n3442](#)) ；
- 直到2014年，经历了7个版本的修订；
- 典型的实现只包含两个成员：指向常量字符串指针和大小。



复杂对象的情况 (-O2)



```
size_t ret_str_byref(const std::string& s) { return s.size(); }
```

```
size_t ret_str_byval(std::string s) { return s.size(); }
```



```
ret_str_byref:  
    mov eax, DWORD PTR [rdi+8]  
    ret
```

```
ret_str_byval:  
    mov eax, DWORD PTR [rdi+8]  
    ret
```



复杂对象的情况 (-O2)



```
size_t foo1() {  
    std::string s(gen());  
    return ret_str_byval(s);  
}  
size_t foo2() {  
    std::string s(gen());  
    return ret_str_byref(s);  
}
```

生成64行汇编代码

生成14行汇编代码



对比string_view (-O2)



```
size_t ret_str_byref(const std::string& s) { return s.size(); }
```

```
size_t ret_str_byval(std::string s) { return s.size(); }
```

```
size_t ret_sv_byval(std::string_view sv) { return sv.size(); }
```

```
ret_str_byref:
    mov eax, DWORD PTR [rdi+8]
    ret
ret_str_byval:
    mov eax, DWORD PTR [rdi+8]
    ret
ret_sv_byval:
    mov eax, edi
    ret
```



对比string_view (-O2)



```
size_t foo3() {  
    std::string s(gen());  
    return ret_sv_byval(s);  
}
```

foo3(): 生成14行汇编代码

```
push rbx  
sub rsp, 32  
mov rdi, rsp  
call gen  
mov rdi, qword ptr [rsp]  
mov rbx, qword ptr [rsp + 8]  
lea rax, [rsp + 16]  
cmp rdi, rax  
je .LBB6_2  
call operator delete  
.LBB6_2:  
mov eax, ebx  
add rsp, 32  
pop rbx  
ret
```



对比string_view (-O2)



```
size_t sv_call_val(std::string_view sv) {return ret_sv_byval(sv);}
```

```
size_t sv_call_ref(std::string_view sv) {return ret_sv_byref(sv);}
```

sv_call_val
 jmp ret_sv_byval

sv_call_ref
 sub rsp, 24
 mov qword ptr [rsp + 8], rdi
 mov qword ptr [rsp + 16], rsi
 lea rdi, [rsp + 8]
 call ret_sv_byref
 add rsp, 24
 ret



对比string_view (-O2)



```
size_t ret_sv_byval(std::string_view sv, size_t& troublemaker) {  
    size_t temp = troublemaker;  
    troublemaker++;  
    size_t retval = sv.size();  
    troublemaker = temp;  
    return retval;  
}  
  
size_t ret_sv_byref(const std::string_view& sv, size_t& troublemaker) {  
    size_t temp = troublemaker;  
    troublemaker++;  
    size_t retval = sv.size();  
    troublemaker = temp;  
    return retval;  
}
```



对比string_view (-O2)



```
ret_sv_byval  
    mov rax, rdi  
    ret
```

```
ret_sv_byref  
    mov rcx, qword ptr [rsi]  
    lea rax, [rcx + 1]  
    mov qword ptr [rsi], rax  
    mov rax, qword ptr [rdi]  
    mov qword ptr [rsi], rcx  
    ret
```

2

返回对象



拷贝消除 (copy elision)



```
#include <iostream>
class X {
public:
    X() { std::cout << "X ctor" << std::endl; }
    X(const X& x) { std::cout << "X copy ctor" << std::endl; }
    ~X() { std::cout << "X dtor" << std::endl; }
};
X make_x() {
    X x1;
    return x1;
}
int main() {
    X x2 = make_x();
}
```



拷贝消除 (copy elision)



| 拷贝消除 | C++14 关闭拷贝消除 | C++17 关闭拷贝消除 |
|------------------|--|---|
| X ctor X dtor | X ctor X copy ctor X dtor X copy ctor X dtor X dtor | X ctor X copy ctor X dtor X dtor |



优化点在哪



```
make_x(): # @make_x()  
    push rbx  
    mov rbx, rdi  
    call X::X() [base object constructor]  
    mov rax, rbx  
    pop rbx  
    ret
```

```
main: # @main  
    push rbx  
    sub rsp, 16  
    lea rdi, [rsp + 8]  
    call make_x()
```

拷贝消除：
直接在x2上进行构造



优化点在哪



C++14禁用拷贝消除：
产生临时对象，发生
两次拷贝

make_x(): # @make_x()

push r14

push rbx

push rax

mov rbx, rdi

mov r14, rsp

mov rdi, r14

call X::X() [base object constructor]

mov rdi, rbx

mov rsi, r14

call X::X(X const&) [base object constructor]

mov rdi, rsp

call X::~~X() [base object destructor]

mov rax, rbx

add rsp, 8

pop rbx

pop r14

ret

main: # @main

push rbx

sub rsp, 16

mov rbx, rsp

mov rdi, rbx

call make_x()

lea rdi, [rsp + 8]

mov rsi, rbx

call X::X(X const&) [base object constructor]

mov rdi, rsp

call X::~~X() [base object destructor]

lea rdi, [rsp + 8]

call X::~~X() [base object destructor]

xor eax, eax

add rsp, 16

pop rbx

ret



优化点在哪



C++17禁用拷贝消除：
没有产生临时对象，
直接拷贝到目标对象
上

make_x(): # @make_x()

```
push r14
push rbx
push rax
mov rbx, rdi
mov r14, rsp
mov rdi, r14
call X::X() [base object constructor]
mov rdi, rbx
mov rsi, r14
call X::X(X const&) [base object constructor]
mov rdi, rsp
call X::~~X() [base object destructor]
mov rax, rbx
add rsp, 8
pop rbx
pop r14
ret
```

main: # @main

```
push rbx
sub rsp, 16
lea rbx, [rsp + 8]
mov rdi, rbx
call make_x()
mov rdi, rbx
call X::~~X() [base object destructor]
xor eax, eax
add rsp, 16
pop rbx
ret
```



减少生成临时对象



6.7.7 Temporary objects [class.temporary]

The materialization of a temporary object is generally delayed as long as possible in order to avoid creating unnecessary temporary objects.



拷贝消除 (copy elision)



<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0135r1.html>

```
class Thing {  
public:  
    Thing();  
    ~Thing();  
    Thing(const Thing&);  
};
```

```
Thing f() {  
    Thing t;  
    return t;  
}
```

```
Thing t2 = f();
```

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class `Thing`: the copying of the local automatic object `t` into the temporary **result** object for the return value of function call `f()` and the copying of that temporary object into object `t2` **which is the global object `t2`**. Effectively, the construction of the local object `t` can be viewed as directly initializing the global object `t2`, and that object's destruction will occur at program exit. Adding a move constructor to `Thing` has the same effect, but it is the move construction from the temporary **local automatic** object to `t2` that is elided.



拷贝消除带来好处



```
#include <iostream>
class X {
public:
    X() { std::cout << "X ctor" << std::endl; }
    ~X() { std::cout << "X dtor" << std::endl; }
private:
    X(const X& x) = delete;
};

X make_x() {
    return X();
}

int main() {
    X x2 = make_x();
}
```

C++17标准可以编译通过，在此之前不能编译。

创建X对象的过程比较复杂的情况下，使用创建对象的函数来完成对象的创建，在拷贝消除之后可以支持对象本身无法拷贝的情况。



拷贝消除优化并非万能



```
#include <iostream>
#include <ctime>

class X {
public:
    X() { std::cout << "X ctor" << std::endl; }
    X(const X& x) { std::cout << "X copy ctor" << std::endl; }
    ~X() { std::cout << "X dtor" << std::endl; }
};

X make_x() {
    X x1, x2;
    if (std::time(nullptr) % 50 == 0) {
        return x1;
    }
    else {
        return x2;
    }
}

int main() {
    X x3 = make_x();
}
```



除了返回对象以外的拷贝消除



- `return`语句中返回类类型，返回对象类型和函数返回类型相同，并且要求类型是非易失且有自动存储周期的对象。
- `throw`表达式，操作数类型也要求是非易失且有自动存储周期的对象，并且作用域不超过最内侧的`try`。
- 异常处理（其实就是`try-catch`中`catch({})`），声明的对象如果和抛出对象类型相同，可以将声明对象看作抛出对象的别名，前提条件是这个对象在这个过程中除了构造和析构是不会被改变的。
- 在协程中，协程参数的拷贝可以被忽略，也就是直接引用参数本身，当然也有前提条件，就是在处理对象的过程中除了构造和析构是不会被改变的。

3

聚合类型扩展



聚合类型的新定义



- 聚合类型需要满足常规条件，包括：
 - 没有用户提供的构造函数；
 - 没有私有和受保护的静态数据成员；
 - 没有虚函数。
- 在新的标准中，如果类存在继承关系，额外满足条件：
 - 必须是公开的基类，不能是私有或者受保护的基类；
 - 必须是非虚继承。

也是一个
聚合类型



甄别聚合类型



```
#include <iostream>
#include <string>
```

```
class MyString : public std::string {};
```

```
int main()
{
    std::cout << "std::is_aggregate_v<std::string> = "
               << std::is_aggregate_v<std::string> << std::endl;
    std::cout << "std::is_aggregate_v<MyString> = "
               << std::is_aggregate_v<MyString> << std::endl;
}
```

std::is_aggregate_v<std::string> = 0

std::is_aggregate_v<MyString> = 1



聚合类型的初始化



```
#include <iostream>
#include <string>
class MyStringWithIndex : public std::string {
public:
    int index_ = 0;
};
std::ostream& operator << (std::ostream &o, const MyStringWithIndex& s) {
    o << s.index_ << ":" << s.c_str();
    return o;
}
int main() {
    MyStringWithIndex s{"hello world", 11};
    std::cout << s << std::endl;
}
```



兼容性问题



```
#include <iostream>
class BaseData {
    int data_;
public:
    int Get() { return data_; }
protected:
    BaseData() : data_(11) {}
};

class DerivedData : public BaseData {};

int main() {
    DerivedData d{};
    std::cout << d.Get() << std::endl;
}
```

C++11 C++14 中为非聚合类型，调用构造函数构造对象。

C++17 C++20 中为聚合类型，使用聚合类型初始化直接初始化对象。



兼容性问题



```
#include <iostream>
class BaseData {
    int data_;
public:
    int Get() { return data_; }
protected:
    BaseData() : data_(11) {}
};

class DerivedData : public BaseData {};

int main() {
    DerivedData d{};
    std::cout << d.Get() << std::endl;
}
```

error: base class 'BaseData' has protected default constructor



问题并没有结束



```
#include <iostream>
struct X {
    X() = default;
};
```

```
std::is_aggregate_v<X> : true
std::is_aggregate_v<Y> : true
```

```
struct Y {
    Y() = delete;
};
```

```
int main() {
    std::cout << std::boolalpha
        << "std::is_aggregate_v<X> : " << std::is_aggregate_v<X> << std::endl
        << "std::is_aggregate_v<Y> : " << std::is_aggregate_v<Y> << std::endl;
}
```



问题并没有结束



```
Y y1;  
Y y2{};
```

error: call to deleted
constructor of 'Y'

```
struct Y {  
private:  
    Y() = default;  
};
```

error: calling a private
constructor of class 'Y'



C++20 禁止聚合类型使用构造函数声明



- 聚合类型需要满足常规条件，包括：
 - ~~没有用户提供的构造函数；~~没有构造函数声明；
 - 没有私有和受保护的静态数据成员；
 - 没有虚函数。



THANKS



2022K+
全球软件研发行业创新峰会