Lambda? Lambda!

吴咏炜
博览首席技术咨询师

内容链接

https://github.com/adah1972/cpp_summit_2024

自我介绍

- 学编程 40 年，超 30 年 C++ 老兵
- 热爱 C++ 和开源技术，偏好精炼、跨平台的代码
- 参加每届 C++ 及系统软件技术大会
- 目前主要从事 C++ 方向的咨询和培训工作
- 参与译作：《编程大师访谈录》《C++ Core Guidelines 解析》《在纷繁多变的世界里茁壮成长：C++ 2006–2020》
- 原创作品（即将出版）：《C++ 实战：核心技术与最佳实践》

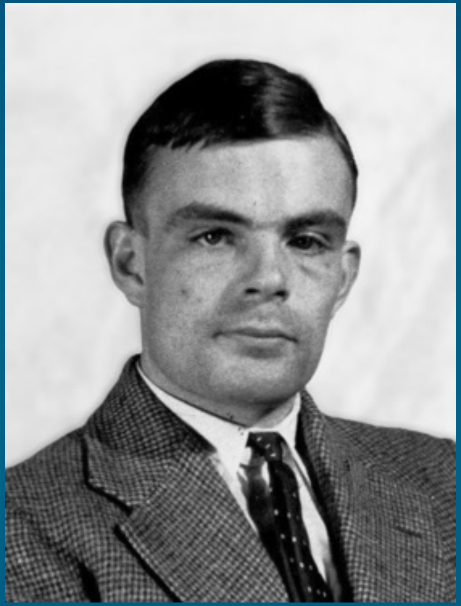
- lambda 演算
- lambda：从无到 C++17
- lambda 的使用场景
- C++20/23 的 lambda 改进

PART 1

lambda 演算

邱奇–图灵命题

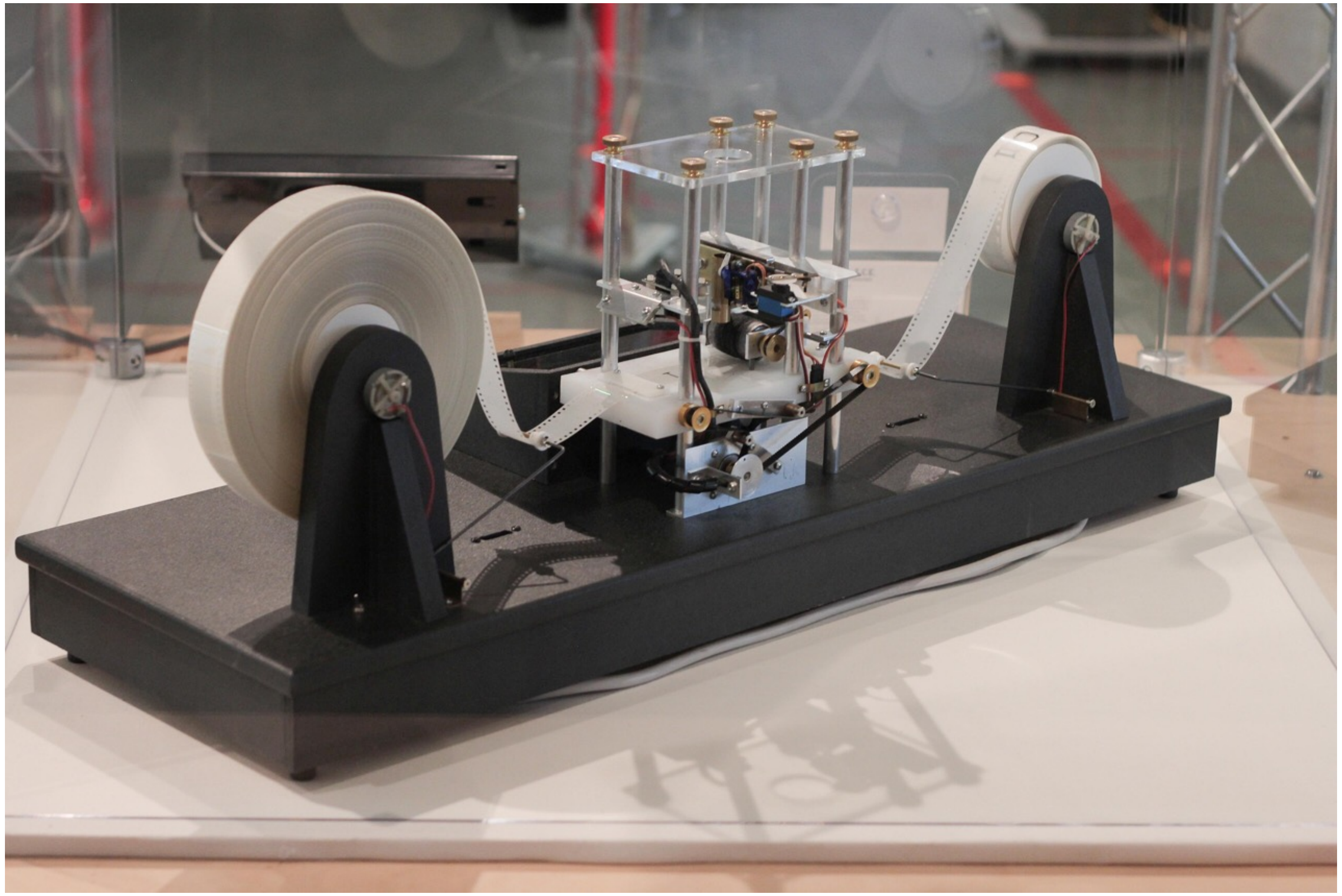
1936

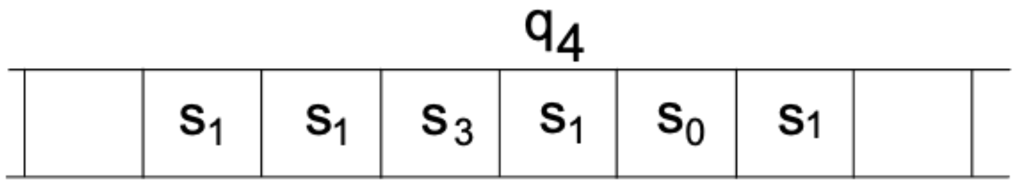


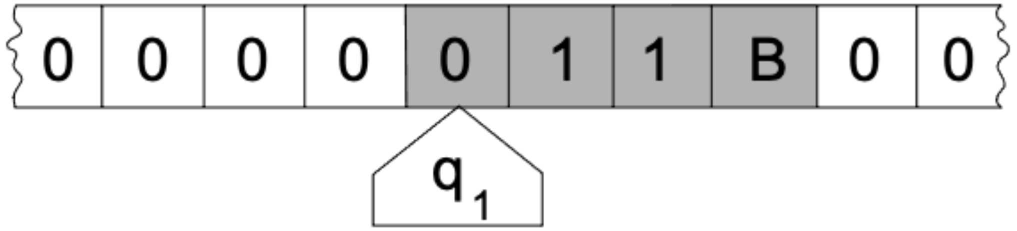
An Unsolvable Problem of Elementary Number Theory

On Computable Numbers, with an Application to the Entscheidungsproblem

图灵机







来源：https://en.wikipedia.org/wiki/Turing_machine

lambda 演算

lambda 项（比编程里的 lambda 表达式更宽泛）

- 变量 x 是 lambda 项
- 如 x 是变量，M 是 lambda 项，则 $(\lambda x. M)$ 是 lambda 项
- 如 M、N 是 lambda 项，则 $(M\ N)$ 是 lambda 项

归约

- α 变换：$(\lambda x. x * x) \rightarrow (\lambda y. y * y)$ ← 等价形式
- β 归约：$((\lambda x. x * x)\ 3) \rightarrow (3 * 3)$ ← 应用
- η 归约：$f \rightarrow (\lambda x. (f\ x))$ ← 延迟求值

lambda 演算……

$$\mathbf{I} \equiv \lambda x.x$$
$$\mathbf{K} \equiv \lambda xy.x$$
$$\mathbf{K_*} \equiv \lambda xy.y$$
$$\mathbf{Y} \equiv \lambda f.\big(\lambda x.f(xx)\big)\big(\lambda x.f(xx)\big)$$
$$\textbf{if\_then\_else} \equiv \lambda x.x$$
$$\textbf{true} \equiv K$$
$$\textbf{false} \equiv K_*$$
$$\mathbf{0} \equiv \lambda fx.x$$
$$\mathbf{1} \equiv \lambda fx.fx$$
$$\mathbf{2} \equiv \lambda fx.f(fx)$$
$$\mathbf{3} \equiv \lambda fx.f\big(f(fx)\big)$$

…

lambda 表达式和编程语言

- 数学函数定义： $\mathrm{sqr}(x) = x * x$
- 使用 lambda： $\mathrm{sqr} = \lambda x.x * x$
- Haskell 代码： `sqr = \ x -> x * x`
- Python 代码： `sqr = lambda x: x * x`
- C++ 代码： `auto sqr = [](auto x) { return x * x; };`

PART 2

lambda：从无到 C++17

STepanov & Lee

STL 算法

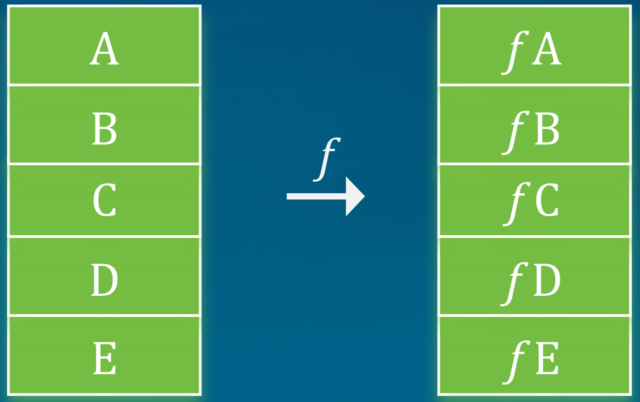
- fill
- iota
- copy
- remove
- find
- count
- unique
- …

- transform
- for_each
- copy_if
- remove_if
- find_if
- count_if
- sort
- …

函数式编程之映射操作

Map

- map f [A, B, C, D, …] = [f A, f B, f C, f D, …]



C++ 对应物

- `Container c{A, B, C, D, …};`
- `Container d(c.size());`
- `transform(c.begin(), c.end(), d.begin(), f);`

非 transform 方式

```
auto it1 = c.begin();
auto it2 = d.begin();
for (; it1 != c.end(); ++it1, ++it2) {
    *it2 = *it1 * *it1;
}
```

transform：使用函数指针

```
int sqr(int x)
{
    return x * x;
}


…


transform(c.begin(), c.end(), d.begin(), sqr);
```

transform：使用（非指针）函数对象

```
struct sqr {
    int operator()(int x) const
    {
        return x * x;
    }
};
…
transform(c.begin(), c.end(), d.begin(), sqr());
```

transform：使用带数据的函数对象

```
struct plus_n {
    plus_n(int n) : n_(n) {}
    int operator()(int x) const { return x + n_; }
private:
    int n_;
};
…
transform(c.begin(), c.end(), d.begin(), plus_n(2));
```

transform：使用 plus 和 bind2nd

```
transform(c.begin(), c.end(), d.begin(),
          bind2nd(plus<int>(), 2));
```

bind2nd函数模板的返回结果是一个函数对象

transform：使用 Boost.Lambda

```
transform(c.begin(), c.end(), d.begin(),
          _1 + 2));
```

利用运算符重载生成后续执行加法的函数对象

transform：使用 Boost.Lambda

```
transform(c.begin(), c.end(), d.begin(),
          (cout << _1 << '\n', _1 + 2));
```

利用了逗号运算符的副作用

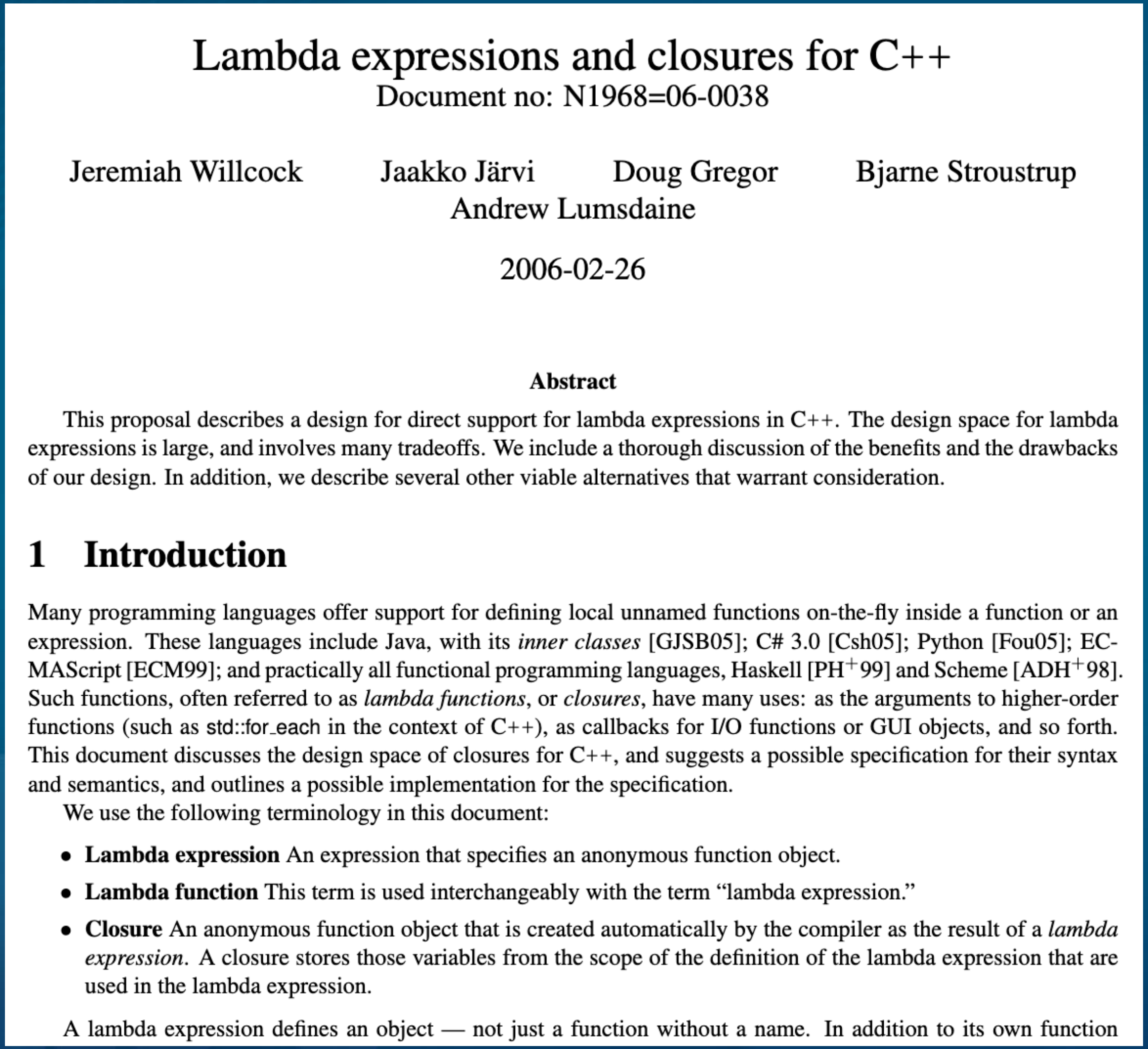
transform：使用 Boost.Lambda

```cpp
transform(c.begin(), c.end(), d.begin(),
          (cout << _1 << endl, _1 + 2));
```

```
test.cpp: In function 'int main()':
test.cpp:29:58: error: no match for 'operator<<' (operand types are 'const
boost::lambda::lambda_functor<boost::lambda::lambda_functor_base<boost::lambda::bitwise_action<boost::lambda::leftshift_action>, boost::tuples::tuple<std::basic_ostream<char>&,
boost::lambda::lambda_functor<boost::lambda::placeholder<1> >, boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type,
boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type> > >' and '<unresolved overloaded function type>')
   29 |     transform(c.begin(), c.end(), d.begin(), (cout << _1 << endl, _1 + 2));
      |                                               ~~~~~~~~~~~~^~~~~~
In file included from /usr/local/include/c++/10.2.0/string:55,
                 from /usr/local/include/c++/10.2.0/bits/locale_classes.h:40,
                 from /usr/local/include/c++/10.2.0/bits/ios_base.h:41,
                 from /usr/local/include/c++/10.2.0/ios:42,
                 from /usr/local/include/c++/10.2.0/ostream:38,
                 from /usr/local/include/c++/10.2.0/iostream:39,
                 from test/test.cpp:3:
/usr/local/include/c++/10.2.0/bits/basic_string.h:6458:5: note: candidate: 'template<class _CharT, class _Traits, class _Alloc> std::basic_ostream<_CharT, _Traits>&
std::operator<<(std::basic_ostream<_CharT, _Traits>&, const std::__cxx11::basic_string<_CharT, _Traits, _Alloc>&)'
 6458 |     operator<<(basic_ostream<_CharT, _Traits>& __os,
      |     ^~~~~~~~
/usr/local/include/c++/10.2.0/bits/basic_string.h:6458:5: note:   template argument deduction/substitution failed:
test/test.cpp:29:61: note:   types 'std::basic_ostream<_CharT, _Traits>' and 'const
boost::lambda::lambda_functor<boost::lambda::lambda_functor_base<boost::lambda::bitwise_action<boost::lambda::leftshift_action>, boost::tuples::tuple<std::basic_ostream<char>&,
boost::lambda::lambda_functor<boost::lambda::placeholder<1> >, boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type,
boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type> > >' have incompatible cv-qualifiers
   29 |     transform(c.begin(), c.end(), d.begin(), (cout << _1 << endl, _1 + 2));
      |                                                                ^~~~
In file included from /usr/local/include/c++/10.2.0/bits/ios_base.h:46
```

lambda 表达式的标准化

CPP-Summit 2024
全球C++及系统软件技术大会



Lambda expressions and closures for C++
Document no: N1968=06-0038

Jeremiah Willcock Jaakko Järvi Doug Gregor Bjarne Stroustrup
Andrew Lumsdaine

2006-02-26

Abstract

This proposal describes a design for direct support for lambda expressions in C++. The design space for lambda expressions is large, and involves many tradeoffs. We include a thorough discussion of the benefits and the drawbacks of our design. In addition, we describe several other viable alternatives that warrant consideration.

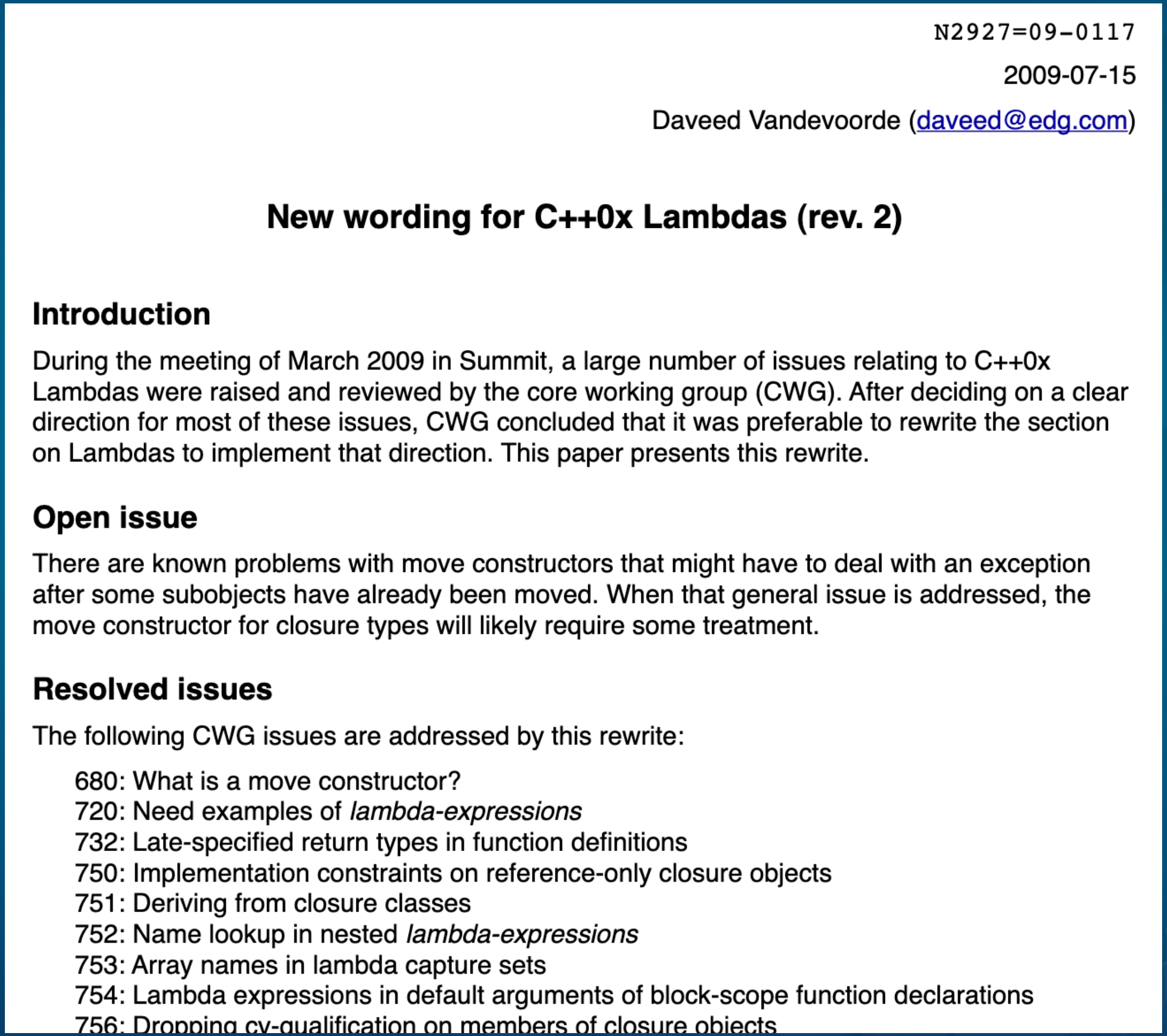
1 Introduction

Many programming languages offer support for defining local unnamed functions on-the-fly inside a function or an expression. These languages include Java, with its *inner classes* [GJSB05]; C# 3.0 [Csh05]; Python [Fou05]; ECMAScript [ECM99]; and practically all functional programming languages, Haskell [PH$^+$99] and Scheme [ADH$^+$98]. Such functions, often referred to as *lambda functions*, or *closures*, have many uses: as the arguments to higher-order functions (such as std::for_each in the context of C++), as callbacks for I/O functions or GUI objects, and so forth. This document discusses the design space of closures for C++, and suggests a possible specification for their syntax and semantics, and outlines a possible implementation for the specification.

We use the following terminology in this document:

- **Lambda expression** An expression that specifies an anonymous function object.
- **Lambda function** This term is used interchangeably with the term "lambda expression."
- **Closure** An anonymous function object that is created automatically by the compiler as the result of a *lambda expression*. A closure stores those variables from the scope of the definition of the lambda expression that are used in the lambda expression.

A lambda expression defines an object — not just a function without a name. In addition to its own function



N2927=09-0117
2009-07-15
Daveed Vandevoorde (daveed@edg.com)

New wording for C++0x Lambdas (rev. 2)

Introduction

During the meeting of March 2009 in Summit, a large number of issues relating to C++0x Lambdas were raised and reviewed by the core working group (CWG). After deciding on a clear direction for most of these issues, CWG concluded that it was preferable to rewrite the section on Lambdas to implement that direction. This paper presents this rewrite.

Open issue

There are known problems with move constructors that might have to deal with an exception after some subobjects have already been moved. When that general issue is addressed, the move constructor for closure types will likely require some treatment.

Resolved issues

The following CWG issues are addressed by this rewrite:

680: What is a move constructor?
720: Need examples of *lambda-expressions*
732: Late-specified return types in function definitions
750: Implementation constraints on reference-only closure objects
751: Deriving from closure classes
752: Name lookup in nested *lambda-expressions*
753: Array names in lambda capture sets
754: Lambda expressions in default arguments of block-scope function declarations
756: Dropping cv-qualification on members of closure objects

transform：使用 lambda 表达式

```
transform(c.begin(), c.end(), d.begin(),
          [](int x) {
              cout << x << endl;
              return x + 2;
          });
```

编译器为 lambda 表达式生成的代码示意

```
struct FuncObj_ADEC24A {
    auto operator()(int x) const
    {
        cout << x << endl;
        return x + 2;
    }
};
…
transform(c.begin(), c.end(), d.begin(), FuncObj_ADEC24A{});
```

lambda 表达式的按值捕获

```
transform(c.begin(), c.end(), d.begin(),
          [n](int x) {
              return x + n;
          });
```

lambda 表达式的按引用捕获

```
transform(c.begin(), c.end(), d.begin(),
          [&n](int x) {
              return x + n;
          });
```

带捕获的 lambda 表达式生成的代码示意

```
struct FuncObj_697A903 {
    FuncObj_697A903(const int& n) : n(n) {}
    auto operator()(int x) const { return x + n; }

private:
    int n;
};
…
transform(c.begin(), c.end(), d.begin(), FuncObj_697A903{n});
```

带捕获的 lambda 表达式生成的代码示意

```
struct FuncObj_8F0A216 {
    FuncObj_8F0A216(int& n) : n(n) {}
    auto operator()(int x) const { return x + n; }

private:
    int& n;
};
…
transform(c.begin(), c.end(), d.begin(), FuncObj_8F0A216{n});
```

lambda 表达式的捕获

- 用“=”按值捕获所有用到的局部变量。**不推荐使用。**
- 用“&”按引用捕获所有用到的局部变量。只应该用于在当前函数里**局部使用**的 lambda 表达式，以免产生悬空引用。
- 用“this”在成员函数里捕获当前对象的 this 指针，以便在 lambda 表达式里访问当前对象的成员。这本质上是一种按引用捕获。
- 用“*this”在成员函数里捕获当前对象的一个复本，以便在 lambda 表达式里访问这个对象复本的成员（自 C++17 起）。
- 用“变量名 = 表达式”捕获表达式的值到指定的变量里（自 C++14 起）。
- 用“&变量名 = 表达式”捕获表达式的引用到指定的变量里（自 C++14 起）。

成员函数模板和泛型 lambda

```
struct Print {
  template <typename T>
  void operator()(const T& x) const
  {
    cout << x << ' ';
  }
};

for_each(d.begin(), d.end(),
         Print{});
```

```
for_each(d.begin(), d.end(),
         [](const auto& x) {
           cout << x << ' ';
         });
```

PART 3

lambda 的使用场景

函数对象的性能优势

	-O0	-O2 -fno-inline	-O2	-O2 对 -O0 的提升
sort 使用函数对象	340981(1717)	197830(675)	24414(333)	14.0x
sort 使用函数指针	334601(1843)	209241(609)	46384(220)	7.21x
qsort	133801(1814)	87014(790)	85323(535)	1.57x

使用函数对象（含 lambda 表达式）传参给函数模板容易进行内联

配合标准算法的 lambda 使用示例

```cpp
sort(c.begin(), c.end(), [](int x, int y) { return abs(x) > abs(y); });

copy_if(c.begin(), c.end(), back_inserter(d), [](int n) { return n % 2 == 0; });

auto it = remove_if(c.begin(), c.end(), [](int x) { return x % 2 != 0; });

cout << any_of(c.begin(), c.end(), [](int x) { return x % 2 == 0; });

generate(v.begin(), v.end(), [&] { return dist(engine); });
```

带状态变化的 lambda 表达式

```
transform(c.begin(), c.end(), d.begin(),
          [flag = true](int x) mutable {
              flag = !flag;
              return x + (flag ? 1 : 0);
          });
```

编译器生成的函数对象的 operator() 不再是 const

C 风格的回调

```
typedef void (*Callback)(void* context, int param);
void registerCallback(Callback callback, void* context);

void myCallback(void* context, int param)
{
    auto ptr = static_cast<Obj*>(context);
    ptr->Op(param);
}

registerCallback(myCallback, &obj);
```

通常需要保证 obj 在回调可能发生时一直存在

使用 function 和 lambda 的回调

```
void registerCallback(function<void(int)> func);
```

同前，需保证 obj 在回调可能发生时一直存在

```
registerCallback([&obj](int param) { obj.Op(param); });

registerCallback([obj](int param) mutable { obj.Op(param); });
```

新的可能，回调在，对象就存在

立即求值的 lambda 表达式

```
Obj obj;
switch (init_mode) {
case init_mode1:
  obj = Obj(…);
  break;
case init_mode2;
  obj = Obj(…);
  break;
…
}
```

```
auto obj = [&]() {
  switch (init_mode) {
  case init_mode1:
    return Obj(…);
  case init_mode2:
    return Obj(…);
  …
  }
}();
```

性能只高不低；防止漏初始化

lambda 表达式作为代码组织方式

```
for (auto& x : c) {
    cout << x << ' ';
}
```

对于图示的循环基于范围的 for 循环最优

```
for_each(c.begin(), c.end(),
         [&](auto& x) {
             cout << x << ' ';
         });
```

比简单条件和循环更复杂的情况下 lambda 表达式具有更大的灵活性

组合示例：错误码检查

```
error_t result{};
result = check1(…);
if (result != SUCCESS) {
  return result;
}
result = check2(…);
if (result != SUCCESS) {
  return result;
}
result = check3(…);
if (result != SUCCESS) {
  return result;
}
result = check4(…);
if (result != SUCCESS) {
  return result;
}
return SUCCESS;
```

```
return checked_exec(
  SUCCESS,
  [&] { return check1(…); },
  [&] { return check2(…); },
  [&] { return check3(…); },
  [&] { return check4(…); });
```

https://godbolt.org/z/KK9sqq8sj

使用折叠表达式实现 checked_exec

```
template <typename R, typename... Fn>
R checked_exec(const R& expected, Fn&&... fn)
{
    R result = expected;
    (void)(((result = std::forward<Fn>(fn)()) == expected)
           && ...);
    return result;
}
```

使用 lambda 表达式访问 variant

```
variant<string, int, char> obj{"Hello World"};

visit([](const auto& v) { cout << v << '\n'; }, obj);

visit(overloaded{
          [](const string& val) { cout << "s: " << val << '\n'; },
          [](int val) { cout << "i: " << val << '\n'; },
          [](char val) { cout << "c: " << val << '\n'; },
      }, obj);
```

expected 的链式调用

```cpp
expected<int, error_code> addDivideSafe(int i, int j, int k)
{
    return divideSafe(j, k).and_then(
        [&](int q) -> expected<int, error_code> {
            if ((i > 0 && q > INT_MAX - i) ||
                (i < 0 && q < INT_MIN - i)) {
                return unexpected(
                    make_error_code(errc::value_too_large));
            }
            return i + q;
        });
}
```

异步网络代码（使用 Asio）

```
class Session : public enable_shared_from_this<Session> {
public:
  Session(tcp::socket socket) : socket_(std::move(socket))
  {}
  void start() { doRead(); }

private:
  void doRead()
  {
    auto self = shared_from_this();
    socket_.async_read_some(
      buffer(data_),
      [this, self](error_code ec, size_t length) {
        if (!ec) doWrite(length);
        else if (ec != asio::error::eof)
          cerr << "Error: " << ec.message() << "\n";
      });
  }

  void doWrite(size_t length)
  {
    auto self = shared_from_this();
    asio::async_write(
      socket_, buffer(data_, length),
      [this, self](error_code ec, size_t /*length*/) {
        if (!ec) doRead();
        else
          cerr << "Error: " << ec.message() << "\n";
      });
  }

  tcp::socket socket_;
  char data_[1024];
};
```

constexpr 参数?

```
constexpr auto make_char_array(string_view str)
{
    constexpr size_t len = str.size();
    array<char, len> result{};
    for (size_t i = 0; i < len; ++i) {
        result[i] = str[i];
    }
    return result;
}
```

constexpr 参数（使用 lambda）

CPP-Summit 2024
全球C++及系统软件技术大会

```cpp
template <CARG S>
constexpr auto make_char_array(S str_wrapped)
{
    constexpr string_view str = CARG_UNWRAP(str_wrapped);
    constexpr size_t len = str.size();
    array<char, len> result{};
    for (size_t i = 0; i < len; ++i) {
        result[i] = str[i];
    }
    return result;
}
```

```cpp
#define CARG typename
#define CARG_WRAP(x) [] { return x; }
#define CARG_UNWRAP(x) (x)()
```

```cpp
constexpr auto sa = make_char_array(CARG_WRAP("Hello"));
```

反射（使用类似 Mozi 的方法）

CPP-Summit 2024
全球C++及系统软件技术大会

```cpp
template <typename T>
void print(const T& obj, std::ostream& os = std::cout,
           const char* fieldName = "", int depth = 0)
{
    if constexpr (is_reflected_struct_v<T>) {
        os << indent(depth) << fieldName << (*fieldName ? ": {\n" : "{\n");
        for_each(obj, [depth, &os](const char* fieldName, const auto& value) {
            print(value, os, fieldName, depth + 1);
        });
        os << indent(depth) << "}" << (depth == 0 ? "\n" : ",\n");
    } else {
        os << indent(depth) << fieldName << ": " << obj << ",\n";
    }
}
```

https://godbolt.org/z/saejMW6Pq

反射打印的效果

```
DEFINE_STRUCT(
    Point,
    (double) x,
    (double) y);

DEFINE_STRUCT(
    Rect,
    (Point) p1,
    (Point) p2,
    (uint32_t) color);

print(Rect{
    {1.2, 3.4},
    {5.6, 7.8},
    12345678,
});
```

```
{
    p1: {
        x: 1.2,
        y: 3.4,
    },
    p2: {
        x: 5.6,
        y: 7.8,
    },
    color: 12345678,
}
```

PART 4

C++20/23 的 lambda 改进

lambda 表达式在 C++20 之前的一些限制

- 不能使用 lambda 表达式的类型来进行默认构造
- 不能在泛型 lambda 表达式里指定类型模板参数
- 不能在 lambda 表达式里捕获结构化绑定
- 不能在 lambda 表达式里引用自身
- ……

lambda 的类型用作 unique_ptr 的模板参数

在未求值上下文使用 lambda 表达式，并根据类型进行默认构造

```
unique_ptr<FILE, decltype([](FILE* fp) { fclose(fp); })>
    ptr{fopen(filename, "r")};
```

明确指定泛型 lambda 表达式的模板参数

明确给出泛型 lambda 表达式的类型模板参数名称

```cpp
auto f = []<typename T>(T&& x) {
    g(std::forward<T>(x));
};

// auto f = [](auto&& x) {
//     g(std::forward<decltype(x)>(x));
// };
```

可在函数体中直接使用

在 lambda 表达式里捕获结构化绑定

让结构化绑定跟普通的局部变量一样易用

```
auto [x, y] = …;

auto f = [x] { … };

// auto f = [x = x] { … };
```

在 lambda 表达式里引用自身

使用 deducing this 进行自引用

```cpp
auto factorial = [](this auto const& self, int n) -> int {
    assert(n >= 0);
    return n == 0 ? 1 : n * self(n - 1);
};
```

不再需要使用类似 Y combinator 的特殊工具

……及更少使用 lambda 表达式（C++26）

CPP-Summit 2024
全球C++及系统软件技术大会

```
template <typename T>
void print(const T& obj, ostream& os = cout, std::string_view name = "",
           int depth = 0)
{
    if constexpr (is_class_v<T>) {
        os << indent(depth) << name << (name != "" ? ": {\n" : "{\n");
        template for (constexpr meta::info member :
                       meta::nonstatic_data_members_of(^T)) {
            print(obj.[:member:], os, meta::identifier_of(member), depth + 1);
        }
        os << indent(depth) << "}" << (depth == 0 ? "\n" : ",\n");
    } else {
        os << indent(depth) << name << ": " << obj << ",\n";
    }
}
```

expansion statement 语法，执行编译期展开

https://cppx.godbolt.org/z/G3EcvhKxK

lambda 表达式是工具，不是目的！

lambda 表达式！

- 改变了我们组织代码的方式
- 在每个 C++ 标准中越来越强大、易用

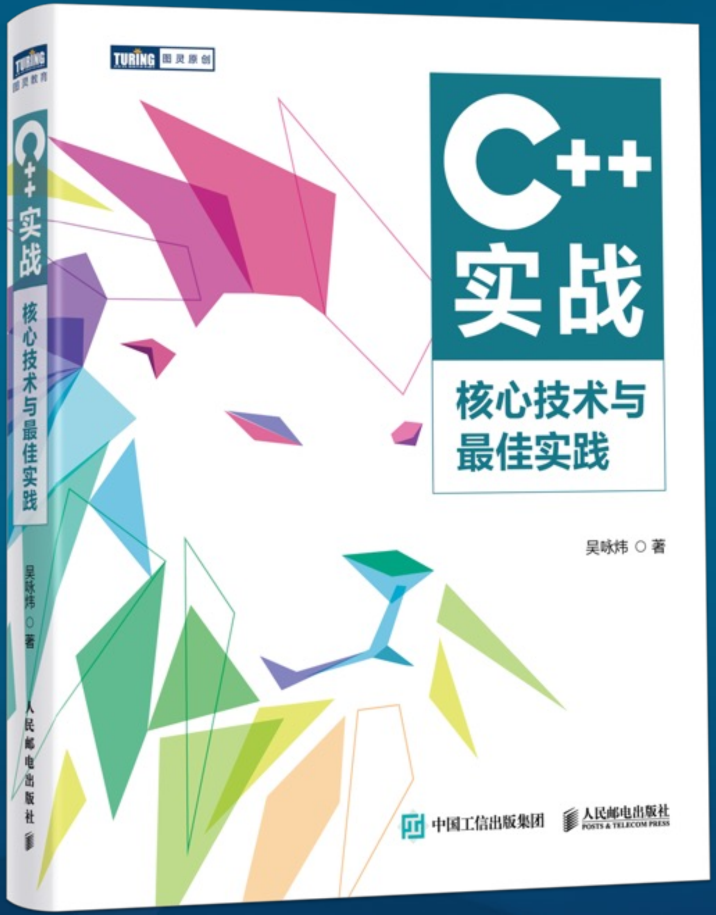
CPP-Summit 2024

全球C++及系统软件技术大会

谢谢大家！

广告

部分内容选摘自我的新书《C++ 实战：核心技术与最佳实践》。



促销签名版（会失效）

永久有效链接：

https://www.ituring.com.cn/book/3410（介绍，含样章）
https://github.com/adah1972/cpp_book1（含示例代码）
https://item.jd.com/14343213.html（京东销售页面）