C++11笔记

辅助知识

- 全文检索
  - vscode: `ctrl+shift+f`
- C++11相关网站
  - https://cplusplus.com
  - Standard C++ (isocpp.org)
  - C++11 FAQ (stroustrup.com)
  - https://cplusplus.com/reference/
- gcc与g++区别
  - gcc与g++的区别*gcc和g++区别*wsqyouth的博客-CSDN博客
- 指定c++版本
- vscode(code runner运行): `code runner` 的c++执行命令中添加参数 `-std=c++17`
- 测试c++版本

```
cout << __cplusplus << endl;
```

Variadic Templates

- 资料
  - c++11-17 模板核心知识（四）—— 可变参数模板 Variadic Template - 知乎 (zhihu.com)
- 侯捷STL课程中提到的Variadic Templates用法
  - 万用的hash function
  - tuple

C++11的小知识点

- `vector<list<int>>` 右边两个 `>` 之间现在不用加空格了（C++11开始）
- `nullptr` 替代了 `NULL` 或 `0`，使用 `NULL` 或 `0` 可能引起歧义
  - nullptr: std::nullptr_t类型(`<stddef>`)

initializer_list

- C++11允许用 `{}` 设初值

```
int values[]{1,2,3};
vector<string> vec{"str1","str2","str3"};
complex<double> c{4.0,3.0};
```

- {}的原理是：编译器看到{}形成 `initializer_list<T>`，其背后关联至一个 `array<T,n>`
  - 容器的ctor一般可以接收 `initializer_list<T>`
  - 对于一般的类，例如 `complex<double> c{4.0,3.0};`，array内的两个元素被分解传给ctor
- 使用注意

```
int x; // undefined
int x{}; // x=0;
int* p; //undefined
int* q{}; // q=nullptr
int x=5.0; // 强制转换
int x{5.0}; // error: narrowing conversion of '5.0e+0' from 'double' to 'int'
```

- initializer_list: https://cplusplus.com/reference/initializer_list/initializer_list/
- intializer_list ctor
  - 如果没有initializer_list ctor，可以通过array分解initializer_list调用构造函数，但参数必须一一对应

```
struct myclass {
  myclass (int,int);
  myclass (initializer_list<int>);
  /* definitions ... */
};

myclass foo {10,20};  // calls initializer_list ctor，如果没有initializer_list ctor，仍然可以通过
myclass bar (10,20);  // calls first constructor
myclass foo2 {10,20,30};  // 如果没有定义initializer_list ctor，则不能通过
```

- intializer_list 不内含 array，只存储array的begin iterator和length
- 其他intializer_list的使用示例

```
min({1,2,3,4});
max({"acd","ace","bcd","bce"});

vector<int> vec({1,2,5,6});
vec.emplace(vec.begin()+2,{3,4});
```

explicit

- C++11后explicit可以接收多个实参
- explicit specifier - cppreference.com

range-based for

- Range-based for loop (since C++11) - cppreference.com
- for-loop语句原理如下

```
init-statement
auto && __range = range-expression ;
auto __begin = begin-expr ;
auto __end = end-expr ;
for ( ; __begin != __end; ++__begin)
{
    range-declaration = *__begin; //取出内容做赋值
    loop-statement
}
```

- 如果声明了explicit，小心隐式类型转换不成功

```
class MyClass{
    public:
     explicit MyClass(string s){...} //explicit声明导致不能通过string类型赋值,例如 MyClass c = string("str1"); 会报错
}

//test
vector<string> strVec{"tet","tee2"};
for(const MyClass& c: strVec ){...}
// error: invalid initialization of reference of type 'const MyClass&' from expression of type 'std::__cxx11::basic_string<char>
```

=default和=delete

- 用户定义构造函数后，会阻止编译器生成默认构造函数
  - =default会强制保留编译器给的默认版本（只有 默认(无参)构造 / 拷贝构造 / 拷贝赋值 / 移动构造 / 移动赋值 / 析构 有默认版本）
  - =delete会强制删除编译器给的默认版本，并且不能再定义

```
struct A
{
    int x;
    A(int x = 1): x(x) {} // user-defined default constructor
};

struct B: A
{
    // B::B() is implicitly-defined, calls A::A()
};
```

```cpp
struct C
{
    A a;
    // C::C() is implicitly-defined, calls A::A()
};

struct D: A
{
    D(int y): A(y) {}
    // D::D() is not declared because another constructor exists
};

struct E: A
{
    E(int y): A(y) {}
    E() = default; // explicitly defaulted, calls A::A()
};

struct F
{
    int& ref; // reference member
    const int c; // const member
    // F::F() is implicitly defined as deleted
};

// user declared copy constructor (either user-provided, deleted or defaulted)
// prevents the implicit generation of a default constructor

struct G
{
    G(const G&) {}
    // G::G() is implicitly defined as deleted
};

struct H
{
    H(const H&) = delete;
    // H::H() is implicitly defined as deleted
};

struct I
{
    I(const I&) = default;
    // I::I() is implicitly defined as deleted
};

int main()
{
    A a;
    B b;
    C c;
//  D d; // compile error,因为定义了构造函数
    E e;
//  F f; // compile error
//  G g; // compile error，因为定义了拷贝构造函数
//  H h; // compile error，因为定义了拷贝构造函数 = delete;
//  I i; // compile error，因为定义了拷贝构造函数 = default;
}
```

- 只能对有默认版本的6个函数添加 `=default`
  - 这6个函数，如果已经实现了自己的版本，则不能再添加 `=default`
  - 自定义带参构造和其他函数，不允许使用 `=default` 修饰（即使有默认参数也不行）
- `=delete` 显式删除可以避免用户使用一些不应该使用的类的成员函数，使用这种方式可以有效的防止某些类型之间**自动进行隐式类型转换产生的错误。**
  - =delete修饰函数后不能再重新实现该函数
  - =delete可以作用于任何函数，例如修饰不同参数类型的构造函数，以防止隐式类型转换
- 一个比较特殊的地方，当用 `=delete` 显示删除默认构造函数后，会出现如下状况

```cpp
struct J {
  J() = delete;
};

J j;   // 不通过
J j1();// 通过
```

alias template

- 参考资料：Type alias, alias template (since C++11) - cppreference.com
- 基础使用

```
template <typename T>
using Vec = std::vector<T,allocator<T>>;

Vec<int> v;
```

- 配合 `模板模板参数` 使用

```
// template<typename> class Container就是一个模板模板参数，其本身应该写成template<typename T> class Container
// 但由于前后两个T相同，就把一个T省略掉了
template <typename T,template<typename> class Container>
class TestContainer{
private:
    Container<T> c;
public:
    TestContainer(){
        srand(time(NULL));
        for(int i=0;i<30000000;i++){
        c.insert(c.end(),T());
        cout<<*(c.end()-1);
    }
};
```

```
template <typename T>
using Vec = std::vector<T,allocator<T>>;
//使用alias template，使得传入的Vec是只有一个模板参数的模板，否则无法与接口对应
TestContainer<MyValueType,Vec> c;
```

type alias

- 参考资料：Type alias, alias template (since C++11) - cppreference.com
- 使用：`using aliasName = someType;`

```
// 等价于 typedef std::ios_base::fmtflags flags;
using flags = std::ios_base::fmtflags;

 //等价于 typedef void (*func)(int, int);
using func = void (*) (int, int);

// the name 'func' now denotes a pointer to function:
void example(int, int) {}
func f = example;
```

- `using` 用法总结：
  - `using namespace std;` 声明命名空间
  - `using std::cout;` 声明命名空间中的成员
  - `using BaseClass::SomeMember;` 声明类中的成员
  - `type alias && template alias`

noexcept

- 参考资料：noexcept specifier (since C++11) - cppreference.com
- 意义：修饰函数，通知编译器函数不丢出异常，如果丢出异常，调用 `std::terminate->std::abort`

```
void f() noexcept; // the function f() does not throw
void (*fp)() noexcept(false); // fp points to a function that may throw
void g(void pfa() noexcept);  // g takes a pointer to function that doesn't throw
// typedef int (*pf)() noexcept; // error, it cannot appear in a typedef or type alias declaration.
```

```
// whether foo is declared noexcept depends on if the expression
// T() will throw any exceptions
template<class T>
void foo() noexcept(noexcept(T())) {}
```

```cpp
void bar() noexcept(true) {}
void baz() noexcept { throw 42; } // noexcept is the same as noexcept(true)

int main()
{
    foo<int>(); // noexcept(noexcept(int())) => noexcept(true), so this is fine

    bar(); // fine
    baz(); // compiles, but at runtime this calls std::terminate
}
```

- move构造/赋值，需要用noexcept修饰，否则对于vector(可扩容、有迁移元素行为)容器来说，在迁移过程中不会调用move构造，只会调用拷贝构造

override

- 参考资料：override specifier (since C++11) - cppreference.com
- 意义：用于修饰虚函数，通知编译器检查基类是否有相同签名的虚函数，若没有则报错
- override不是保留词，可以用于函数名或变量名

```cpp
#include <iostream>

struct A
{
    virtual void foo();
    void bar();
    virtual ~A();
};

// member functions definitions of struct A:
void A::foo() { std::cout << "A::foo();\n"; }
A::~A() { std::cout << "A::~A();\n"; }

struct B : A
{
//  void foo() const override; // Error: B::foo does not override A::foo
                               // (signature mismatch)
    void foo() override; // OK: B::foo overrides A::foo
//  void bar() override; // Error: A::bar is not virtual
    ~B() override; // OK: `override` can also be applied to virtual
                   // special member functions, e.g. destructors
    void override(); // OK, member function name, not a reserved keyword
};

// member functions definitions of struct B:
void B::foo() { std::cout << "B::foo();\n"; }
B::~B() { std::cout << "B::~B();\n"; }
void B::override() { std::cout << "B::override();\n"; }

int main()
{
    B b;
    b.foo();
    b.override(); // OK, invokes the member function `override()`
    int override{42}; // OK, defines an integer variable
    std::cout << "override: " << override << '\n';
}
```

final

- 参考资料：final specifier (since C++11) - cppreference.com
- 意义：修饰类或函数，表示类不能被继承，或者函数不能被覆写

```cpp
struct Base
{
    virtual void foo();
};

struct A : Base
```

```
{
    void foo() final; // Base::foo is overridden and A::foo is the final override
    void bar() final; // Error: bar cannot be final as it is non-virtual
};

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden as it is final in A
};

struct C : B {}; // Error: B is final
```

decltype

- 参考资料：[decltype specifier - cppreference.com]
- 意义：声明某个表达式的类型，类似于typeof(xxx)
    - 由于lambda函数是匿名的functor，经常需要用decltype声明类型

```
#include <iostream>
#include <type_traits>

struct A { double x; };
const A* a;

decltype(a->x) y;       // type of y is double (declared type)
decltype((a->x)) z = y; // type of z is const double& (lvalue expression)

template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u) // return type depends on template parameters
                                      // return type can be deduced since C++14
{
    return t + u;
}

const int& getRef(const int* p) { return *p; }
static_assert(std::is_same_v<decltype(getRef), const int&(const int*)>);
auto getRefFwdBad(const int* p) { return getRef(p); }
static_assert(std::is_same_v<decltype(getRefFwdBad), int(const int*)>,
    "Just returning auto isn't perfect forwarding.");
decltype(auto) getRefFwdGood(const int* p) { return getRef(p); }
static_assert(std::is_same_v<decltype(getRefFwdGood), const int&(const int*)>,
    "Returning decltype(auto) perfectly forwards the return type.");

// Alternatively:
auto getRefFwdGood1(const int* p) -> decltype(getRef(p)) { return getRef(p); }
static_assert(std::is_same_v<decltype(getRefFwdGood1), const int&(const int*)>,
    "Returning decltype(return expression) also perfectly forwards the return type.");

int main()
{
    int i = 33;
    decltype(i) j = i * 2;

    std::cout << "i and j are the same type? " << std::boolalpha
              << std::is_same_v<decltype(i), decltype(j)> << '\n';

    std::cout << "i = " << i << ", "
              << "j = " << j << '\n';

    auto f = [](int a, int b) -> int
    {
        return a * b;
    };

    decltype(f) g = f; // the type of a lambda function is unique and unnamed
    i = f(2, 2);
    j = g(3, 3);

    std::cout << "i = " << i << ", "
              << "j = " << j << '\n';
}
```

lambda

- 参考资料：[Lambda expressions (since C++11) - cppreference.com]()
- 意义：声明一个匿名的仿函数，其是内联的，可以像用对象一样使用
    - lambda是匿名的，`没有默认构造函数`，如果要当成仿函数一样用，需要小心，比如创建 `关联式容器` 和 `unordered容器` 时，需要传入用于比较的仿函数对象

```
class MyClass{
    bool operator <(const MyClass& other) const {
        ...
    }
};
// Cmp是仿函数的一个对象
auto Cmp = [](const MyClass& m1,const MyClass& m2){
    return m1<m2;
};
set<MyClass,decltype(Cmp)> MySet; //error，因为会调用decltype(Cmp)的默认构造函数创建一个仿函数的对象，但因为lambda没有构造函数会出错
set<MyClass,decltype(Cmp)> MySet(Cmp); //必须传入Cmp
```

- 基础格式：[`captures`] (`params`) `specs requires(optional)` { `body` }

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> c = {1, 2, 3, 4, 5, 6, 7};
    int x = 5;
    c.erase(std::remove_if(c.begin(), c.end(), [x](int n) { return n < x; }), c.end());

    std::cout << "c: ";
    std::for_each(c.begin(), c.end(), [](int i){ std::cout << i << ' '; });
    std::cout << '\n';

    // the type of a closure cannot be named, but can be inferred with auto
    // since C++14, lambda could own default arguments
    auto func1 = [](int i = 6) { return i + 4; };
    std::cout << "func1: " << func1() << '\n';

    // like all callable objects, closures can be captured in std::function
    // (this may incur unnecessary overhead)
    std::function<int(int)> func2 = [](int i) { return i + 4; };
    std::cout << "func2: " << func2(6) << '\n';

    constexpr int fib_max {8};
    std::cout << "Emulate `recursive lambda` calls:\nFibonacci numbers: ";
    auto nth_fibonacci = [](int n)
    {
        std::function<int(int, int, int)> fib = [&](int n, int a, int b)
        {
            return n ? fib(n - 1, a + b, a) : b;
        };
        return fib(n, 0, 1);
    };

    for (int i{1}; i <= fib_max; ++i)
    {
        std::cout << nth_fibonacci(i) << (i < fib_max ? ", " : "\n");
    }

    std::cout << "Alternative approach to lambda recursion:\nFibonacci numbers: ";
    auto nth_fibonacci2 = [](auto self, int n, int a = 0, int b = 1) -> int
    {
        return n ? self(self, n - 1, a + b, a) : b;
    };

    for (int i{1}; i <= fib_max; ++i)
    {
        std::cout << nth_fibonacci2(nth_fibonacci2, i) << (i < fib_max ? ", " : "\n");
```

```
    }

#ifdef __cpp_explicit_this_parameter
    std::cout << "C++23 approach to lambda recursion:\n";
    auto nth_fibonacci3 = [](this auto self, int n, int a = 0, int b = 1)
    {
         return n ? self(n - 1, a + b, a) : b;
    };

    for (int i{1}; i <= fib_max; ++i)
    {
        std::cout << nth_fibonacci3(i) << (i < fib_max ? ", " : "\n");
    }
#endif
}
```

```
c: 5 6 7
func1: 10
func2: 10
Emulate `recursive lambda` calls:
Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13
Alternative approach to lambda recursion:
Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13
```

- `[captures]`：可以传入外部的变量，mutable表示可以修改， `=` 传值， `&` 传引用

```
struct S2 { void f(int i); };
void S2::f(int i)
{
    [&]{};          // OK: by-reference capture default
    [&, i]{};       // OK: by-reference capture, except i is captured by copy
    [&, &i] {};     // Error: by-reference capture when by-reference is the default
    [&, this] {};   // OK, equivalent to [&]
    [&, this, i]{}; // OK, equivalent to [&, i]
}
```

```
struct S2 { void f(int i); };
void S2::f(int i)
{
    [=]{};        // OK: by-copy capture default
    [=, &i]{};    // OK: by-copy capture, except i is captured by reference
    [=, *this]{}; // until C++17: Error: invalid syntax
                  // since C++17: OK: captures the enclosing S2 by copy
    [=, this] {}; // until C++20: Error: this when = is the default
                  // since C++20: OK, same as [=]
}
```

```
#include <iostream>

int main()
{
    int a = 1, b = 1, c = 1;

    auto m1 = [a, &b, &c]() mutable
    {
        auto m2 = [a, b, &c]() mutable
        {
            std::cout << a << b << c << '\n';
            a = 4; b = 4; c = 4;
        };
        a = 3; b = 3; c = 3;
        m2();
    };

    a = 2; b = 2; c = 2;

    m1();                             // calls m2() and prints 123
    std::cout << a << b << c << '\n'; // prints 234
}
```

Rvalue Reference

- 右值：临时对象，不可放在 `=` 左边
- `std::move` 可以将任何一值变成右值，变成右值后不能再使用【Defined in header `<utility>`】
- 资料
  - https://cplusplus.com/reference/utility/move/
  - Move constructors - cppreference.com
  - std::move - cppreference.com

```
struct A
{
    std::string s;
    int k;

    A() : s("test"), k(-1) {}
    A(const A& o) : s(o.s), k(o.k) { std::cout << "move failed!\n"; }
    A(A&& o) noexcept :
        s(std::move(o.s)),       // explicit move of a member of class type
        k(std::exchange(o.k, 0)) // explicit move of a member of non-class type
    {}
};
```

```
#include <iomanip>
#include <iostream>
#include <string>
#include <utility>
#include <vector>

int main() {
  std::string str = "Salut";
  std::vector<std::string> v;

  // uses the push_back(const T&) overload, which means
  // we'll incur the cost of copying str
  v.push_back(str);
  std::cout << "After copy, str is " << std::quoted(str) << '\n';

  // uses the rvalue reference push_back(T&&) overload,
  // which means no strings will be copied; instead, the contents
  // of str will be moved into the vector.  This is less
  // expensive, but also means str might now be empty.
  v.push_back(std::move(str));
  std::cout << "After move, str is " << std::quoted(str) << '\n';

  std::cout << "The contents of the vector are { " << std::quoted(v[0]) << ", "
            << std::quoted(v[1]) << " }\n";
  system("Pause");
}
```

forward

- 问题：右值传入函数后再次传递后变成左值
- 解决：使用 `std::forward`【Defined in header `<utility>`】
- 资料：std::forward - cppreference.com

```
template<class T>
void wrapper(T&& arg)
{
    // arg is always lvalue
    foo(std::forward<T>(arg)); // Forward as lvalue or as rvalue, depending on T
}
```

move ctor对容器的影响

- 插入容器中的类型需要实现move ctor/move assignment（浅拷贝）
- move ctor对元素插入容器的影响
  - 对vector影响大，vector需要扩容和迁移元素
  - 对其他节点类型的容器影响较小，deque/list/map/set等
- move ctor对复制容器的影响
  - 对所有容器的影响都很大，因为是浅拷贝，所以不需要复制容器元素