Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 4
26.09.2020

Захват данных класса

```
class T {
  private:
    std::vector<int> Data;
  public:
    T(const std::vector<int>& data)
        :Data(data)
    {}
    void Do() {
        auto f = [=](int a, int b) {
            return Data[a] < Data[b];
        };
    }
};
```

Указатели на методы внутри класса

```
class C {
 private:
  int a, b;
 public:
  C(int a, int b) : a(a), b(b) {};
  void f() const { ... }
  void g() const { ... }
};
```

Хотим сделать указатель на функцию f из класса,

```
void (*ptr) ();                 //
ptr = &C::f;                    // так нельзя

void (C::*ptr)() const;
ptr = &C::f;                    // ok
```

mem_fn в C++11

std::mem_fn

Defined in header <functional>

```
template< class M, class T >                                  (since C++11)
/*unspecified*/ mem_fn(M T::* pm) noexcept;                   (until C++20)

template< class M, class T >
constexpr /*unspecified*/ mem_fn(M T::* pm) noexcept;         (since C++20)
```

Function template `std::mem_fn` generates wrapper objects for pointers to members, which can store, copy, and invoke a pointer to member. Both references and pointers (including smart pointers) to an object can be used when invoking a `std::mem_fn`.

variadic — может работать с произвольным числом аргументов.

```
C c(1);
auto g = std::mem_fn(&C::f);

// the same:
auto g2 = [] (C& c, int x) { return c.f(x); } ;
g(c, 10);
```

std::mem_fn

```
class C {
    int a;
  public:
    C(int a) : a(a) {}
    void f() const {
        std::cout << a << std::endl;
    }
};

int main() {
    C c [] = {C(1), C(2), C(3), C(4)};
    C* cptr [] = {c, c + 1, c + 2, c + 3};
    std::for_each(c, c + 4, std::mem_fn(&C::f));
    std::for_each(cptr, cptr + 4, std::mem_fn(&C::f));
    std::for_each(c, c + 4,
                  std::function<void(const C&)>(&C::f));
    std::for_each(cptr, cptr + 4,
                  std::function<void(const C*)>(&C::f));
}
```

std::not_fn

Создает функцию с инвертированным результатом.

```
std::vector<int> v1 = { /*....*/ };
auto divisible_by_3 = [](int i){ return i % 3 == 0; };

int divisible =
    std::count_if(v1.begin(), v1.end(), divisible_by_3);

int not_divisible =
    std::count_if(v1.begin(), v1.end(),
    std::not_fn(divisible_by_3));
```

Ссылки и копии

(C++ Core Guidelines)
F.16: For "in"parameters, pass cheaply-copied types by value and others by reference to const
Both let the caller know that a function will not modify the argument, and both allow initialization by rvalues. What is “cheap to copy” depends on the machine architecture, but two or three words (doubles, pointers, references) are usually best passed by value. When copying is cheap, nothing beats the simplicity and safety of copying, and for small objects (up to two or three words) it is also faster than passing by reference because it does not require an extra indirection to access from the function.

Ссылки и копии

(C++ Core Guidelines)
F.20: For "out"output values, prefer return values to output parameters
A return value is self-documenting, whereas a & could be either in-out or out-only and is liable to be misused. This includes large objects like standard containers that use implicit move operations for performance and to avoid explicit memory management.
If you have multiple values to return, use a tuple or similar multi-member type.

Строки

```
void f(const std::string& s);
```

Что если нужно вызвать эту функцию от подстроки или от `char*`?

```
void f(std::string_view s);
```

std::string_view

```
std::string str = "Crazy Fredrick bought many jewels";

// плохо: создаем новую строчку
std::cout << str.substr(10, 10) << std::endl;

// норм: никакого копирования
std::string_view view = str;

// string_view::substr возвращает новый string_view
std::cout << view.substr(10, 10) << std::endl;
```

Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void DoConst(const std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

int main() {
    Do(std::vector<int>(10));
    DoConst(std::vector<int>(10));
}
```

Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void Do(const std::vector<int> &v) {
    std::cout << "const v.size() == " << v.size() << std::endl;
}

int main() {
    std::vector<int> v(10);
    Do(v);
    Do(std::vector<int>(10));
}
```

Типы ссылок

```
#include <iostream>

int rvalue() {
    return 5;
}

int& lvalue() {
    static int tmp = 0;
    return tmp;
}

int main() {
    &lvalue();  // ok
    &rvalue();  // error
    lvalue() = rvalue(); // ok
}
```

move

```
template <typename T>
void Swap(T& a, T& b) {
    T t(a);
    a = b;
    b = t;
}
```

```
template <typename T>
void Swap(T& a, T& b) {
    T t(std::move(a));
    a = std::move(b);
    b = std::move(t);
}
```

Еще пример

```
A& ref = A(); // error
A&& ref = A(); // ok
```

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

int main() {
    std::string x = "abc";
    std::string y = std::move(x);
    std::cout << x << "-" << y << std::endl;
    return 0;
}
```

Типы ссылок

```
void f(T&& x);  // rvalue-ссылка

T&& x = T();  // rvalue-ссылка

auto&& y = x; // не rvalue-ссылка

template<typename T>
void f(T&& x);  // не rvalue-ссылка

template<typename T>
void f(std::vector<T>&& x);  // rvalue-ссылка
```

Типы ссылок

При инициализации универсальной ссылки определяется, какую ссылку она представляет – rvalue/lvalue.

```
template<typename T>
void f(T&& x);  // не rvalue-ссылка

T a;
f(a); // lvalue-ссылка
f(std::move(a)); // rvalue-ссылка
```

Типы ссылок

```
template<typename T>
void f(std::vector<T>&& x);

std::vector<int> v;
f(v); // error
```

```
template<typename T>
void f(const T&& x); // rvalue-ссылка
```

Типы ссылок

- **lvalue**
  - обычные ссылки, константные ссылки, ...
  - это не временный объект и не тот объект, который вскоре будет уничтожен.
- **xvalue**
  - объект который вот-вот должен быть уничтожен (expired)
  - пример — то, что приходит в оператор перемещения
- **prvalue**
  - pure, настоящее rvalue
  - пример — результат вызова функции, у которой возвращаемое значение – не ссылка.

glvalue — lvalue или xvalue
rvalue — xvalue, или временный объект, или значение, не ассоциированное с объектом.

xvalue: два примера

1.
```
int&& f(){   return 3; }
// ...
f();
```

2.
```
static_cast<int&&>(7);
std::move(7);
```

Типы ссылок: практическое правило

1. Если у выражения можно взять адрес — это **lvalue**
2. Если тип выражения — `T&` или `const T&`, — это **lvalue**
3. Иначе это **rvalue**. Обычно — литералы, результат вызова функций и т. п.

emplace_back

```
int main() {
    std::vector<A> as;
    // as.push_back(A(3));
    as.reserve(5);
    as.emplace_back(1);
    as.emplace_back(2);
    as.emplace_back(3);
    as.emplace_back(4);
    as.emplace_back(5);
}
```

std::move

```
class TSuperClass {
  public:
    // ...
    TSuperClass(const TSuperClass&);
    TSuperClass(TSuperClass&&);
    // ...
};
class TMyType {
  public:
    TMyType(const TSuperClass val) : field(std::move(val));
  private:
    TSuperClass field;
}
```

В чем тут проблема?