Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 2
18.09.2019

Псевдонимы

typedef:

```
typedef
    std::shared_ptr<std::map<std::string, std::string> >
    TMyPtr;
typedef bool (*FPtr)(int, int);
```

using (C++11):

```
using TMyPtr =
    std::shared_ptr<std::map<std::string, std::string> >;
using FPtr = bool (*)(int, int);
```

В чем отличие typedef от using?

Псевдонимы

typedef:

```
typedef
    std::shared_ptr<std::map<std::string, std::string> >
    TMyPtr;
typedef bool (*FPtr)(int, int);
```

using (C++11):

```
using TMyPtr =
    std::shared_ptr<std::map<std::string, std::string> >;
using FPtr = bool (*)(int, int);
```

В чем отличие typedef от using?

Объявление псевдонимов поддерживает шаблонизацию.

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

scoped enumerations

```
enum Color {black, white, blue};
bool white; // error!
```

C++11:

```
enum class Color { red, green = 20, blue };
Color r = Color::blue;
switch (r) {
    case Color::red:  // ...
    case Color::green:  // ...
    case Color::blue:  // ...
}
int n = r; // ошибка
int n = static_cast<int>(r);
```

Базовый тип — int.

constexpr

```
const int a = 10;
const int b = std::numeric_limits<int>::max();
const int c = INT_MAX;
```

```
int a;
const int b = a; // ok
constexpr auto s = a; // error
```

```
constexpr int f() {return 1024;}
```

constexpr-функция должна состоять из одного return (C++11), возвращать константу или вызывать такую же функцию. Вычисление должно производиться во время компиляции (с аргументами, значения которых известны во время компиляции).

Пример: проверка простоты числа в compile-time

```
constexpr bool is_div(int a, int b) {
  return (b == 1) || (a % b != 0 && is_div(a, b - 1) );
}

constexpr bool is_prime(int number) {
  return number != 1 && is_div(number, number / 2);
}

int main() {
  static_assert(is_prime(29) , " 29 is not prime");
  static_assert(is_prime(36) , " 36 is prime");
  return 0;
}
```

Еще про constexpr

Вычисления в момент компиляции:

```
constexpr int fact(int n) {
    return n == 0 ? 1 : n * fact(n - 1);
}

static_assert (fact(7) == 5040);
```

- только return;
- только операции над константами и аргументами;
- можно вызывать contexpr-функции;
- sizeof;
- можно рекурсию и тернарный оператор.

Еще про constexpr

C++14:

```
constexpr int fact(int n) {
    int result = 1;
    for (int i = 0; i <= n; ++i) result *= i;
    return result;
}
```

Еще про constexpr

C++17:

```
constexpr auto GetArray() {
    std::array<int, 3> a = {1, 2, 3};
    a[0] = 5;
    return a;
}

int main() {
    auto x = GetArray();
    cout << x[0];
}
```

constexpr if

C++17:

```
if constexpr (/*constant expression */) {
    // if true this block is compiled
} else {
    // if false this block is compiled
}
```

Variadic templates

Шаблоны с переменным числом аргументов (C++11).

```
#include <iostream>
void myprintf (const char *str) {
    std::cout << str;
}

template <typename T, typename... Targs>
void myprintf(const char* str, T value, Targs... Fargs) {
    for (; *str != '\0' ; ++str) {
        if (*str == '%') {
            std::cout << value;
            myprintf(str + 1, Fargs...);
            return;
        }
        std::cout << *str;
    }
}
```

Variadic templates

Получение i-го значения:

```
#include <iostream>

template <unsigned n, class T, class... Args>
constexpr auto Get(T value, Args... args) {
    if constexpr(n > sizeof...(args)) {
        return;
    } else if constexpr (n > 0) {
        return Get<n - 1>(args...);
    } else {
        return value;
    }
}

int main() {
    std::cout << (Get<2>(1, "abc", 'c'));
}
```

Variadic templates

На этом определен std::tuple.

```
template <typename... Args>
class Tuple;
template<>
class Tuple{};
```

```
template <typename Head, typename... Tail>
class Tuple<Head,Tail...> : Tuple<Tail...> {
   // ...
}
```

std::tuple

```
auto t = std::make_tuple(1, 10.0, "abc");
// std::get<0>(t); std::get<1>(t); std::get<2>(t);
```

std::tuple

```
auto t = std::make_tuple(1, 10.0, "abc");
// std::get<0>(t); std::get<1>(t); std::get<2>(t);
```

Если хочется вернуть кортеж из функции, то приходилось писать при помощи std::tie:

```
std::tuple<int, std::string, int> func();

int a, b;
std::string s;
std::tie(a, s, b) = func();
```

std::tuple

```
auto t = std::make_tuple(1, 10.0, "abc");
// std::get<0>(t); std::get<1>(t); std::get<2>(t);
```

Если хочется вернуть кортеж из функции, то приходилось писать при помощи std::tie:

```
std::tuple<int, std::string, int> func();

int a, b;
std::string s;
std::tie(a, s, b) = func();
```

C++17:

```
auto [a, s, b] = func();
```

Структурное связывание

```
// C++17
for (const auto &[key, value] : get_pairs()) {
    // do smth with key, value
}
```

Сортировка

```
template <typename T>
void sort(T * begin, T * end) {
    for (T* p1 = begin; p1 != end; ++p1) {
        for (T* p2 = p1 + 1; p2 != end; ++p2) {
            if (*p1 > *p2)
                std::swap(*p1, *p2);
        }
    }
}

int main() {
    int a[] = {3, 5, 2, 8, 8, 1 ,0, 15, 12, -1, 3, 4, 7};
    sort(a, a + sizeof(a) / sizeof(a[0]));
    for (const auto &v : a) {
        std::cout << v << " ";
    }
    std::cout << std::endl;
}
```

Сортировка

```
template <typename T>
bool CompareTLess(const T&a, const T&b) { return a < b;}

template <typename T>
bool CompareTGreater(const T&a, const T&b) { return a > b;}

template <typename T>
void sort(T * begin, T * end, bool (*cmp)(const T&, const T&)) {
    for (T* p1 = begin; p1 != end; ++p1) {
        for (T* p2 = p1 + 1; p2 != end; ++p2) {
            if (cmp(*p2, *p1))
                std::swap(*p1, *p2);
        }
    }
}
// ...
sort(a, a + sizeof(a) / sizeof(a[0]), &CompareTLess<int>);
sort(a, a + sizeof(a) / sizeof(a[0]), &CompareTGreater<int>);
```

Функторы

```
template <typename T>
class TComparer {
    private:
        bool IsLess;
    public:
        TComparer(bool IsLess)
            : IsLess(IsLess)
        {}
        bool operator() (const T&a, const T&b) const {
            return IsLess ? a < b : a > b;
        }
};
```

Функторы

```
template <typename T, typename TComparerType>
void sort(T * begin, T * end, const TComparerType& cmp) {
    for (T* p1 = begin; p1 != end; ++p1) {
        for (T* p2 = p1 + 1; p2 != end; ++p2) {
            if (cmp(*p2, *p1))
                std::swap(*p1, *p2);
        }
    }
}
// ...

sort(a, a + sizeof(a) / sizeof(a[0]), TComparer<int>(true));
sort(a, a + sizeof(a) / sizeof(a[0]), TComparer<int>(false));
```

Функторы

Функции сравнения, конечно же, уже есть — std::less, std::greater

```
sort(a, a + sizeof(a) / sizeof(a[0]), std::less<int>());

sort(a, a + sizeof(a) / sizeof(a[0]), std::greater<int>());
```

Что неудобно, если это не готовый функтор? Функции описываются вне места применения.
Локальные функции и шаблонные классы запрещены.

Функторы

Можно так:

```
int main() {
    struct TC {
        bool operator() (int a, int b) const {
            return a < b;
        }
    };
    int a[] = {3, 5, 2, 8, 8, 1 ,0, 15, 12, -1, 3, 4, 7};

    sort(a, a + sizeof(a) / sizeof(a[0]), TC());
    for (const auto &v : a) {
        std::cout << v << " ";
    }
    std::cout << std::endl;
}
```

Но это некрасиво.

lambda-объекты

```
sort(a, a + sizeof(a) / sizeof(a[0]),
    [](int a, int b) {
        return a < b;
    }
);
```

Лямбда-выражения

Быстрый способ создать такую структуру с оператором ():

```
struct T {
    bool operator()(int x){};
};
Do(T(), ...);
```

Лямбда-выражения

Быстрый способ создать такую структуру с оператором ():

```
struct T {
    bool operator()(int x){};
};
Do(T(), ...);
```

- [] — список переменных, которые захватывает лямбда-выражение;
- () — входные аргументы функции;
- {} — тело функции.

Лямбда-выражения

```
[capture] (params) mutable exception_attribute -> ret {body}
[capture] (params) -> ret {body}
[capture] (params) {body}
[capture] {body}
```

Пример:

```
std::vector<int> v = {-1, -2, -3, -4, -5, 1, 2, 3, 4 ,5};
std::sort(v.begin(), v.end(), [](int l, int r) {
    return l * l < r * r;
});
```

Лямбда-выражения

- `[]` — без захвата переменных
- `[=]` — все переменные захватываются по значению
- `[&]` — все переменные захватываются по ссылке
- `[x]` — захват `x` по значению
- `[&x]` — захват `x` по ссылке
- `[x, &y]` — захват `x` по значению, `y` по ссылке
- `[=, &x, &y]` — захват всех переменных по значению, но `x`,`y` – по ссылке
- `[&, x]` — захват всех переменных по ссылке, кроме `x`
- `[this]` — для доступа к переменной класса

return

Вот так не работает:

```
auto cmp = [&data](int a, int b) {
    if (a == 12)
        return -1;
    return data[a] < data[b];
};
```

return

Вот так не работает:

```
auto cmp = [&data](int a, int b) {
    if (a == 12)
        return -1;
    return data[a] < data[b];
};
```

А вот так всё хорошо:

```
auto cmp = [&data](int a, int b) -> int {
    if (a == 12)
        return -1;
    return data[a] < data[b];
};
```

mutable

Те элементы, которые захвачены по значению, автоматически становятся константами внутри лямбды.

```
auto cmp = [&d, d1](int a, int b) mutable -> int {
    d[0] = d1[0] = 0;
    if (a == 12)
        return -1;
    return d[a] < d[b];
};
```

Захват данных класса

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

Упражнение

Отсортировать массив, не испортив его — вывести перестановку, сохранив исходные данные.

```
    const int a[] = {3, 5, 2, 8, 15, 12, -1, 3, 4, 7};  // 
    size_t n = sizeof(a) / sizeof(a[0]);
    std::vector<size_t> idx(n);
    for (int i = 0; i < n; ++i) {
        idx[i] = i;
    }

    // ... ?

    for (const auto &i : idx) {
        std::cout << a[i] << " ";
    }
    std::cout << std::endl;
}
```