Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 3
21.09.2018

Задача

Реализовать `TStaticAssert` без использования C++11. Ловим ошибки на этапе компиляции:

```
TStaticAssert<1 == 1> assert_one_is_one;
```

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, throw;
- 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;
}
```

Опасности захвата по умолчанию

```
using T = std::vector<std::function<bool(int)>>;

void AddFunc(T& funcs) {
    static int x = 2;
    funcs.emplace_back(
        [=](int v) { return v % x == 0;}
    );
    ++x;
}

int main() {
    T funcs;
    AddFunc(funcs);
    AddFunc(funcs);
    funcs[0](5);
    funcs[1](5);
}
```

Захватывать явно — заметнее ошибки.

Захватывать указатель тоже может быть опасно.

C++17 constexpr-lambdas

```
constexpr auto add(int y) {
    return [=](int x) { return x + y;};
}
int main() {
    constexpr auto inc = add(5);
    static_assert(inc(3) == 8);
}
```

std::function

```
struct Foo {
    void print(int i) const { std::cout << i << std::endl; }
};
void print_num(int i) {
    std::cout << i << std::endl;
}
int main() {
    std::function<void(int)> f_display = print_num;
    f_display(1);

    std::function<void()> f_display_2 = []() { print_num(2); };
    f_display_2();

    std::function<void(const Foo&, int)> f_add_display =
            &Foo::print;
    f_add_display(Foo(), 1);
}
```

Отступление: 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
```

Отступление: 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
```

Функторы в <functional>

```
greater
less
greater_equal
less_equal
equal_to
not_equal_to
plus
minus
multiply
divide
negate -
modulus
logical_and
logical_or
logical_not
```

Функторы в <functional>

```
greater
less
greater_equal
less_equal
equal_to
not_equal_to
plus
minus
multiply
divide
negate -
modulus
logical_and
logical_or
logical_not
```

```
sort(a, a + 5, std::greater<int>());
transform(a, a + 5, b, ostream_iterator<int> (cout, " "),
          plus<int>());
```

Связыватели в C++98

```
remove_copy_if (a,
                a + 5,
                ostream_iterator<int> (cout, " "),
                bind2nd(less<int>(), 3));
```

Связыватели в C++98

Но вот так не работает:

```
class MyCmp {
  public:
    bool operator()(int a, int b) const {
        return a > b;
    }
};

remove_copy_if (a.begin(),
                a.end(),
                b.begin(),
                std::bind2nd(MyCmp(), 3));
```

Связыватели в C++98

А так ok:

```
class MyCmp : public std::binary_function<int, int, bool>{
  public:
    bool operator()(int a, int b) const {
        return a > b;
    }
};

remove_copy_if (a.begin(),
                a.end(),
                b.begin(),
                std::bind2nd(MyCmp(), 3));
```