~~Обобщенное программирование в C++~~

Как сделать свою жизнь проще через страдания

Гомон Сергей, Regula (Siarhei.Homan@regula.by)

Обо мне

Основные сферы интересов:

- Защита информации
- Сетевая разработка
- Обработка изображений

Siarhei.Homan@regula.by

Содержание

Понятие обобщенного программирования
Правила вывода шаблонных типов C++
Основные определения и терминология
Примеры
Ссылки

Что такое обобщенное программирование

Обобщенное программирование

Шаблоны C++ генерируют программы

Обобщенная программа

+

Конкретные типы и значения

=

Обычные программы

STL значит STepanov and Lee

Александр Степанов, автор STL

Базовый синтаксис

```
template<typename ArrayType, int ArraySize>
class FixedSizeArray
{
public:
    FixedSizeArray() { std::fill(std::begin(arr_), std::end(arr_), 0); }
    ArrayType& at(int idx) { return arr_[idx]; }
    const ArrayType& at(int idx) const { return arr_[idx]; }
private:
    ArrayType arr_[ArraySize];
};
```

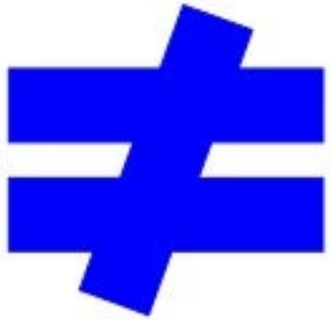
```
FixedSizeArray<int, 5> array;
array.at(0) = 5;
int first_elem = array.at(0);
```

Шаблоны в Си

```
#define VECTOR(ArrayType, ArraySize, TypeName) \
  class FixedSizeArray##TypeName { \
  public: \
    FixedSizeArray##TypeName() { std::fill(std::begin(arr_), std::end(arr_), 0); } \
    ArrayType& at(int idx) { return arr_[idx]; } \
    const ArrayType& at(int idx) const   { return arr_[idx]; } \
  private: \
    ArrayType arr_[ArraySize]; \
  };
  VECTOR(int, 5, Int5)

FixedSizeArrayInt5 array;
array.at(0) = 5;
int first_elem = array.at(0);
```

Шаблоны в других языках



C++

Правила вывода типов

Вывод типов

```
template<typename T>
void f(ParamType param);
f(expr);
```

```
// Example:
template<typename T>
void f(const T& param);

int x = 0;
f(x); // T is int, ParamType is const int&
```

PyramType - не ссылка и не указатель

```
template<typename T>
void f(T param);

int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // T is int, ParamType is int
f(cx); // T is int, ParamType is int
f(rx); // T is int, ParamType is int
```

ParamType - ссылка

```
template<typename T>
void f(T& param); // param is a reference

int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // T is int, ParamType is int&
f(cx); // T is const int, ParamType is const int&
f(rx); // T is const int, ParamType is const int&
```

ParamType - константная ссылка

```
template<typename T>
void f(const T& param);

int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // T is int, ParamType is const int&
f(cx); // T is int, ParamType is const int&
f(rx); // T is int, ParamType is const int&
```

ParamType - указатель

```
template<typename T>
void f(T* param);

int x = 27;
const int *px = &x;

f(&x); // T is int, ParamType is int*
f(px); // T is const int, ParamType const int*
```

ParamType - универсальная ссылка

```
template<typename T>
void f(T&& param); // param is now a universal reference

int x = 27;
const int cx = x;
const int& rx = x;

// l-value examples
f(x);  // T is int&, ParamType is int&
f(cx); // T is const int&, ParamType is const int&
f(rx); // T is const int&, ParamType is const int&

// r-value example
f(27); // T is int, ParamType is int&&
```

ParamType - массив

```
const char name[] = "J. P. Briggs";

template<typename T>
void f(T param);
f(name); // T is const char*, ParamType is const char*

template<typename T>
void f(T& param);
f(name); // T is const char [13], ParamType is const char (&)[13]


// Compile time array size
template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept {
        return N;
}
```

ParamType - функция

```
void someFunc(int, double);
template<typename T>
void f1(T param);

template<typename T>
void f2(T& param);

f1(someFunc); // T is void (*)(int, double), ParamType is void (*)(int, double)
f2(someFunc); // T is void (&)(int, double), ParamType is void (&)(int, double)
```

СХЕМА КОТА


махало-топырило
навострилы
смекало
топорщилы
душещипалы
нюхало
вздыбило
урчало
пукало-какало
умывало
мурчало-мявкало
набивало
приседалы-почесалы
точилы-царапалы

Основные определения

Шаблонная функция
(function template)

Функция шаблона
(template function)

```
template class std::array<int, 5>; // Explicit instantiation

std::array<int, 5> arr1; // Explicit instantiation used
std::array<int, 6> arr2; // Implicit instantiation used
```

Концепты

```
template<typename T>
class HasClone {
public:
          static void Constraints() {
          typedef T* (T::*CloneFunc)() const;
          CloneFunc hasCloneFunc = &T::Clone;
          }
          HasClone() { void (*p)() = Constraints; }
};

template<typename T>
class C : HasClone<T> {
          // ...
};
```

Специализация

```
template<>
class A<int> { /*...*/ }; // Specialization
```

```
template<typename S, typename U>
class A<std::map<S, U>> { /*...*/ }; // partial specialization
```

Характеристики (traits)

```
template <class T, T val>
struct integral_constant {
   typedef integral_constant<T, val> type;
   typedef T value_type;
   static const T value = val;
};

typedef integral_constant<bool, true>  true_type;
typedef integral_constant<bool, false> false_type;
```

Характеристики (traits) (2)

```
template <typename T> struct is_void : public false_type {};
template <> struct is_void<void> :     public true_type {};

std::cout << is_void<void>::value << std::endl;  // true
std::cout << is_void<int >::value << std::endl;  // false
```

```
template <typename T> struct is_pointer :          public false_type{};
template <typename T> struct is_pointer<T*> : public true_type{};

std::cout << is_pointer<int*>::value << std::endl;  // true
std::cout << is_pointer<int >::value << std::endl;  // false
```

Variadic Templates

Разворачивание шаблоно! может иметь место только в определенных контекстах:

- Список аргументов функции: `f(&args...);`
- Список аргументов шаблона: `container<A,B,C...> t1;`
- Список параметров функции: `template<typename ...Ts> void f(Ts...) {}`
- Список параметров шаблона: `template<T... Values>`
- Базовые классы и список инициализации класса: `class X : public Mixins...`
- Список инициализации: `int res[sizeof...(args) + 2] = {1,args...,2};`

Примеры

Факториал времени компиляции

```
template<int n>
struct Factorial {
        static const int f = Factorial<n-1>::f * n;
};

template<>
struct Factorial<0> {
        static const int f = 1;
};



std::cout << Factorial<5>::f << std::endl; // 120
```

Обфрускация строк времени компиляции

```
template<std::size_t SIZE>
struct hiddenString {
          short s[SIZE];
          constexpr hiddenString(): s{0}  {}

          std::string decode() const {
             std::string rv;
             rv.reserve(SIZE + 1);
             std::transform(s, s + SIZE - 1,
                       std::back_inserter(rv),
                       [](auto ch) { return ch - 1; });
             return rv;
          }
};
```

Обфрускация строк времени компиляции (2)

```
template<typename T, size_t N>
constexpr std::size_t sizeCalculate(const T(&)[N]) { return N; }

template<std::size_t SIZE>
constexpr auto encoder(const char str[SIZE]) {
          hiddenString<SIZE> encoded;
          for(std::size_t i = 0; i < SIZE - 1; i++)
             encoded.s[i] = str[i] + 1;
          encoded.s[SIZE - 1] = 0;
          return encoded;
}
#define CRYPTEDSTRING(name, x) \
          constexpr auto name = encoder<sizeCalculate(x)>(x)
```

Обфрускация строк времени компиляции (3)

```
CRYPTEDSTRING(str, "Big big secret!");
std::cout << str.decode() << std::endl;
```

Tuple

```
template<typename... Args> struct tuple;

template<typename Head, typename... Tail>
struct tuple<Head, Tail...> : tuple<Tail...> {
        tuple(Head h, Tail... tail) : tuple<Tail...>(tail...), head_(h) {}

        typedef tuple<Tail...> base_type;
        typedef Head value_type;

        base_type& base = static_cast<base_type&>(*this);
        Head head_;
};

template<> struct tuple<> {};

tuple<int, double, int> t(12, 2.34, 89);
std::cout << t.head_ << " " << t.base.head_ << " " << t.base.base.head_ <<
std::endl;
```

Функции для Tuple

```
template<int I, typename Head, typename... Args>
struct getter {
        typedef typename getter<I-1, Args...>::return_type return_type;
        static return_type get(tuple<Head, Args...> t) {
           return getter<I-1, Args...>::get(t);
        }
};

template<typename Head, typename... Args>
struct getter<0, Head, Args...> {
        typedef typename tuple<Head, Args...>::value_type return_type;
        static return_type get(tuple<Head, Args...> t) {
           return t.head_;
        }
};
```

Функции для Tuple (2)

```
template<int I, typename Head, typename... Args>
typename getter<I, Head, Args...>::return_type get(tuple<Head, Args...> t) {
        return getter<I, Head, Args...>::get(t);
}

test::tuple<int, double, int> t(12, 2.34, 89);
std::cout << get<0>(t) << " " << get<1>(t) << “ “ << get<2>(t) << std::endl;
```

Применение функции к Tuple

```
template<typename F, typename Tuple, int... N>
auto call(F f, Tuple &&t) {
        return f(std::get<N>(t)...);
}
```

```
int sum(int a, int b, int c) {
        return a + b + c;
}
td::tuple<int, int, int> args(1, 2, 3);

call<int(&)(int,int,int), std::tuple<int,int,int>&, 0, 1, 2>(sum, args);
```

Применение функции к Tuple (2)

```
template<typename F, typename Tuple, bool Enough, int TotalArgs, int... N>
struct call_impl {
        auto static call(F f, Tuple&& t) {
            return call_impl<F, Tuple, TotalArgs == 1 + sizeof...(N),
                                       TotalArgs, N..., sizeof...(N)
                        >::call(f, std::forward<Tuple>(t));
        }
};
```

Применение функции к Tuple (3)

```
template<typename F, typename Tuple, int TotalArgs, int... N>
struct call_impl<F, Tuple, true, TotalArgs, N...> {
          auto static call(F f, Tuple&& t) {
              return f(std::get<N>(std::forward<Tuple>(t))...);
          }
};
```

Применение функции к Tuple (4)

```
template<typename F, typename Tuple>
auto call(F f, Tuple&& t) {
          typedef typename std::decay<Tuple>::type type;
          return call_impl<F, Tuple, 0 == std::tuple_size<type>::value,
                                     std::tuple_size<type>::value
                    >::call(f, std::forward<Tuple>(t));
}


call(sum, args);
```

PRC на основе boost.asio и boost.serialization

https://github.com/gomons/CppRpcLight

Service.h

```
//RPC_DEFINE(funcname, returnType , args...);
  RPC_DEFINE(sum     , int        , int, int);
  RPC_DEFINE(echo    , std::string, std::string);
```

Service.cpp (Выполняется на сервере)

```
RPC_DECLARE(sum, int, int a, int b) {
        return a + b;
}

RPC_DECLARE(echo, std::string, std::string str) {
        return str;
}
```

Запуск сервера

```
boost::asio::io_service io_service;
RpcServer prc_server(io_service);
io_service.run();
```

Запуск клиента

```
boost::asio::io_service io_service;
cpp_rpc_light::ClientConnection client_connection(io_service);
std::thread thread([&io_service]() {
   io_service.run();
});
client_connection.WaitForConnect();
```

Вызов удаленной функции

```
auto sum_res = sum(client_connection, 5, 6);
auto sum_future = sum_async(client_connection, 5, 6);
auto sum_res2 = sum_future.get();

auto echo_res = echo(client_connection, std::string("Ping!"));
auto echo_future = echo_async(client_connection, std::string("Ping!"));
auto echo_res2 = echo_future.get();
```

Немного о страданиях

Разные компиляторы поддерживают немного разный синтаксис

Трудно найти проблему по диагностическому сообщению

Только новые компиляторы

Сложно писать, поддерживать, разбираться

Долгая компиляция, большой размер бинарника

Код шаблонов должне находиться в заголовочном файле

Спасибо за внимание!

Ссылки

Scott Meyers - Effective Modern C++

Stroustrup - A Tour of C++ - 2013

Sutter, Alexandrescu - C++ Coding Standart

https://m.habrahabr.ru/post/54762/

https://habrahabr.ru/post/166849/

https://habrahabr.ru/post/228031/

https://habrahabr.ru/post/245719/