C++14 консоль для сетевого оборудования

Денис Панин, Старший разработчик @ RDP.ru

Наша старая консоль

- 1 функция, принимающая std::string
- 20 тысяч строк лапши из конвертирования, форматирования и выбора коллбэка
- Рекурсивный вызов самой себя
- Куча внутренних магических переменных
- С командами из нескольких слов – просто ужас.

Наша старая консоль

- 1 функция, принимающая std::string
- 20 тысяч строк лапши из конвертирования, форматирования и выбора коллбэка
- Рекурсивный вызов самой себя
- Куча внутренних магических переменных
- С командами из нескольких слов – просто ужас.

- Но все идеально работает и переписывать это никто не будет.

Run-time подход

```
std::unordered_map<std::string, function_or_recursive_map>

void callback(const std::vector<std::string>& args) {
    // проверка размера аргументов
    // конверсия строк в нужный тип и проверка корректности
}
```

- Легко ошибиться в куче copy-paste
- Сигнатура функции ничего нам не говорит об аргументах

Что хотелось бы?

- Говорящая сигнатура коллбэка -> void set_ipv4(IPv4 new_ip);
- Автоматическая проверка количества элементов
- Автоматическое преобразование строковых аргументов в нужные типы
- Перегрузка команд (по кол-ву и конвертируемости аргументов)
- Поддержка многословных команд -> “set log all fatal”

Что хотелось бы?

- Говорящая сигнатура коллбэка -> void set_ipv4(IPv4 new_ip);
- Автоматическая проверка количества элементов
- Автоматическое преобразование строковых аргументов в нужные типы
- Перегрузка команд (по кол-ву аргументов и конвертируемости)
- Поддержка многословных команд -> “set log all fatal”

- **Добавление новой команды 1 строкой**

std::tuple

```
using T = tuple<tuple<int, double>, float>;
T nested = make_tuple(make_tuple(1, 2.0), 3.f);

double TWO = get<1>(get<0>(nested));

template <typename...ARGS>
tuple<ARGS...> make_tuple(ARGS&&...args){
     return tuple<ARGS...>{ forward<ARGS>(args)... };
}
```

Итерация по std::tuple

```
template <typename FN, typename ... ARGS>
void ForEach(tuple<ARGS...>& tpl, bool& b, FN& func)
{
  ForEachIter_t<0, sizeof...(ARGS)>::
    Do(tpl, b, func);
}
```

Так же версия без “bool b”, где не нужно прерывание обхода.

Итерация по std::tuple

```
template <size_t CUR, size_t END>
struct ForEachIter_t {
	template <typename FN, typename...TYPES>
	static void Do(tuple<TYPES...>& tpl, bool& b, FN&& func) {
		b = func(get<CUR>(tpl));
		if (!b) return;
		ForEachIter_t<CUR + 1, END>::Do(tpl, func);
	}
};
template<size_t CUR>
struct ForEachIter_t<CUR, CUR> { … };
```

Итерация по std::tuple

```
auto tpl = make_tuple(1, 2, 3, 4, 5);
bool found = false;


ForEach(tpl, found, [](auto elem) {
     cout << elem;
     return elem != 4;
});
// 1 2 3, found == false
```

std::apply

```
template <typename Func, typename Tuple>
decltype(auto) apply(Func&& fn, Tuple&& tpl) {
    return applyHelper(forward<Func>(fn), forward<Tuple>(tpl),
                make_index_sequence<tuple_size<Tuple>::value>);
}

template <typename Func, typename Tuple, size_t... I>
decltype(auto) applyHelper(Func&& fn, Tuple&& tpl, index_sequence<I...>){
    return fn(move(get<I>(tpl))...);
}
```

std::apply

void func(int a1, const char* a2, float a3) {};

auto tpl = make_tuple(123, “abc”, 4.5f);

std::apply(func, tpl) превращается в:

func(get<0>(tpl), get<1>(tpl), get<2>(tpl));

Совпадение по типам и количеству аргументов проверяется в компайл-тайме.

Класс Command_t

- Мы просто храним функцию, ее имя… и типы всех аргументов!

```
template <typename RET, typename … ARGS>
struct Command_t {
  Command_t(string&& name, function<RET(ARGS…)>&& func);
  string& GetName();
  function<RET(ARGS…)>& GetFunc();
}

template <typename RET, typename … ARGS>
auto MakeCommand(string&& name, function<RET(ARGS…)>&& func);
```

Дерево команд

```
void sum(int a, int b) { /* вывод суммы a + b */ }
void reverse(string&& str) { /*переворот & вывод*/ }

auto commands = std::make_tuple(
        MakeCommand(“sum”, sum),
        MakeCommand(“reverse”, reverse)
);
```

Пример работы

```
# dummycmd
Command ‘dummycmd’ is not found
# sum 1 2
1 + 2 = 3
# sum 1
Incorrect argument count
# sum 2 TWO
Unable to convert arguments
# reverse C++
++C
```

Конвертер

```
template <typename T>
T Converter(std::string&&) {
    static_assert(false, “Converter is not specialized!”);
}
template <>
int Converter<int>(std::string&& arg){
    return convert_to_int_or_throw(arg);
}
// И куча еще всяких разных специализаций!
```

Начинаем обходить дерево команд

```
void StartLookup(tuple<ARGS...>& tpl, vector<string>& tokens)
{
  if (tokens.empty()) return;
  bool found = false;
  ForEach(tpl, found, [&] (auto&& elem) {
    return ProcessElem(tokens.begin(), tokens.end(), elem);
  }
  if (!found) Throw(“Command not found!”);
}
```

И процессим команды

```
template <typename RET, typename ... ARGS>
bool ProcessElem(It cur, It end, Command_t<RET(ARGS...)>& cmd) {
  if (*cur != cmd.GetName()) {
    if (distance(cur, end) != sizeof...(ARGS) + 1)
      Throw(“Incorrect argument count”);
    Invoke(cmd.GetFunc(), cur);
    return true;
  }
  return false;
}
```

Вызов callback’а

```
template <typename RET, typename ... ARGS>
void Invoke(function<RET(ARGS...)>& func, It cur) {
  tuple<ARGS...> tpl;
  ForEach(tpl, [&] (auto& elem) {
    using TP = decay_t<decltype(elem)>;
    elem = Converter<TP>(move(*++cur));
  }
  apply(func, move(tpl));
}
```

Итого

1. Прошлись по tuple из команд, нашли функцию
   - Если не смогли, то упали.
2. Преобразовали arg1 и arg2 в параметры из сигнатуры функции
   - Если не смогли, то упали.
3. Вызвали найденную функцию при помощи tuple из сконвертированных аргументов!

Результат

- ✓ Нормальный вид коллбэка
- ✓ Добавление команды 1 строчкой
- ✓ Автоматическая проверка кол-ва аргументов
- ✓ Автоматическая конвертация

Но хочется больше!

Команды из нескольких слов

```
template <typename ... ARGS>
struct Branch_t {
    Branch_t(string&& name, ARGS&&... cmds);
    std::string& GetName();
    tuple<ARGS...>& GetChildren();
}
```

И аналогичный MakeBranch.

Дерево для команд из нескольких слов

```
auto commands = make_tuple(
  MakeBranch(“nested”,
    MakeBranch(“math”,
      MakeCommand(“sum”, sum)
    ) ) );
```

```
# nested math sum 1 2
3
# nested fake1 fake2
command ‘fake1’ is not found in branch ‘nested’
```

Как это сделано?

```
template <typename ... ARGS>
bool ProcessElem(It cur, It end, Branch_t<ARGS...>& br) {
  if (*cur != br.GetName()) {
    if (distance(cur, end) == 1) Throw(“Empty cmd”);
    bool found{ false };
    ForEach(br.GetChildren(), found, [&](auto& elem) {
      return ProcessElem(cur + 1, end, elem);
    }
    if (!found) Throw(“cmd not found in branch ”, *cur);
    return true;
  }
  return false; }
```

А если аргумент - массив элементов?

Сигнатура - void callback(vector<T>&& arg);

```
template<typename T>
struct is_vector : false_type {};

template <typename T>
struct is_vector<vector<T>> : true_type{};
```

А если аргумент – массив элементов?

```
template <typename T>
std::enable_if_t<is_vector<T>::value, T>
Converter(std::string&& str) {
  auto tokens = split(str, ‘,’);
  T ret;
  for (auto& elem : tokens)
    ret.push_back(Converter<T::value_type>(elem));
  return ret;
} // обычный converter получает enable_if для !is_vector
```

А если аргумент – массив элементов?

```
void accum(vector<int>&& v) {
     cout << accumulate(v.begin(), v.end(), 0);
}
```

```
# accumulate 1,2,3,4,5
15
```

Нам не пришлось писать специализацию на каждый вектор!

А если мы хотим перегрузку функций?

```
void sum2i(int a, int b)          { cout << “2i\n”; }
void sum2f(float a, float b)      { cout << “2f\n”; }
void sum3i(int a, int b, int c)   { cout << “3i\n”; }

auto commands = make_tuple(
  MakeCommand(“sum”, sum2i),
  MakeCommand(“sum”, sum2f),
  MakeCommand(“sum”, sum3i)
);
```

А если мы хотим перегрузку функций?

```
enum class FindStatus {
  Executed, ErrConvert, ErrArgCount, NotFound
}
```

ForEachFindStatus:

Прекращает работу, если Executed.

Возвращает самую “серьезную” ошибку.

А если мы хотим перегрузку функций?

```
template <typename RET, typename ... ARGS>
FindStatus ProcessElem(It cur, It end, Command_t<RET(ARGS...)>& cmd) {
  if (*cur != cmd.GetName()) {
    if (distance(cur, end) != sizeof...(ARGS) + 1)
      return FindStatus::ErrArgCount;
    try { Invoke(cmd.GetFunc(), cur); }
      catch(...) return FindStatus::ErrConvert;
    return FindStatus::Executed;
  }
  return FindStatus::NotFound;
}
```

А если мы хотим перегрузку функций?

```
FindStatus ProcessElem(It cur, It end, Branch_t<ARGS...>& branch) {
  if (*cur == branch.GetName()) {
    const auto lmb = [&] (const auto& elem) {
      return ProcessElem(cur + 1, last, elem);
    }
    const FindStatus fs = ForEachFindStatus(branch.GetChildren(), lmb);
    switch (fs) {
        // Либо возвращаем Executed, либо кидаем Exception
    }
  } else return FindStatus::NotFound;
}
```

А если мы хотим перегрузку функций?

```
# fn 1 2
2 ints
# fn 1.0 2.0 3.0
3 doubles
# fn 1 2.0
2 doubles, because “1” converts to double :D
# fn 2 TWO
Unable to convert arguments
# fn 1 2 3 4
Incorrect argument count
```

А мы хотим если дополнение и помощь?

```
bool std::equal(Iterator It1Beg, Iterator It1End, Iterator It2Beg);

template <typename T>
constexpr string NameOfType() {
      static_assert(false, “Same as in Converter ^_^”);
}

template <>
constexpr string NameOfType<int> {
      return “Int32”s;
}
```

А мы хотим если дополнение и помощь?

```
void StartHelp(tuple<ARGS...>& cmds, vector<string>&& tokens) {
	vector result;
	ForEach(cmds, [&](const auto& elem) {
		auto v = GenHelp(tokens.begin(), tokens.end(), "", elem);
		if (!v.empty()) {
			result.insert(vc.end(), v.begin(), v.end());
		}
	});
	// Вывод вектора VC
}
```

А мы хотим если дополнение и помощь?

GenHelp – аналог ProcessElem, так же обходим Branch и Command.

Где мы в каждой итерации сохраняем текущий поисковый путь, и в конце возвращаем вектор найденных строк, с путями и командами.

equal(cmdName.begin(), cmdName.begin() + minSize, currIt.begin());

“sum” и “s” - true

“sum” и “summa” - true

А мы хотим если дополнение и помощь?

Интересно преобразовывать типы в строки!

```
template <typename T>
string NameOfType() {
      static_assert(false, “Specialize me!”);
}

template <>
string NameOfType<>() {
      return “No arguments!”s;
}
```

А мы хотим если дополнение и помощь?

```
template <typename … ARGS>
string NameOfTypes() {
      string ret;
      ret += NameOfTypes_concat<decay_t<ARGS>…>();
      return ret;
}
template <typename T, typename … ARGS>
string NameOfTypes_concat() {
      return NameOfType<T>() + ‘ ‘ + NameOfTypes<ARGS…>();
}
// И тоже самое с 1 аргументом, но без рекурсии.
```

А мы хотим если дополнение и помощь?

```
# n m s?
nested math square | Func (Int32)
nested math sum | Func (Double Double)
n-test m-test s-1 | Func (IPv4)
n-test m-test s-2 | Func (IPv6)
# n m?
Nested math | Branch
n-test m-test | Branch
# fake?
<волшебное ничего>
```

А если мы хотим сохранять параметры?

Мы можем обрабатывать разные функции в зависимости от сигнатуры!

```
string GetIp();
void SetIp4(IPv4 ip);
void SetIp6(IPv6 ip);

auto commands = make_tuple(
       MakeCommand(“ip”, SetIp4),
       MakeCommand(“ip”, SetIp6),
       MaKeCommand(“ip”, GetIp)
);
```

А если мы хотим сохранять параметры?

```
void Dump(StrVec& vec, string path, const T&) {};

void Dump(StrVec& vec, string path, Branch_t<ARGS...>& br) {
	ForEach(br.GetChildren(), [&](auto& elem) {
		Dump(vec, path + br.GetName() + “ “, elem);
	});
}
void Dump(StrVec& vec, string path, Command_t<string()>&) {
	vec.emplace_back(path + cmd.GetName() + ' ' + cmd.GetFunc()());
}
```

А если мы хотим сохранять параметры?

```
void Dump(StrVec& vec, string path, const T&) {};

void Dump(StrVec& vec, string path, Branch_t<ARGS...>& br) {
	ForEach(br.GetChildren(), [&](auto& elem) {
		Dump(vec, path + br.GetName() + “ “, elem);
	});
}
void Dump(StrVec& vec, string path, Command_t<string()>&) {
	vec.emplace_back(path + cmd.GetName() + ' ' + cmd.GetFunc()());
}
```

А если мы хотим сохранять параметры?

```
auto tpl = make_tuple(
        MakeBranch(“param”,
                MakeCommand(“str”, func_that_returns_std_string_1),
                MakeCommand(“str”, func_that_accepts_std_string),
                MakeCommand(“int”, func_that_returns_std_string_2),
                MakeCommand(“int”, func_that_accepts_int)
        ) );
```

```
# dump
param str hello
param int 3
```

Как нам помогут новые стандарты?

- std::apply вместо костылей (C++17)

std::apply(FUNC&&, TUPLE&&);

вызвать FUNC с распакованными параметрами из TUPLE

Как нам помогут новые стандарты?

- std::apply вместо костылей (C++17)
- Class template argument deduction (C++17)

Было -> MakeBranch(“func”, func);

Стало -> Branch_t(“func”, func);

Как нам помогут новые стандарты?

- std::apply вместо костылей (C++17)
- Class template argument deduction (C++17)
- std::optional в конвертере вместо исключений (C++17)

std::optional<IPv4> Converter(string&&);

Как нам помогут новые стандарты?

- std::apply вместо костылей (C++17)
- Class template argument deduction (C++17)
- std::optional в конвертере вместо исключений (C++17)
- std::variant в конвертере, как аргумент (C++17)

std::variant<IPv4, IPv6, Domain> Converter(string&&);

И даже вектор их!

Как нам помогут новые стандарты?

- std::apply вместо костылей (C++17)
- Class template argument deduction (C++17)
- std::optional в конвертере вместо исключений (C++17)
- std::variant в конвертере, как аргумент (C++17)
- Familiar template syntax for generic lambdas (C++20)

```
auto lambda = []<typename T>(const T& arg)
typename T::nested_type tmp; // вместо decay_t<decltype(...)>
```

Чем в итоге это закончилось

- В старом проекте консоль осталась без изменений

Чем в итоге это закончилось

- В старом проекте консоль осталась без изменений
- В новый проект этот доделанный прототип был внедрен.

Чем в итоге это закончилось

- В старом проекте консоль осталась без изменений
- В новый проект этот доделанный прототип был внедрен.
- В связи с точечностью Converter’а, у нас стала гораздо лучше система типов:

Вместо IPv4 с методами IsRange(), HaveMask(), появились IPv4Range, IPv4wMask, etc…

Чем в итоге это закончилось

- В старом проекте консоль осталась без изменений
- В новый проект этот доделанный прототип был внедрен.
- В связи с точечностью Converter’а, у нас стала гораздо лучше система типов:

  Вместо IPv4 с методами IsRange(), HaveMask(), появились IPv4Range, IPv4wMask, etc…
- Огромный заряд мотивации у людей, работающих над этим проектом, когда они увидели, что мы можем использовать модерновый C++14 и метапрограммирование.

Чем в итоге это закончилось

- В старом проекте консоль осталась без изменений
- В новый проект этот доделанный прототип был внедрен.
- В связи с точечностью Converter’а, у нас стала гораздо лучше система типов:

  Вместо IPv4 с методами IsRange(), HaveMask(), появились IPv4Range, IPv4wMask, etc…
- Огромный заряд мотивации у людей, работающих над этим проектом, когда они увидели, что мы можем использовать модерновый C++14 и метапрограммирование.
- Появилась культура внутренних лекций, начатая именно лекцией с обьяснением этого материала.

The End!

- Посмотреть слайды и код можно тут:

https://github.com/Starl1ght/EcoConsoleTPL

- Написать мне о том, как можно сделать compile-time Trie:

starl1ght@mail.ru