Вариативные шаблоны

Практическое объектно-ориентированное программирование

16.10.2024

План

- Пачки параметров
- Свёртки
- Вариабельные концепты
- Семантика размещения

Как это делали в C

```
int sum_all(int nargs, ...) {
  va_list ap;
  int cnt = 0;
  va_start(ap, nargs);
  for (int i = 0; i < nargs; ++i)

  cnt += va_arg(ap, int);
  va_end(ap);
  return cnt;
}
#define NUMARGS(...) (sizeof((int[]){__VA_ARGS__})/sizeof(int))
#define DOSUM(...) sum_all(NUMARGS(__VA_ARGS__),
__VA_ARGS__)
```

Пачки параметров

• Пример вариабельно шаблонной функции.
template<typename ... Args> void f(Args ... args);
• Способы вызова:
f(); // OK, пачка не содержит аргументов
f(1); // OK, пачка содержит один аргумент: int
f(2, “Hello“); // OK, пачка состоит из: int, char[5]
• Использование sizeof...(Args) либо sizeof...(args) возвращает размер пачки в штуках.

Паттерны раскрытия

• Говорят, что пачка параметров "раскрывается"в теле функции или класса.

```
// → g(int x, double y);
template<typename ... Types> void g(Types ... args) {
  f(args...); // → f(x, y);
  f(&args...); // → f(&x, &y);
  f(h(args)...); // → f(h(x), h(y));
}
```

C++ style троеточия

- Вместо variadic функции у нас рекурсивный variadic template.

```
int sum_all() { return 0; }
template <typename Head, typename ... Tail>
T sum_all(Head&& arg, Tail&& ... args) {
  return arg + sum_all(args...);
}
```

- Увы в таком виде это не будет работать (почему?). Кто поправит?

Исправляем ситуацию

```
int sum_all() { return 0; }
```

- Первый способ починить.

```
template <typename Head, typename ... Tail>
Head sum_all(Head&& arg, Tail&& ... args) {
  return arg + sum_all(std::forward<Tail>(args)...);
}
```

- Второй способ починить.

```
template <typename Head, typename ... Tail>
auto sum_all(Head&& arg, Tail&& ... args) {
  return arg + sum_all(args...);
}
```

Возвращаемся к прозрачной оболочке

- Почти идеальная прозрачная оболочка для одного аргумента.

```
template<typename Fun, typename Arg>
decltype(auto) transparent(Fun fun, Arg&& arg) {
  return fun(std::forward<Arg>(arg));
}
```

- То же для произвольного количества аргументов.

```
template<typename Fun, typename... Args>
decltype(auto) transparent(Fun fun, Args&&... args) {
  return fun(std::forward<Args>(args)...);
}
```

- Изменение чисто техническое.

“Тривиальная“ задачка

Задача: раскрытие пачек

```
int f(int x)  return x;
int f(int x, int y, int z)  return x + y + z;
int f(int x, int y, int z, int v)  return x + y + z + v;

template <typename ... T>
int foo(T ... args)  return f(f(args...) + f(args)...);

template <typename ... T>
int bar(T ... args)  return f(f(args, args...)...);
foo(1, 2, 3); // → ?
bar(1, 2, 3); // → ?
```

Синтаксические странности

• Списки базовых классов и списки инициализации в конструкторах ведут к странным конструкциям если пачки оказываются пустыми.

```
template <typename ... Mixins>
class mixture : public Mixins ... {
// здесь тело для класса
public:
  mixture(Mixins... ms) : Mixins(ms)... {}
};
mixture<C1, C2> m (C1, C2); // m : C1, C2
mixture<> mnothing; // это ок, но выглядит нелепо
```

Свертка

Паттерн свёртки	Результирующее выражение
`... op pack`	`( ... (p1 op p2) op p3) ... op pN)`
`init op ... op pack`	`( ... (init op p1) op p2) ... op pN)`
`pack op ...`	`(p1 op (p2 op ( ... (pN-1 op pN) ...)`
`pack op ... op fini`	`(p1 op (p2 op ( ... (pN op fini) ...)`

```
template <typename ... Ts>
auto sum_all(Ts&& ... args) { return (forward<Ts>(args) + ...); }
template <typename ... T>
void print_all(T ... args) { (cout « ... « args) « std::endl; }
```

Проблемы print_all

```
template <typename ... T>
void print_all(T ... args) {
  (std::cout « ... « args) « std::endl;
}
```

- Очевидно, что print_all записанный как есть не вставляет между выводимыми числами пробельные символы.

```
print_all(1, 1.5, 3); // -> 11.53
```

- Как заставить его это сделать?

По одному: функтор AddSpace

- Можем обработать каждый аргумент независимо.

```
template <typename T>
class AddSpace {
  const T& ref;
public:
  AddSpace(const T& r): ref(r) {}
  std::ostream& operator«(std::ostream& os, AddSpace s) {
    return os « s.ref « ' ';
  }
};
template <typename ... T> void print_all(T ... args) {
  (std::cout « ... « AddSpace(args)) « std::endl;
};
```

Иногда проще без сверток: if constexpr

```
template <typename ... T>
void print_all(T && arg, T && ... args) {
  std::cout « arg « “ “;
  if constexpr(sizeof...(args) != 0)
    print_all(std::forward<T>(args)...);
}
```

Свертки с пустыми пачками

• Внезапно две эти свёртки с пустыми пачками сильно отличаются.

```
auto sum_all(int ... args) {
  return (... + args);
}
auto and_all(bool ... args) {
  return (... && args);
}
```

• Как вы думаете, во что вычислится:

```
auto x = sum_all();
auto y = and_all();
```

https://godbolt.org/z/z4h9E6ME9

Вариабельные концепты

Вернемся к суммированию

- Рассмотрим нашу функцию sum_all.

```
template <typename ... Ts>
auto sum_all(Ts&& ... args) {
  return (... + std::forward<Ts>(args));
}
```

- Кажется тут чего-то не хватает.
- Например мы хотели бы чтобы все приходящие типы были одинаковыми. Как мы это сделаем?

Концепты: синтаксис записи с пачками

• Базовый концепт для пачки параметров это свёртка.

```
template <typename ... Ts>
  requires (EqualityComparable<Ts> && ... && true)
void f(Ts ... ts);
```

• Шаблонный параметр сворачивает с true.

```
template <EqualityComparable ... Ts>
void f(Ts ... ts);
```

• Благодаря каррированию, это легко распространяется на более сложные концепты.

Другие варианты синтаксиса

- Если в концепте есть дополнительные аргументы...

```
template <typename ... Ts>
requires (ConvertibleTo<Ts, int> && ... && true)
void f(Ts ... ts);
```

- То именно каррирование спасает

```
template <ConvertibleTo<int> ... Ts> // ok
void f(Ts ... ts);
```

- Давайте это распространим на задачу суммирования.

Pack requirement для суммирования

- Так можно сделать, но выглядит если честно не очень

```
template <typename T, typename ... Ts>
auto sum_all(T&& arg, Ts&& ... args)
requires (std::is_same_v<T, Ts> && ... && true) {
  return std::forward<T>(arg) + (... + std::forward<Ts>(args));
}
```

- Может быть занесём внутрь?

Повесим простое ограничение

- Чуть симпатичней смотрится с одним простым ограничением.
- Как бы вы написали are_same_v? В стандарте такого нет.

```
template <typename ... Ts>
auto sum_all(Ts&& ... args) requires are_same_v<Ts...> {
  return (... + std::forward<Ts>(args));
}
```

- Какие ещё проблемы вы видите здесь даже если вы это написали?

Теперь хороший концепт

- Как бы вы написали first_arg_t? В стандарте такого нет.

```
template <typename ... Ts>
concept Addable = requires(Ts&&... args) {
  { (... + std::forward<Ts>(args)) } ->
    std::same_as<first_arg_t<Ts...> >;
  requires are_same_v<Ts...>;
  requires sizeof...(Ts) > 1;
};
```

- Можем ли мы упростить концепт?

Небольшая засада

- Давайте вообще обойдёмся без first_arg_t!

```
template <typename T, typename ... Ts>
concept Addable2 = requires(T&& arg, Ts&&... args) {
  { (std::forward<T>(arg) + ... + std::forward<Ts>(args)) } ->
std::same_as<T>;
  requires are_same_v<Ts...>;
  requires sizeof...(Ts) > 0;
};
```

- Будет ли это работать?

Опциональность сверток в шаблонах

• Свёртки очень важны для концептов так как у нас нет рекурсии и выхода.

```
template <typename T, typename ... Ts>
auto sum_all(T&& arg, Ts&& ... args) {
 if constexpr(sizeof...(args) != 0)
   return arg + sum_all(std::forward<Ts>(args)...); // OK
return 0;
}

template <typename ... Ts>
auto sum_all(Ts&& ... args) {
 return (std::forward<Ts>(args) + ...); // OK
}
```

Важность свёрток в концептах

• Свёртки очень важны для концептов так как у нас нет рекурсии и выхода.

```
template <typename T, typename ... Ts>
concept Addable = requires(T&& arg, Ts&& ... args) {
   arg + Addable(std::forward<Ts>(args)...); // FAIL
}

template <typename ... Ts>
concept Addable = requires(Ts&& ... args) {
  (std::forward<Ts>(args) + ...); // OK
}
```

Тяжёлые классы

• В мире C++ иногда встречаются тяжёлые объекты

```
class Heavy {
// детали реализации
public:
Heavy(int sz, int x, int s) { выделение кучи ресурсов }
Heavy(const Heavy& rhs) {
// выделение такой же кучи ресурсов на копию
}
Heavy(Heavy&& rhs) { довольно дорогое перемещение }
};
```

• Перемещение таких объектов может быть не дешевле копирования

Контейнеры тяжёлых классов

• Увы, иногда нужно хранить тяжёлые классы в контейнерах

```
template <typename T>
class Stack {
  struct StackNode {
    T elem; StackNode *next;
    StackNode(T e, StackNode *nxt) : elem(e), next(nxt) {}
  };
public:
  void push(const T& elem) { top_ = new StackNode(elem, top_); }
// .... и так далее ....
```

• Подумаем о следующем коде:

```
s.push(Heavy(100, 200, 300)); // всё очень плохо
```

Давайте посчитаем копирования

- Нам нужно просто поместить элемент в контейнер

```
s.push(Heavy(100, 200, 300)); // всё очень плохо
```

- Вместо этого происходит:
  - Создание
  - Копирование аргументом в push_back
  - Копирование для окончательного хранения в узел

- Даже если сделать перегрузку

```
void push(T&& elem) { top_ = new StackNode(move(elem), top_); }
```

- Мы всё равно попадаем на довольно дорогое перемещение

Основная идея

- Хорошее решение должно создавать объект прямо внутри стека
- Это называется размещением

```
struct StackNode {
  T elem;
  StackNode *next;
  StackNode(параметры конструктора, StackNode *nxt) :
   elem(параметры конструктора), next (nxt) {}
```

- Разумеется параметры могут быть любыми и в любом количестве

Небольшие проблемы размещения

Хорошее решение должно создавать объект прямо внутри стека

- Это называется размещением

```
struct StackNode {
  T elem;
  StackNode *next;
  template<typename ... U>
  StackNode(U ... cargs, StackNode *nxt) :
    elem(cargs ...), next (nxt) {}
```

- Кто-нибудь видит проблемы в этом коде?

Жадный матчинг

```
struct StackNode {
  T elem;
  StackNode *next;
  template<typename ... U>
  StackNode(U ... cargs, StackNode *nxt) :
    elem(cargs ...), next (nxt) {}
```

- Проблема в том, что так не будет работать вывод типов
- Пачка матчится жадно

Основная идея

- Хорошее решение должно создавать объект прямо внутри стека
- Это называется размещением

```
struct StackNode {
T elem;
StackNode *next;
template<typename ... U>
StackNode(StackNode *nxt, U ... cargs) :
elem(cargs ...), next (nxt) {}
```

- Так в целом будет работать. Кто-нибудь видит более мелкие проблемы в этом коде?

Не забываем форвардинг

- Хорошее решение должно создавать объект прямо внутри стека
- Это называется размещением

```
struct StackNode {
  T elem;
  StackNode *next;
  template<typename ... U>
  StackNode(StackNode *nxt, U&& ... cargs) :
    elem(std::forward<U>(cargs) ...), next (nxt) {}
```

- Так в целом будет работать. Разумеется, параметры конструктора лучше пробрасывать

Emplace

• Обычно метод контейнера, который размещает объект, а не пробрасывает его называют emplace

```
template <typename T>
class Stack {
// детали реализации
public:
  void push(const T& elem) { top_ = new StackNode (top_, elem); }
  template <typename U> void emplace(U&& ... args) {
    top_ = new StackNode(top_, forward<U>(args)...);
}
```

• В стандартной библиотеке размещение поддерживают все последовательные контейнеры

Давайте снова посчитаем операции

- Нам нужно просто поместить элемент в контейнер
s.emplace<Heavy>(100, 200, 300); // всё куда лучше

- Теперь происходит:
  - Перемещение параметров конструктора
  - Ещё одно перемещение параметров конструктора
  - Создание объекта по месту назначения