Что такое variadic templates и как их готовить

Аржанников Илья

Яндекс. Web-робот.

EkbCpp, Июнь 2014

Функции с переменным количеством аргументов

- `va_list`
- `printf, vprintf`

```
1  template <class TFuncSig> struct TFunctor;
2
3  template <class R, class Arg1>
4  struct TFunctor<R (Arg1)> {
5      R operator()(Arg1);
6  };
7
8  template <class R, class Arg1, class Arg2>
9  struct TFunctor<R (Arg1, Arg2)> {
10     R operator()(Arg1, Arg2);
11 };
12 /* ... */
13 template <class R, class Arg1, class Arg2, /*...*/class Arg100500>
14 struct TFunctor<R (Arg1, Arg2, /*...*/Arg100500)> {
15     R operator()(Arg1, Arg2, /*...*/Arg100500);
16 };
```

Вернуть много значений из функции

- Передача по ссылке или по указателю:

```
T1 Foo(Arg1 inArg1, OutT1& t1, OutT2* t2);
```

- Определение структуры для возврата:

```
struct FooReturn { T1 t1; T2 t2; T3 t3; };
struct BazReturn { T4 t4; T5 t5; T6 t6; };
FooReturn Foo(Arg1 inArg1);
BazReturn Baz(Arg2 inArg2);
```

Вернуть много значений из функции

- Передача по ссылке или по указателю:

```
T1 Foo(Arg1 inArg1, OutT1& t1, OutT2* t2);
```

- Определение структуры для возврата:

```
struct FooReturn { T1 t1; T2 t2; T3 t3; };
struct BazReturn { T4 t4; T5 t5; T6 t6; };
FooReturn Foo(Arg1 inArg1);
BazReturn Baz(Arg2 inArg2);
```

- `T&` — l-value references
- `T&&` — r-value references
- `T&&` — universal references

Виды ссылок

- `T&` — l-value references
- `T&&` — r-value references
- `T&&` — universal references

Виды ссылок

- `T&` — l-value references
- `T&&` — r-value references
- `T&&` — universal references

```
1  int main(int, char**) {
2      int&  lr = 10; // ERROR
3      const int& clr = 10; // OK
4      int&& rr = 10; // OK
5      std::cout << clr << std::endl;
6      std::cout << rr << std::endl;
7      clr = 20; // ERROR
8      rr = 20; // OK
9      std::cout << rr << std::endl;
10     return 0;
11 }
```

Если переменная или параметр объявлены как T&&, где T — **выводимый тип**, то это universal reference.

```
1  //Взято у Скота Мэйрса
2  //http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers
3  Widget&& var1 = someWidget; // R-Value reference
4
5  auto&& var2 = var1; // Universal reference
6
7  template<typename T>
8  void f(std::vector<T>&& param); // R-Value reference
9
10 template<typename T>
11 void f(T&& param); // Universal reference
```

Если переменная или параметр объявлены как T&&, где T — **выводимый тип**, то это universal reference.

```
1  //Взято у Скота Мэйрса
2  //http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers
3  Widget&& var1 = someWidget; // R-Value reference
4
5  auto&& var2 = var1; // Universal reference
6
7  template<typename T>
8  void f(std::vector<T>&& param); // R-Value reference
9
10 template<typename T>
11 void f(T&& param); // Universal reference
```

```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t );

template< class T >
T&& forward( typename std::remove_reference<T>::type& t );

template< class T >
T&& forward( typename std::remove_reference<T>::type&& t );
```

```
1  void Foo(int&& a) { std::cout << "Foo(int&&)" << std::endl; }
2  void Foo(const int& a) { std::cout << "Foo(const int&)" << std::endl; }
3
4  template <class I>
5  void Move(I&& a) {
6      Foo(std::move(a));
7  }
8
9  template <class I>
10 void Forward(I&& a) {
11     Foo(std::forward<I>(a));
12 }
13
14 template <class I>
15 void AsIs(I&& a) {
16     Foo(a);
17 }
```

```
1 int a = 20;
2 Move(a); // Foo(int&&);
3 Move(10); // Foo(int&&);
4
5 Forward(a); // Foo(const int&);
6 Forward(10); // Foo(int&&);
7
8 AsIs(a); // Foo(const int&);
9 AsIs(10); // Foo(const int&);
```

```
struct Type {
    Type(Type&& type)
        : member(std::move(type.member))
    { }

    OtherType member;
};
```

```
Type t1;
Type t2(std::move(t1));
// t1 — должен сохранить инварианты. Как? Другой вопрос.
```

```
struct Type {
    Type(Type&& type)
        : member(std::move(type.member))
    { }

    OtherType member;
};
```

```
Type t1;
Type t2(std::move(t1));
// t1 — должен сохранить инварианты. Как? Другой вопрос.
```

```
1  struct String {
2      size_t size;
3      char* buffer;
4
5      String(const char* str) : size(strlen(str)), buffer(new char[size]) {
6          memcpy(buffer, str, size);
7      }
8      ~String() {
9          delete[] buffer;
10     }
11     String(const String& str); // Честно реализованы
12     String& operator=(const String& str);
13 };
```

```
String::String(String&& str); // Как?
```

```
String::String(String&& str) : size(str.size), buffer(str.buffer) {}
```

```
String::String(String&& str) : size(str.size), buffer(str.buffer) {
    std.size = 0; str.buffer = nullptr;
}
```

```
String::String(String&& str); // Как?
```

```
String::String(String&& str) : size(str.size), buffer(str.buffer) {}
```

```
String::String(String&& str) : size(str.size), buffer(str.buffer) {
    std.size = 0; str.buffer = nullptr;
}
```

```
String::String(String&& str); // Как?
```

```
String::String(String&& str) : size(str.size), buffer(str.buffer) {}
```

```
String::String(String&& str) : size(str.size), buffer(str.buffer) {
    std.size = 0; str.buffer = nullptr;
}
```

- Класс или структура:

```
1 template <class... ManyTemplatesArgs>
2 struct SomeClassWithManyTemplatesArgs { };
```

- Функции тоже могут в variadic templates:

```
1 template <class... SoManySoDifferent>
2 int Foo(SoManySoDifferent&&... args) {
3     return 42;
4 }
```

- Класс или структура:

```
template <class... ManyTemplatesArgs>
struct SomeClassWithManyTemplatesArgs { };
```

- Функции тоже могут в variadic templates:

```
template <class... SoManySoDifferent>
int Foo(SoManySoDifferent&&... args) {
    return 42;
}
```

- Класс или структура:

```
1 template <size_t... Numbers>
2 struct SomeClassWithManyTemplatesArgs { };
```

- Функции:

```
1 template <char... Symbols>
2 int Foo(SoManySoDifferent&&... args);
```

Может содержать любое количество аргументов, в том числе и **0**.

```
type... Args
class... Args
template <parameter-list> class... Args
```

```
1 template <class... Types>
2 void Foo(Types... args);
3
4 //...
5
6 Foo();          //OK: нет аргументов
7 Foo(1);         //OK: один аргумент типа int
8 Foo(1, 3.0);    //OK: два аргумента: int и double
9
```

Объявление с другими параметрами

- **Parameter pack** — последний аргумент.

```
1 template <class Head, class... Tail>
2 struct CoolClass { };
```

- Если только там нет еще одного **Parameter pack**

```
1 template <class... Type, size_t... Numbers>
2 struct CoolClass { };
```

Объявление с другими параметрами

- **Parameter pack** — последний аргумент.

```
1 template <class Head, class... Tail>
2 struct CoolClass { };
```

- Если только там нет еще одного **Parameter pack**

```
1 template <class... Type, size_t... Numbers>
2 struct CoolClass { };
```

sizeof...(Parameters pack) — Возвращает количество параметров

```
1 template <class... Args>
2 int Foo(const Args&...) {
3     return sizeof...(Args);
4 }
```

sizeof...(Parameters pack) — Объект времени компиляции

```
1 template <class... Args>
2 struct SomeGreedyStruct {
3     static_assert(sizeof...(Args) >= 10, "Too few arguments");
4 };
```

```
1  template <class Func, class... Args>
2  typename std::result_of<Func(Args&&...)>::type SafeCall(Func func,
       Args&&... args) {
3      try {
4          return func(std::forward<Args>(args)...);
5      } catch (std::exception& e) {
6          std::cout << e.what() << std::endl;
7      }
8  }
9
10 int Sum(int a, int b) { return a + b; }
11 int Bar(const std::string&) {
12     throw std::invalid_argument("Bazzinga!!");
13 }
14
15 SafeCall(Sum, 10, 20);
16 SafeCall(Bar, "mama mila ramu"); // Bazzinga!!
17
```

```
1  SafeCall(Sum, 10, 20);
2
3  int SafeCall(int (*func)(int), int&& arg1, int&& arg2) {
4      try {
5          return func(std::forward<int>(arg1), std::forward<int>(arg2));
6      } catch (std::exception& e) {
7          std::cout << e.what() << std::endl;
8      }
9  }
10
```

Распаковка. Выдрать голову.

```
1  template <class Head>
2  void Foo(Head head) { // 1
3      DoSomethingWith(head);
4  }
5
6  template <class Head, class... Tail>
7  void Foo(Head head, Tail... tail) { // 2
8      DoSomethingWith(head);
9      Foo(tail...);
10 }
11
12 // ...
13
14 Foo();              // ERROR: нет подходящей функции
15 Foo(1);            // OK: будет вызвана Foo из 1 с Head == int
16 Foo(1, 2.0, 3.0f); // OK: будет вызвана Foo из 2 с Head == int и Tail ==
       <double, float>
17
```

Распаковка. Expression expected.

```
template <class... Types>
void Foo(Types... args) {
    DSResult r = (DoSomethingWith(args)...); // ERROR
}
//...
Foo(1, 2.3, 4.5f);
```

```
void Foo(int i, double d, float f) {
    DSResult r = (DoSomethingWith(i), DoSomethingWith(d),
    DoSomethingWith(f));
}
//...
Foo(1, 2.3, 4.5f);
```

```
Foo();
```

Распаковка. Expression expected.

```
template <class... Types>
void Foo(Types... args) {
    DSResult r = (DoSomethingWith(args)...); // ERROR
}
//...
Foo(1, 2.3, 4.5f);
```

```
void Foo(int i, double d, float f) {
    DSResult r = (DoSomethingWith(i), DoSomethingWith(d),
    DoSomethingWith(f));
}
//...
Foo(1, 2.3, 4.5f);
```

```
Foo();
```

Распаковка. Expression expected.

```
template <class... Types>
void Foo(Types... args) {
    DSResult r = (DoSomethingWith(args)...); // ERROR
}
//...
Foo(1, 2.3, 4.5f);
```

```
void Foo(int i, double d, float f) {
    DSResult r = (DoSomethingWith(i), DoSomethingWith(d),
    DoSomethingWith(f));
}
//...
Foo(1, 2.3, 4.5f);
```

```
Foo();
```

```
1  template <class T>
2  void Out(T t) { std::cout << t << std::endl; }
3
4  template <class H, class... T>
5  void Out(H h, T... t) { std::cout << h << ' '; Out(t...); }
6
7  template <class... Args>
8  void Plus1(Args... args) { Out(++args...); }
9
10 Plus1(1, 2.3, 3.5f); // 2 3.3 4.5
11
12 template <class To, class... Args>
13 void CastTo(Args... args) { Out(static_cast<To>(args)...); }
14
15 CastTo<int>(1, 2.3, 3.5f); // 1 2 3
16
```

```
1 template <class T>
2 std::string ToString(T t);
3
4 template <class... Args>
5 std::vector<std::string> ToStrings(Args... args) {
6     std::vector<std::string> result = {ToString(args)...};
7     return result;
8 }
9
```

```cpp
template <class T> void Out(T t) { std::cout << t << std::endl; }
template <class H, class... T> void Out(H h, T... t) { std::cout << t <<
    ' '; Out(t...); }
//...
std::string someHavyString1;
std::string someHavyString2;
std::string someHavyString3;
Out(someHavyString1, someHavyString2, someHavyString3);
```

```cpp
template <class T> void Out(const T& t) { std::cout << t << std::endl; }
template <class H, class... T> void Out(const H& h, const T&... t) {
    std::cout << t << ' '; Out(t...); }
//...
std::string someHavyString1;
std::string someHavyString2;
std::string someHavyString3;
Out(someHavyString1, someHavyString2, someHavyString3);
```

```
template <class T> void Out(T t) { std::cout << t << std::endl; }
template <class H, class... T> void Out(H h, T... t) { std::cout << t <<
    ' ';  Out(t...); }
//...
std::string someHavyString1;
std::string someHavyString2;
std::string someHavyString3;
Out(someHavyString1, someHavyString2, someHavyString3);
```

```
template <class T> void Out(const T& t) { std::cout << t << std::endl; }
template <class H, class... T> void Out(const H& h, const T&... t) {
    std::cout << t << ' ';  Out(t...); }
//...
std::string someHavyString1;
std::string someHavyString2;
std::string someHavyString3;
Out(someHavyString1, someHavyString2, someHavyString3);
```

```
1 template <class... Mixins>
2 struct SuperClass: public Mixins... {
3     SuperClass(const Mixins&... m) : Mixins(m)... {}
4 };
5
```

```
1 template <class... T> class std::tuple;
2
3 template< std::size_t I, class... Types >
4 typename std::tuple_element<I, tuple<Types...> >::type& get(
      tuple<Types...>& t );
5
6 template< class... Types >
7 tuple<Types&...> tie( Types&... args );
8
```

```
1  //http://en.cppreference.com/w/cpp/utility/tuple
2  #include <tuple>
3  #include <iostream>
4  #include <string>
5  #include <stdexcept>
6
7  std::tuple<double, char, std::string> get_student(int id)
8  {
9      if (id == 0) return std::make_tuple(3.8, 'A', "Lisa Simpson");
10     if (id == 1) return std::make_tuple(2.9, 'C', "Milhouse Van Houten");
11     if (id == 2) return std::make_tuple(1.7, 'D', "Ralph Wiggum");
12     throw std::invalid_argument("id");
13 }
14
```

```
int main()
{
    auto student0 = get_student(0);
    std::cout << "ID: 0, " << "GPA: " << std::get<0>(student0) << ", "
              << "grade: " << std::get<1>(student0) << ", "
              << "name: " << std::get<2>(student0) << '\n';

    double gpa1;
    char grade1;
    std::string name1;
    std::tie(gpa1, grade1, name1) = get_student(1);
    std::cout << "ID: 1, "
              << "GPA: " << gpa1 << ", "
              << "grade: " << grade1 << ", "
              << "name: " << name1 << '\n';
}
```

```
template <class... Args>
iterator std::vector::emplace(const_iterator pos, Args&&... args);

template <class... Args>
std::pair<iterator, bool> std::map::emplace(Args&&... args);
```

printf(const char* format, ...)

d,i,o,u,x,X,e,E,f,F,g,G,a,A,c,s,p,n — Модификаторы типов

hh,h,l,ll,L,q,j,z,t — Модификаторы длины

- Плюсы:
  - Выводить стандартные типы
  - В произвольной последовательности
- Минусы:
  - Ничего не знает о пользовательских типах
  - Опасен для стека

Чего хотим получить:

- Выводить стандартные и пользовательские типы
- В произвольной последовательности
- Убрать модификаторы типов и длин

```
template <class... Args>
std::ostream& Format(std::ostream& out,
                     const std::string& format,
                     const Args&... args);
//format:
//   symbol != '%' — выводим symbol
//   %\d+ — выводим аргумент за номером \d+ ( считаем с нуля)
//   %% — выводим %
//   Format(std::cout, "mama mila ramu %0 raz", 7); // mama mila ramu 7
     raz
```

Чего хотим получить:

- Выводить стандартные и пользовательские типы
- В произвольной последовательности
- Убрать модификаторы типов и длин

```
template <class... Args>
std::ostream& Format(std::ostream& out,
                     const std::string& format,
                     const Args&... args);
//format:
//   symbol != '%' – выводим symbol
//   %\d+ – выводим аргумент за номером \d+ ( считаем с нуля)
//   %% – выводим %
//   Format(std::cout, "mama mila ramu %0 raz", 7); // mama mila ramu 7
     raz
```

```
1 template <class It>
2 size_t ParseNumber(It* begin, It end) throw(std::invalid_argument);
```

Format. Первые шаги.

```
template <class... Args>
std::ostream& Format(std::ostream& out, const std::string& format,
                     const Args&... args) {
    auto it = format.cbegin();
    auto end = format.cend();
    for (; it != end; ++it) {
        if (*it != '%') { out.put(*it); continue; }
        ++it;
        if (it == end) {
            throw std::invalid_argument("bad format");
        }
        if (*it == '%') { out.put(*it); continue; }
        size_t argIndex = ParseNumber(&it, end);
        if (argIndex >= sizeof...(Args)) {
            throw std::invalid_argument("bad format");
        }
        //Тут надо вывести аргумент за индексом argIndex
    }
}
```

```
1 void OutArg(std::ostream& out, size_t n) {
2     throw std::runtime_error("Hm!");
3 }
4
5 template <class H, class... T>
6 void OutArg(std::ostream& out, size_t n, const H& h, const T&... t) {
7     if (n == 0) { out << h; return;}
8     OutArg(out, n - 1, t...);
9 }
```

```
1  template <class... Args>
2  std::ostream& Format(std::ostream& out, const std::string& format,
3                       const Args&... args) {
4      auto it = format.cbegin();
5      auto end = format.cend();
6      for (; it != end; ++it) {
7          if (*it != '%') { out.put(*it); continue; }
8          ++it;
9          if (it == end) {
10             throw std::invalid_argument("bad format");
11         }
12         if (*it == '%') { out.put(*it); continue; }
13         size_t argIndex = ParseNumber(&it, end);
14         if (argIndex >= sizeof...(Args)) {
15             throw std::invalid_argument("bad format");
16         }
17         OutArg(out, argIndex, args...);
18     }
19 }
```

```
1 struct Outer {
2     std::function<void(std::ostream&)> Out;
3
4     template <class T>
5     Outer(const T& t)
6         : Out([&t](std::ostream& out) {return out << t;})
7     {}
8 };
9
```

```
template <class... Args>
std::ostream& Format(std::ostream& out, const std::string& format,
                     const Args&... args) {
    Outer outers[] = { Outer(args)... };
    auto it = format.cbegin();
    auto end = format.cend();
    for (; it != end; ++it) {
        if (*it != '%') { out.put(*it); continue; }
        ++it;
        if (it == end) {
            throw std::invalid_argument("bad format");
        }
        if (*it == '%') { out.put(*it); continue; }
        size_t argIndex = ParseNumber(&it, end);
        if (argIndex >= sizeof...(Args)) {
            throw std::invalid_argument("bad format");
        }
        outers[argIndex].Out(out);
    }
}
```

Показали

```
time ./gcc_rec_format_O3 1000000
        1.27 real         1.27 user         0.00 sys
time ./gcc_outer_format_O3 1000000
        2.79 real         2.78 user         0.00 sys
time ./gcc_printf_O3 1000000
        1.16 real         1.16 user         0.00 sys
time ./clang_rec_format_O3 1000000
        2.73 real         2.72 user         0.00 sys
time ./clang_outer_format_O3 1000000
        2.60 real         2.60 user         0.00 sys
time ./clang_printf_O3 1000000
        1.11 real         1.11 user         0.00 sys
```

- Исходники презентации на GitHub
- xandox@yandex-team.ru
- cppreference.com

Вопросы?