Variadic templates

CODERS.SCHOOL

http://coders.school

- Kamil Szatkowski, kamil.szatkowski@nokia.com
- Łukasz Ziobroń, lukasz@coders.school

About authors

Kamil Szatkowski

- Work at Nokia:
  - C++ software engineer @ CCH
  - C++ software engineer @ LTE CPlane
  - RAIN Developer @ LTE Cplane
  - Code Reviewer
  - Code Mentor
- Trainer:
  - Practial Aspects Of Software Engineering
  - Nokia Academy
  - Internal Nokia trainings
- Occassional speaker:
  - Academic Championships in Team Programming
  - code::dive community
  - code::dive conference

Łukasz Ziobroń

- Work at Nokia:
  - C++ software engineer @ LTE Cplane
  - C++ software engineer @ LTE OAM
  - Python developer @ LTE LOM
  - Scrum Master
  - Code Reviewer
- Trainer:
  - Practial Aspects Of Software Engineering
  - Nokia Academy
  - Coders.school
  - Internal Nokia trainings
- Occassional speaker:
  - Academic Championships in Team Programming
  - code::dive community
  - code::dive conference

Variadic templates
Motivation

Variadic templates can be used to create template functions or classes which accept any numer of arguments of any type.

```
int main()
{
    printf("Hello %s, you are %d years old\n", "John", 25);
    printf("Just a text\n");
    return 0;
}
```

Variadic templates
Syntax

Templates with variable number of arguments (*variadic template*) use new syntax of parameter pack, that represents many or zero parameters of template.

```
template<class... Types>
class variadic_class
{
/*...*/
};

template<class... Types>
void variadic_foo(Types&&... args)
{
/*...*/
}

variadic_class<float, int, std::string> v;
variadic_class v{2.0, 5, "Hello"};  // automatic template type deduction for classes from C++17


variadic_foo(1, "", 2u);
```

Variadic templates
Unpacking function parameters

Unpacking group parameters uses new syntax of elipsis operator (...).

In case of function arguments it unpacks them in order given in template function call.

It is possible to call a function on a parameter pack. In such case given function will be called on every argument from a function call.

It is also possible to use recursion to unpack every single argument. It requires the variadic template Head/Tail and non-template function to be defined.

Variadic templates

Example

```
template<class... Types>
void variadic_foo(Types&&... args)          // variadic_foo(1, 3.5, "f");
{
   callable(args...);                       // callable(1, 3.5, "f");
}


template<class... Types>
void variadic_perfect_forwarding(Types&&... args)
{
   callable(std::forward<Types>(args)...);
}

void variadic_foo() {}

template<class Head, class... Tail>
void variadic_foo(Head const& head, Tail const&... tail)
{
   /*action on head*/
   variadic_foo(tail...);
}
```

Variadic templates
Unpacking template class parameters

Unpacking template class parameters looks the same as unpacking template function arguments but with use of template classes.

It is possible to unpack all types at once (e.g. in case of base class that is variadic template class) or using partial and full specializations.

Variadic templates
Example

```
template<int... Number>
struct Sum;

template<int Head, int... Tail>
struct Sum<Head, Tail...>
{
   const static int RESULT = Head + Sum<Tail...>::RESULT;
};

template<>
struct Sum<>
{
   const static int RESULT = 0;
}

Sum<1, 2, 3, 4, 5>::RESULT; // = 15
```

Variadic templates

Handling inheritance from variadic classes

```
template<class... Types>
struct Base
{};

template<class... Types>
struct Derived : Base<Types...>
{};
```

Variadic templates
sizeof... operator

sizeof... returns the number of parameters in parameter pack.

```
template<class... Types>
struct NumOfArguments
{
   const static unsigned NUMBER_OF_PARAMETERS = sizeof...(Types);
};
```

Variadic templates
Fold expressions (C++17)

Allows to write compact code with variadic templates without using explicit recursion.

```
template<typename... Args>
auto Sum(Args... args){
    return (0 + ... + args);
}


template<typename... Args>
bool f(Args... args) {
    return (true && ... && args); // OK
}

template<typename... Args>
bool f(Args... args) {
    return (args && ... && args); // error: both operands
                                  // contain unexpanded
                                  // parameter packs
}
```

Operator	Value when param pack is empty
*	1
+	int()
&	-1
|	int()
&&	true
||	false
,	void()

CODERS.SCHOOL

http://coders.school