Computer Science

Variadic Templates in C++11/C++14 An Introduction

CPPCon 2015

slides: http://wiki.hsr.ch/PeterSommerlad/

IFS INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad
Director IFS Institute for Software
Bellevue September 2015

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

C++ Varargs example

```
#include <ostream>
#include <cstdarg>

void printListOfInts(std::ostream& out, size_t n, ...) {
  va_list args;
  va_start(args, n);
  while (n--) {
    out << va_arg(args, int) << ", ";
  }
  va_end(args);
  out << '\n';
}
```

C/C++ Varargs sucks

- printf/scanf require it, therefore it is in the language
- error prone: printf(“%d”,3.14);
- unchecked by compiler
- difficult to apply yourself correctly
- only pass-by-value, no references
- need explicit/implicit specification of
  - number of arguments (count of % for printf)
  - type of arguments (all the same or specified)

Problem: varying number of arguments

```
println(out,1,2,3,"hello",' ',"world");
```

- some languages provide means to pass a variable number of arguments to a function
- The C solution uses void func(...) but is inherently not type safe, e.g., printf()
- In Java you specify one type for all arguments (or Object)
- In C++11 you use a type-safe "variadic template"

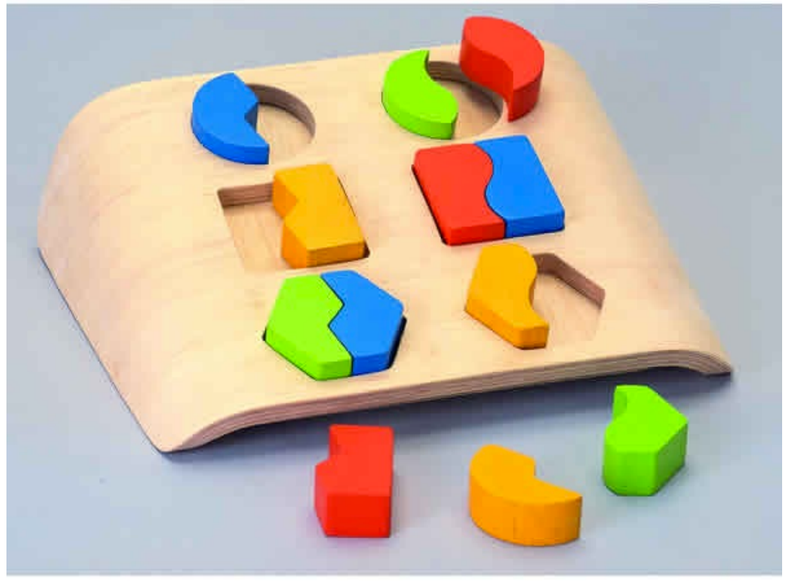
variadic templates pre C++11

- The Boost library contained infrastructure for features heavily relying on templates that can be used with any number of arguments. Examples are:
  - boost::bind() - function binder
  - boost::tuple<> - value holder
- Requires tedious repetition or macro magic (may be the main use case of Boost.Preprocessor)
  - and a built-in maximum

“pseudo” variadic template: Overloads

```
#ifndef PRINTLN_H_
#define PRINTLN_H_
#include <ostream>
inline
void printLn(std::ostream& out) {
   out.put('\n');
}
template<typename T1>
void printLn(std::ostream& out, T1 const & v1) {
   out << v1;
   out.put('\n');
}
template<typename T1,typename T2>
void printLn(std::ostream& out, T1 const & v1,T2 const & v2) {
   out << v1 << v2;
   out.put('\n');
}
template<typename T1,typename T2,typename T3>
void printLn(std::ostream& out, T1 const & v1,T2 const & v2,T3 const &v3) {
   out << v1 << v2 << v3;
   out.put('\n');
}
```

Reminder: functions, templates, patterns



- Function overload resolution
  - Pattern matching through call arguments
- Function template: template argument deduction from call args
- Class template: implementation selection through (partial) specialization
  - Pattern matching through template arguments

Defining a variadic template function

```
template <typename...ARGS>
void variadic(ARGS...args){
  println(std::cout,args...);
}
```

any number of types

any number of arguments

expand parameters as arguments

also a variadic template function

- Syntax uses `...` (ellipsis) symbol in several places
  - in front of a name -> **define** name as a placeholder for a variable number of elements
  - after a name -> **expand** the name to a list of all elements it represents
  - between two names -> define second name as a list of parameters, given by the first (treated as in front of)

variadic template terminology

```
template <typename...ARGS>
void variadic(ARGS...args){
  println(std::cout,args...);
}
```

template type parameter pack

function parameter pack

pack-expansion

- … left of a name : define parameter pack
- … right of a name/expression: pack expansion
- pack expansion can also be done after an expression using the parameter pack
  - this means the expression is repeated for each argument usually separated by commas
- special version of sizeof…(pack) yields n-of-args

Implementing a variadic function

```
void println(std::ostream &out) {
   out << "\n";
}
```

base case overload

- Key is recursion in the definitions
  - base case with zero arguments (matches in the end)
  - recursive case with 1 explicit argument and a tail consisting of a variadic list of arguments

```
template<typename Head, typename... Tail>
void println(std::ostream &out, Head const& head, Tail const& ...tail) {
   out << head;
   if (sizeof...(tail)) {
      out << ", ";
   }
   println(out,tail...); //recurse on tail
}
```

cut-off head

Gotcha 1 of Variadic Templates

```
template<typename Head, typename... Tail>
void println(std::ostream &out, Head const& head, Tail const& ...tail) {
   out << head;
   if (sizeof...(tail)) {
      out << ", ";
      println(out,tail...); //recurse on tail, at least one arg left
   } else {
      out<< '\n';
   }
}
```

- Recursion needs a base case, even if never called
  - compiler needs to instantiate all versions of parameter count and type combinations down to an empty parameter pack
- The above definition alone won't compile without base case

Olve Maudal's pub quiz (Q10)

```
#include <iostream>
template<typename T> void P(T x) {
   std::cout << x << ' ';
}
void foo(char a) {  P(3); P(a); }
template<typename ... A>
void foo(int a, A ... args) {
   foo(args...);
   P(a);
}
template<typename ... A>
void foo(char a, A ... args) {
   P(a);
   foo(args...);
}
int main() {
   foo('1', '2', 48, '4', '5');
}
```

- Gotcha #2
- Guess the output!
- What is the problem here?

Problem with Overload Resolution

```
void foo(char a) { P(3); P(a); }
template<typename ... A>
void foo(int a, A ... args) {
   foo(args...);
   P(a);
}
```

does not see foo(char,…)

- In template functions overload resolution happens based on the visible declarations at the point of the template definition, not the template instantiation
- Third foo(char,…) overload is invisible in second foo's body
- Forward-declare multiple overloads before first call
- not an issue within a class template, all member functions visible
  - but there we need to prefix any name from the class with this->

Variadic templates: CAUTION

```
template<typename Head, typename... Tail>
void println(std::ostream &out, Head const& head, Tail const& ...tail) {
   out << head;
   if (sizeof...(tail)) {
      out << ", ";
   }
   println(out,tail...); //recurse on tail
}
```

- When defining a variadic template function make sure all declarations of its overloads for function recursion or all declarations of specializations for class templates happen **before** the definition

- see also Olve's pub quiz question with variadic templates

```
void println(std::ostream &out) {
   out << "\n";
}
```

What about variadic class templates?

```
std::tuple<> empty{};
```

- required for std::tuple and its corresponding infrastructure
- hard to imagine significant other uses for plain class templates with typename parameter packs
- std::tuple is a means to pass parameter packs around as a single element

Simple std::tuple example

```
auto atuple= make_tuple(1,2.0,"three"s);
int a;
double b;
string s;
tie(a,b,s)=atuple;
ASSERT_EQUAL(1,a);
ASSERT_EQUAL(2.0,b);
ASSERT_EQUAL("three"s,s);
```

- std::make_tuple() creates a tuple of arbitrary values
- std::tie() creates a tuple of references
- multi-assignment using such tuples of references is possible
- main good use of tuple is in generic code only (see later)

std::tuple easy to misuse

```
std::tuple<bool,bool,bool> in = parseCmd(0, IN, cmd);
// expected: false, false, true
std::tuple<bool,bool,bool> out = parseCmd(0, OUT, cmd);
// expected: true, false, true
setInPtr(Aptr, Asize, std::get<0>(in),std::get<1>(in), std::get<2>(in));
setOutPtr(Aptr, Asize, std::get<0>(out), std::get<1>(out), std::get<2>(out));
```

real-world code example!

- returning multiple values ad-hoc with a tuple looks intriguing
- **It is a VERY BAD IDEA™.**
- STL is not without guilt using std::pair<A,B> as a return value for some functions with different semantics
- Better use the right abstraction and name it and its members. structs are OK! (or enums representing bitsets)

Other variadic class templates?

- use … in template template parameters
  - pass standard containers to templates
- use non-type template parameter pack
  - integer_sequence, apply
  - compile time computation of value sequences

Example template class Sack<T>

```
template <typename T>
class Sack
{
   using SackType=std::vector<T>;
   using size_type=typename SackType::size_type;
   SackType theSack{};
public:
   bool empty() const { return theSack.empty() ; }
   size_type size() const { return theSack.size();}
   void putInto(T const &item) { theSack.push_back(item);}
   T getOut() ;
};
template <typename T>
inline T Sack<T>::getOut(){
     if (! size()) throw std::logic_error{"empty Sack"};
     auto index = static_cast<size_type>(rand()%size());
     T retval{theSack.at(index)};
     theSack.erase(theSack.begin()+index);
     return retval;
}
```

how to parameterize underlying container template?

Varying Sack's Container

- A std::vector might not be the optimal data structure for a Sack, depending on its usage. E.g., removal from the "middle" requires shifting values in the vector
- What if we could specify the container type as well as a template parameter
- Requires using some more general API (usually without compromising performance, due to STL's design)
  - i.e., std::advance instead of +,
    insert() instead of push_back()

template template parameters

```
template <typename T, template<typename> class container>
class Sack1;
```

must be class!

- A template can take templates as parameters
  - template template parameters
- the template-template parameter must specify the number of typename parameters
- Problem: std:: containers take usually more than just the element type as template parameters

```
Sack1<int,std::vector> doesntwork;
```

std::vector<> is defined with 2 template parameters

template variadic template parameters

```
template <typename T,
    template<typename...> class container=std::vector>
class Sack
{
```

- Solution: use an arbitrary number of typename parameters: typename... (variadic template)

- This allows us also to specify a default container template as std::vector

- But also other containers for theSack member

```
Sack<int,std::list> listsack{1,2,3,4,5};
```

template-template argument factory

```
auto setsack=makeOtherSack<std::set>({'a','b','c','c'});
ASSERT_EQUAL(3,setsack.size());
```

- We can not determine the template-template argument in our factory function, only the element type
  - template parameter sequence must be switched

```
template <template<typename...> class container,typename T>
Sack<T,container> makeOtherSack(std::initializer_list<T> list)
{
  return Sack<T,container>{list};
}
```

FizzBuzz

(inspired by Kevlin Henney)

Function Overloading FizzBuzz

```
template<size_t ...values>
constexpr
auto fizzbuzz(std::index_sequence<values...>);
```

- API Design: pass values as compile-time numbers
  - uses std::index_sequence
  - different types for different value sequences
- unpack sequence of numbers, return tuple with individual values

```
constexpr auto fizzbuzz(std::index_sequence<>){
   return std::tuple<>{};
}
template<size_t value,size_t ...values>
constexpr auto fizzbuzz(std::index_sequence<value,values...>){
   return tuple_cat(
    fizzbuzz_number(std::index_sequence<value>{}),
    fizzbuzz(std::index_sequence<values...>{}));
}
```

base case overload

handle head

recurse tail

integer_sequence/index_sequence

```
template<class _Tp, _Tp... _Ip>
struct integer_sequence{…};
template<size_t... _Ip>
using index_sequence = integer_sequence<size_t, _Ip...>;
```

- C++14 introduces std::integer_sequence for representing a sequence of numbers as template arguments. (thanks to Jonathan Wakely)
- Can be deduced at compile time

```
fizzbuzz_number(std::index_sequence<1>());
fizzbuzz(std::make_index_sequence<17>{});//0,1,. . .,16
```

- make_index_sequence<n>{}: index_sequence<0,1,2,…,n-1>

FizzBuzz: numbers simple, but wrong

```
template<size_t value>
constexpr auto fizzbuzz_number(std::index_sequence<value>) {
   return std::make_tuple(value);
}
```

- Provide overloads for special cases: 0, 3, 5, …15 -> scale?

```
constexpr
auto fizzbuzz_number(std::index_sequence<0>){
   return std::tuple<>{};
}
constexpr
auto fizzbuzz_number(std::index_sequence<3>){
   return std::make_tuple("Fizz");
}
constexpr
auto fizzbuzz_number(std::index_sequence<5>){
   return std::make_tuple("Buzz");
}
constexpr
auto fizzbuzz_number(std::index_sequence<15>){
   return std::make_tuple("FizzBuzz");
}
```

0 -> nothing

3 -> Fizz

5 -> Buzz

- BUT… 6,9,10,12

Testing it

```
void testFizzBuzzNumberOne(){
   ASSERT_EQUAL(std::make_tuple(1),fizzbuzz_number(std::index_sequence<1>()));
}
void testFizzBuzzNumberTwo(){
   ASSERT_EQUAL(std::make_tuple(2),fizzbuzz_number(std::index_sequence<2>()));
}
void testFizzBuzzNumberThree(){
   ASSERT_EQUAL(std::make_tuple("Fizz"),fizzbuzz_number(std::index_sequence<3>()));
}
void testFizzBuzzNumberFive(){
   ASSERT_EQUAL(std::make_tuple("Buzz"),fizzbuzz_number(std::index_sequence<5>()));
}
void testFizzBuzzNumberFifteen(){
ASSERT_EQUAL(std::make_tuple("FizzBuzz"),fizzbuzz_number(std::index_sequence<15>()));
}
```

std::enable_if SFINAE Overload selection

```
template<size_t value>
constexpr auto fizzbuzz_number(std::index_sequence<value>,
     std::enable_if_t<value%3&&value%5,void*> =nullptr)
{  return std::make_tuple(value); }

template<size_t value>
constexpr auto fizzbuzz_number(std::index_sequence<value>,
      std::enable_if_t<value%3==0&&value%5,void*> =nullptr)
{  return std::make_tuple("Fizz"); }

template<size_t value>
constexpr auto fizzbuzz_number(std::index_sequence<value>,
     std::enable_if_t<value%3&&value%5==0,void*> =nullptr)
{  return std::make_tuple("Buzz"); }

template<size_t value>
constexpr auto fizzbuzz_number(std::index_sequence<value>,
     typename std::enable_if<value%15==0,void*>::type =nullptr)
{  return std::make_tuple("FizzBuzz"); }
```

C++11 style

- 4 cases: other and divisors:3, 5, 15

Testing the sequence

```
void testFizzBuzzSequenceUpTo16() {
   auto expected=std::make_tuple(
        1,2,"Fizz",4,"Buzz","Fizz",7,8,
        "Fizz","Buzz",11,"Fizz",13,14,"FizzBuzz",16);
   auto result=fizzbuzz(std::make_index_sequence<17>());
#ifndef SIMPLE_BUT_WRONG
   ASSERT_EQUAL(expected,result);
#else
   ASSERTM("can not compare tuples",false);
#endif
}
```

Testing the Sequence statically

- Talk description promised compile-time computation. Here it is for FizzBuzz:

```
constexpr auto expected=std::make_tuple(
     1,2,"Fizz",4,"Buzz","Fizz",7,8,
     "Fizz","Buzz",11,"Fizz",13,14,"FizzBuzz",16);
constexpr auto result=fizzbuzz(std::make_index_sequence<17>());

static_assert(expected==result, "Fizzbuzz should work at compile time");
```

- Demo: Compile Error when violated.

Outputting Fizzbuzz

```
namespace std{ // cheat for ADL
template <typename ...T>
std::ostream& operator<<(std::ostream &out,std::tuple<T...> const &toprint){
   auto const printer=[&out](auto&&...t){ println(out,t...);};
   apply(printer,toprint);
   return out;
}
} // std
```

printer(get<0>(toprint),get<1>(toprint)…)

Variadic Lambda! can only delegate, recursion impossible

- Could reuse our println variadic template function
  - but how do we get from a tuple to function arguments
- Need to put operator<< for tuple as a standard type into namespace std. Otherwise it won't be found, due to ADL.
  - You shouldn't put your own stuff into namespace std, but…

std::experimental::apply

```
template <typename F, typename Tuple, size_t... I>
constexpr decltype(auto) apply_impl(F&& f, Tuple&& t, std::index_sequence<I...>){
   return std::forward<F>(f)(std::get<I>(std::forward<Tuple>(t))...);
}
template <typename F, typename Tuple>
constexpr decltype(auto) apply(F&& f, Tuple&& t) {
   using Indices = std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>;
   return apply_impl(std::forward<F>(f), std::forward<Tuple>(t), Indices{});
}
```

pack expansion I... over whole expression

- Solution: std::experimental::apply (or its DIY version)
  - apply makes use of std::index_sequence and std::get<n> to access tuple elments
  - Indices keeps 0,…,n-1 as template arguments, empty object!

Can we do with class templates as well?

- Try class template partial specializations (doesn't scale :-( ):

```
template <size_t ...nums>
struct FizzBuzz;

template <>
struct FizzBuzz<>{…};

template <size_t n,size_t ...rest>
struct FizzBuzz<n,rest...>:FizzBuzz<rest...> {…};

template <size_t ...rest>
struct FizzBuzz<3,rest...>:FizzBuzz<rest...> {…};

template <size_t ...rest>
struct FizzBuzz<5,rest...>:FizzBuzz<rest...> {…};
```

Class Template Specialization

- How should it be done right then?
  - SFINAE to the rescue with std::enable_if_t
  - And we generate output directly from the objects -> pass ostream& to constructor and remember it (a bit hacky)

```
template <size_t ...nums>
struct FizzBuzz;
template <>
struct FizzBuzz<>{
   FizzBuzz(std::ostream&out):os{out}{}
   virtual ~FizzBuzz(){}
protected:
   std::ostream& os;
};
```

base case specialization

Variadic Class Templates: recurse through inheritance

```
template <size_t ...nums>
struct FizzBuzz;

template <size_t n,size_t ...rest>
struct FizzBuzz<n,rest...>:FizzBuzz<rest...>,FizzBuzzSingle<n> {
   FizzBuzz(std::ostream &out):FizzBuzz<rest...>{out}, FizzBuzzSingle<n>{out}{}
};

template <size_t ...rest>
struct FizzBuzz<0,rest...>:FizzBuzz<rest...> {
   FizzBuzz(std::ostream &out):FizzBuzz<rest...>{out}{}
};
```

single number dispatch

special case 0 specialization

Specialize single number dispatch

prepare for enable_if as template argument

```
template <size_t n,typename=void>
struct FizzBuzzSingle{
    FizzBuzzSingle(std::ostream &out):os{out}{}
    ~FizzBuzzSingle(){ this->os << n <<'\n'; }
private:
    std::ostream& os;
};

template <size_t n>
struct FizzBuzzSingle<n,
typename std::enable_if<n%3==0&&n%5>::type >{
    FizzBuzzSingle(std::ostream &out):os{out}{}
    ~FizzBuzzSingle(){
        this->os << "Fizz\n";
    }
private:
    std::ostream& os;
};
```

C++14 enable_if_t saves typename

```
template <size_t n>
struct FizzBuzzSingle<n,
        std::enable_if_t<n%3&&n%5==0>>{
    FizzBuzzSingle(std::ostream &out):os{out}{}
    ~FizzBuzzSingle(){
        this->os << "Buzz\n";
    }
private:
    std::ostream& os;
};

template <size_t n>
struct FizzBuzzSingle<n,
      std::enable_if_t<n%3==0&&n%5==0>>{
    FizzBuzzSingle(std::ostream &out):os{out}{}
    ~FizzBuzzSingle(){
        this->os << "FizzBuzz\n";
    }
private:
    std::ostream& os;
};
```

Testing it:

- need scope, because dtor side effect actually outputs

```
void FizzBuzzSequenceToBuzz(){
   std::ostringstream out;
   {
   FizzBuzz<1,2,3,4,5> fb(out);
   }
   ASSERT_EQUAL(
R"(1
2
Fizz
4
Buzz
)",out.str());
}
```

Create a whole sequence for testing

```
template<size_t ...nums>                                    Fizz
auto makeFizzBuzz(std::ostream &out,std::index_sequence<nums...>){    13
   return FizzBuzz<nums...>{out};                           14
}                                                           FizzBuzz
void FizzBuzzSequenceToFizzBuzzAndBeyond(){                 16
   std::ostringstream out;                                  17
   makeFizzBuzz(out,std::make_index_sequence<21>());        Fizz
   ASSERT_EQUAL(                                            19
R"(1                                                        Buzz
2                                                           )",out.str());
Fizz                                                        }
4
Buzz
Fizz
7
8
Fizz
Buzz
11
```

Wrap up

- using variadic templates is no magic, but requires recursion for compiler, even when never called
- variadic class templates are a bit less useful than variadic functions or lambdas
- template template parameters with variadic typename parameters can be handy for container wrappers
- stay tuned for Templator Demo and Variable Templates
  - with more on compile-time computation

More Questions?

- contact: peter.sommerlad@hsr.ch
- Looking for a better IDE:

Cevelop
Your C++ code deserves it

Download IDE at:
www.cevelop.com

- examples will be available at: https://github.com/PeterSommerlad/Publications