Object Oriented Scientific Programming in C++ (WI4771TU)

Matthias Möller
Numerical Analysis

TU Delft

Goal of this lecture

- Clarification on type traits
- Better understandaing of the different concepts
  - Function evaluation / recursion at run-time
  - Evaluation / recursion at compile-time
- Variadic templates
- Aliases & Enumerators
- C++ Standard container classes

Type traits

- Generic structure

```
template<typename type, type v>
struct trait {
    typedef <some type> type;
    type    value =     v;
};
```

- Examples
  - `session5::is_int`
  - `std::is_same`
  - `std::conditional`

Tasks 1

- Write a function that computes at run time „n!“ using
  – A for-loop (no recursion!)
  – Recursive function calling
- Write a function that computes at compile time „n!“ using
  – Template meta programming

TUDelft

Tasks 2

- Write a function that computes at compile time the Fibonacci number fib(n) using
  - Recursive function calling
  - A for-loop (no recursion!)
- Write a function that computes at run time the Fibonacci number using
  - Template meta programming

Summary

- Recursion formulas can be cast into for-loops can vice versa

```
double array_sum_for_loop(const double& array, int n)
    double d = 0;
    for (auto int=0, i<n; i++)
        d = d + array[i];
    return d;

double array_sum_recursion(const double& array, int n)
    return array[0] + array_sum_recursion(array[1],n-1);
```

TUDelft

Summary, cont'd

- Run-time recursion
  - Gives rise to repeated function calls (data/instruction copies)
  - Smart compilers try to eliminate recursive calls if possible
- Compile-time recursion
  - Expression is evaluated by the compiler (no repeated function calls and data/instruction copies at run time)
  - Maximal admissible recursion level depends on compiler

Variadic templates

- Task: implement a function that takes an **arbitrary number** of possible **different variables** and computes their sum
  `cout << sum(1.0, (int)1, (float)1.3, (double)1.3) << endl;`
- None of the template meta programming techniques we know so far will solve this probem satisfactory
  - New concept in C++11: **variadic template parameters**
  - Idea: reformulate the task as follows
    task(n) = task (1) ◆ task(n-1)

Intermezzo: recursive templates

- Recall the Fibonacci assignment

```
template<int n>
struct fib {
  static const int value = fib<n-1>::value + fib<n-2>::value;
};

// Specialisations for n=0 and n=1
template<>
struct fib<0> { static const int value = 1; }
template<>
struct fib<1> { static const int value = 1; }
```

Variadic templates, cont'd

- **Forward declaration** of the generic function

```
template<typename ... Ts>
static double sum(Ts ... args);
```

- Specialisation for one arguments

```
template<typename T>
static double sum(T a) { return a; }
```

- Specialisation that peals-off the first task

```
template<typename T, typename ... Ts>
static double sum(T a, Ts ... args)
{ return a + sum(args...); }
```

TUDelft

Variadic templates, cont'd

- The template parameter pack (type ... Args) accepts zero or more template arguments, hence, we need a specialisation for zero arguments as well

```
template<>
static double sum() { return 0; }
```

- There cannot be more than one template parameter pack since it is impossible to deduce where the second starts

```
template<typename ... Ts, typename ... Us>
static double sum(Ts ... args, Us ... Args) {...}
```

Variadic templates, cont'd

- The number of arguments in the parameter pack can be detected using the sizeof...() function

```
template<typename ... Ts>
static double sum(Ts ... args)
{
    const int size = sizeof...(args);
}
```

- Task: Write a trait that determines the number of arguments in a parameter pack

Tuples

- The container std::tuples makes it possible to store an arbitrary number of heterogenous types

```
#include <tuple>
auto t = std::make_tuple(3.8, ‚A‘, „String“);
```

- Access to individual elements

```
std::cout << std::get<0>(t) << std::endl;
```

- Create tuple from parameter pack

```
auto t = std::tuple<Ts...>(args...);
```

Aliases

- Concept to introduce a synonyme for a type or template

```
template<typename T>
struct demo {
    using type = T;
};
```

- Task: reimplement the session5::is_int trait that way

Enumerators

- Enumerators make it possible to collect named values

```
enum class Color { red, green, blue };
```

- Named values are mapped to, e.g., `red=0, green=1, blue=2`
- Usage (more human-readable than `case 1:, case 2:`, …)

```
Color col = Color::red;
switch col {
    case Color::red:   // do something
    case Color::green: // do something else
}
```