Program specialization and metaprogramming

Laboratory work No. 3

Static table generation using variadic templates in C++

1. Aims

To get acquainted with the advanced topics of template metaprogramming in C++.

2. Tasks

Learn how to use C++ variadic templates for performing data specialization.

3. Work

Data specialization aims at encoding results of early computations in data structures. The execution of a program is divided into two stages. First, a part of the algorithm is pre-computed in advance and the results are saved in a data structure such as look-up table (LUT) or static table. A LUT usually is an array (cache), which replaces a runtime computation with a simpler memory access operation. The speed gain can be significant, because retrieving a value from memory is faster than undergoing an expensive computation.

Specialization of the algorithm to use data tables instead of computations is performed as follows. (1) Analyze the application source code to identify references to the computation costly functions. (2) Generate a LUT for the specialized function using the metaprogramming techniques. (3) Replace all references to the function by the reference to its LUT.

The benefit of static tables is the replacement of "expensive" calculations with a simple array indexing operation (for examples see lookup table). In C++ exists more than one way to generate a static table at compile time.

In this lab we will learn to generate data tables at compile time using recursive struct definitions and variadic templates.

Variadic templates are class templates that take a variable number of arguments.

```
template<typename... Values> class tuple;              // takes zero
or more arguments
```

Similarly, variadic functions are function templates that take a variable number of arguments (only in C++ 17).

```
template<typename... Args>
auto adder(Args... args) {
  return (... + args);
  // or (args + ...)
}
```

The following listing shows an example of creating a very simple table by using recursive structs and Variadic templates.

```
#include <array>

constexpr int TABLE_SIZE = 10;

/**
 * Variadic template for a recursive helper struct.
 */
template<int INDEX = 0, int ...D>
struct Helper : Helper<INDEX + 1, D..., INDEX * INDEX> { };

/**
 * Specialization of the template to end the recursion when the table
size reaches TABLE_SIZE.
 */
template<int ...D>
struct Helper<TABLE_SIZE, D...> {
  static constexpr std::array<int, TABLE_SIZE> table = { D... };
};

constexpr std::array<int, TABLE_SIZE> table = Helper<>::table;

enum  {
  FOUR = table[2] // compile time use
};
```

The idea behind this is that the struct Helper recursively inherits from a struct with one more template argument (in this example calculated as INDEX * INDEX) until the specialization of the template ends the recursion at a size of 10 elements. The specialization simply uses the variable argument list as elements for the array.

To check the generated table, the following code can be used in the main() function:

```
  for(int i=0; i < TABLE_SIZE; i++) {
    std::cout << table[i]  << std::endl; // run time use
  }
  std::cout << "FOUR: " << FOUR << std::endl;
```

What is the result of program execution?

To show a more sophisticated example the code in the following listing has been extended to have a helper for value calculation (in preparation for more complicated computations), a table specific offset and a template argument for the type of the table values (e.g. uint8_t, uint16_t, ...).

```
constexpr int TABLE_SIZE = 20;
constexpr int OFFSET = 12;

/**
```

```
 * Template to calculate a single table entry
 */
template <typename VALUETYPE, VALUETYPE OFFSET, VALUETYPE INDEX>
struct ValueHelper {
  static constexpr VALUETYPE value = OFFSET + INDEX * INDEX;
};

/**
 * Variadic template for a recursive helper struct.
 */
template<typename VALUETYPE, VALUETYPE OFFSET, int N = 0, VALUETYPE
...D>
struct Helper : Helper<VALUETYPE, OFFSET, N+1, D...,
ValueHelper<VALUETYPE, OFFSET, N>::value> { };

/**
 * Specialization of the template to end the recursion when the table
size reaches TABLE_SIZE.
 */
template<typename VALUETYPE, VALUETYPE OFFSET, VALUETYPE ...D>
struct Helper<VALUETYPE, OFFSET, TABLE_SIZE, D...> {
  static constexpr std::array<VALUETYPE, TABLE_SIZE> table = { D... };
};

constexpr std::array<uint16_t, TABLE_SIZE> table = Helper<uint16_t,
OFFSET>::table;
```

To check the generated table, the following code can be used in the main() function:

```
  for(int i = 0; i < TABLE_SIZE; i++) {
    std::cout << table[i] << std::endl;
  }
```

What is the result of program execution?

4. Challenge

1. Modify the program to create static tables for calculating approximate values of sine and cosine functions at compile time.
2. Modify the program to create static tables for calculating approximate values of gaussian distribution at compile time.
3. Modify the program to create static matrices (2D tables) rather than a single dimension table for functions with two parameters.
4. Perform time measurements, what is the execution time of calling a static table entry vs standard (library-based or custom) implementations using C++. Use clock() function or other to measure CPU time.

Present and discuss the results in your report.

5. Report

Prepare a report of your work in this lab using a standard word processing application. The report is a single chapter of your final semester report. Final semester report will have to be uploaded to course Moodle page.

The content of report:

1. Title
2. Aims and tasks
3. Work done, described in steps and illustrated by screenshots and written or modified source code. Provide comments what you have done and what were the results (program outputs). Present answers to the questions given in the lab description.
4. Conclusions – what you have learned?