# Systems Engineering and Data Processing in C++

**Summer Term 2021**

Hasso Plattner Institute
Computer Graphics Systems Group

Prof. Dr. Jürgen Döllner
and Willy Scheibel

# Assignments #5: Generic Programming and Templates

## Objectives

In these assignments you will learn how to:

- write types and functions with a variable number of arguments,
- safely implement type erasure,
- write functions that operate on a variety of types,
- make use of template argument deduction, and
- convert template meta programming constructs to modern `constexpr`.

## Rewards

For each successfully solved task you and your partner (optional) gain one bonus exam point. A serious shortcoming in fulfilling the requirements of a task will result in no bonus point regarding this task. You can gain at most 4 bonus points through this exercise sheet.

## Submission

All described artifacts has to be submitted to the course moodle system[1] as a zipped archive. The naming convention includes the assignment number and your *personal assessment numbers* with following naming convention: `assignment_5_paNr1.zip` or `assignment_5_paNr1_paNr2.zip` whether or not pair programming was applied. Compiled, intermediate, or temporary files should not be included. If pair programming is used, the results are turned in only once. The assignments #5 are due to the next tutorial on July 1st, 9:15 a.m. GMT+2.

## Presentation

Feel free to present any of your solutions during the Zoom meeting in the upcoming tutorial. If you want to present your solution of a task, please send the source code beforehand via e-mail to willy.scheibel@hpi.de.

## General Instructions

**Pair Programming** On these assignments, you are encouraged (not required) to work with a partner, provided you practice pair programming. Pair programming "is a practice in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test." One partner is driving (designing and typing the code) while the other is navigating (reviewing the work, identifying bugs, and asking questions). The two partners switch roles every 30–40 minutes, and on demand, brainstorm.

**Cross-Platform** The assignments can be solved on all major platforms (i.e., Windows, Linux, macOS). The evaluation of submitted assignments can be carried out on any of these platforms. All results should not use platform-specific dependencies. Platform-specific build and macro code is permitted.

**Violation of Rules** a violation of rules results in grading the affected assignments with 0 points.

- Writing code with a partner without following the pair programming instructions listed above represents a serious violation of the course collaboration policy.
- Plagiarism represents a serious violation of the course policy.

[1] https://moodle.hpi.de/course/view.php?id=174

## Task 5.1: Generic Signal Handler

Files: `signal/Signal.h`, `signal/main.cpp`

The `Signal` class represents a generic handler for signals that call a set of connected callbacks. A `Signal` object can register an arbitrary number of callable contexts. If emitted, the signal invokes all registered callbacks, passing on the given arguments.
Implement the template class `Signal` with a variadic template argument list to support passing arbitrary arguments from the signal emitter to the connected callbacks. The signal class should be capable of registering different types of callable contexts (e.g., functions, member functions applied to objects, and lambdas).

Constraint:

- The solution must contain a single `Signal` variadic template class, written without template specialization.
- Callbacks can be registered using the `connect` method.
- A signal can be fired by calling the `emit` method for the `Signal` object.
- Do not modify the given test case.

Example:

```
struct MyStruct
{
    void foo(int i, std::string s) {}
};

// Define a Signal
Signal<int, std::string> mySignal;

// Connect some callbacks
mySignal.connect([](int i, std::string s) { std::cout << i << ": " << s; });

MyStruct s;
mySignal.connect(s, &MyStruct::foo);

// Fire signal
mySignal.emit(42, "FOO");
```

Artifacts:

a) `signal/Signal.h`, `signal/...` : Source code of the solution.

Objectives: `std::function`, member function pointers, variadic templates, signal system

## Task 5.2: Dynamic, Type-Safe Variant

Files: `variant/Variant.h`, `variant/main.cpp`

Design and implement a non-template `Variant` class that can hold objects and values of arbitrary types. More specifically, the `Variant` must behave as follows:

- It can hold any value (type and value passed to the constructor).
- A query function can be used to test the variant for a specific type.
- The value can be obtained if the correct type is used. If not, an `std::logic_error` is thrown.
- A default-constructed (or moved-from) Variant is *empty*, i.e., has no type or value.
- Destruction of the variant must not leak memory.

Example:

```
Variant v1 = Variant(4);
v1.hasType<int>();   // => true
v1.hasType<float>(); // => false
v1.get<int>();       // => 4
v1.get<float>();     // => std::logic_error thrown
```

Constraints:

- The `Variant` class must not be a class template. It is permitted to use class and function templates as members.

Artifacts:

a) `variant/Variant.h`, `variant/main.cpp`, `variant/...` : Source code of the solution.

Objectives: dynamic type system, run-time type information, template parameter deduction, member templates

## Task 5.3: Collection Transformation

Files: `collect/collect.h`, `collect/main.cpp`

Design and implement a series of higher-order function overloads and specializations named `collect` that traverse an iterable object or sequence and perform a given per-element transformation, collecting the results into a new iterable object of parameterizable type.
This function could be used, e.g., to convert a vector of integers to a vector of floats or to call a getter function or query function on each object of a set and return the resulting values as a new list.

Example:

```
auto floatVector = std::vector<float>{ 0.0f, 0.5f, 2.4f, 3.0f, 3.0f };

// Get the set of unique integers from a vector of floats
auto uniqueInts  = collect<std::set<int>>(floatVector, [](float v) { return static_cast<int>(v); });

// Ideal version if result container matches source container type
// (e.g., vector -> vector)
auto intVector = collect(floatVector, [](float v) { return static_cast<int>(v); });
```

Extend the implementation such that the test code compiles and links successfully.

- The type of the result container must be specified as template parameter and all overloads must be named `collect`, do not provide functions with different names (e.g., `collectVector`, `collectList`, `...`).
- You are not required to support custom allocators (e.g., `std::vector<int, MyAllocator>`).

Artifacts:

a) `collect/collect.h`, `collect/main.cpp`, `collect/...` : Source code of the solution.

Objectives: template parameter deduction, template specialization, function templates

Competition: Extend your solution with a template argument deduction system that can handle as many of the advanced test cases as possible. If you want to compete in the open competition submit your isolated source code of this task via e-mail to willy.scheibel@hpi.de until June 29$^{th}$. Your source code is checked for conformity with respect to the expected output. If the source code behaves as expected, we assess the number of test cases your code can handle.

## Task 5.4: Refactoring of Template Meta Programs

Files: `template-refactoring/ISBN.h`, `template-refactoring/main.cpp`

Given a compile-time ISBN parser implemented using template meta-programming, refactor the source code to use `constexpr` functions instead. Because of the nature of user-defined literals in combination with either templates or `constexpr` functions, the interface entry points have to be slightly adjusted.
For example, the original code `978'4873'113'68'5_isbn` shall then be expressed as `"978'4873'113'68'5"_isbn`, but the parser logic should stay similar.

Artifacts:

a) `template-refactoring/main.cpp`, `template-refactoring/...` : Source code of the solution.

Objectives: template meta-programming, `constexpr`, re-engineering