Extensions for Disambiguation Tags

Introduction

Currently, there are disambiguation tag templates defined in the standard, including `in_place_type`, `in_place_index`, etc. However, these components are not enough to carry every sort of metadata required by function templates, such as enumerations, floating numbers or user-defined static data structures.

This paper proposes 2 disambiguation tag templates, which provide generic expressions to pass various sort of metadata to function templates. I think this design is useful when recursively calling constexpr functions in non-constexpr ones with custom input, and therefore would help standardize the technical specifications in compile-time programming.

Technical Specification

```cpp
namespace std {

    template <class T, T V>
    struct in_place_arg_t {
        explicit in_place_arg_t() = default;
    };

template <class T, T V>
    inline constexpr in_place_arg_t<T, V> in_place_arg{};

    template <class T, const T& V>
    inline constexpr in_place_arg_t<T, const V> in_place_arg{std::move(V)};

}```
struct in_place_resource_t {
    explicit in_place_resource_t() = default;
};
template <class T, const T& V>
inline constexpr in_place_resource_t<T, V> in_place_resource{};

Users are allowed to pass constexpr values by `in_place_arg`, and pass static constexpr resources by `in_place_resource`.
Additionally, I suggest that `in_place_index_t` should be an alias of `in_place_arg_t`:

```
template <size_t I>
using in_place_index_t = in_place_arg_t<size_t, I>;
```

Comparing to `in_place_arg_t`, I think `integral_constant` is inappropriately named, and there seem to be little necessity to define any member types or constants in it, because these metadata is already passed by templates.

3 Sample Usage

With the support of `in_place_arg`, it becomes easy to pass any constexpr value (providing its type is valid for a template non-type parameter) to a function template using a uniform disambiguation tag, especially with constructors.
Providing there is a enum class defined as follows:

```
enum class State {
    RUNNINE, AVAILABLE, OFFLINE
};
```

And there is a constexpr function that could convert a `State` to its string expression:

```
constexpr const char* make_state_str(State s) {
    switch (s) {
        case State::RUNNINE: return "Running State";
        case State::AVAILABLE: return "Available State";
        case State::OFFLINE: return "Offline State";
        default: return "Unknown State";
    }
}
```

It is relatively easy to design a class with `in_place_arg`, which is explicitly constructible from a `constexpr State` and stores its string expression without executing the constexpr function `make_state_str` at runtime:

```
class Machine {
```
public:
    template <State S>
    explicit Machine(std::in_place_arg_t<State, S>) : state_str_(STATE_STR<S>()) {}

    const char* get_state_str() { return state_str_; }

private:
    const char* state_str_;

    template <State S>
    static constexpr const char* STATE_STR = make_state_str(S);

Machine machine(std::in_place_arg<State, State::AVAILABLE>);
puts(machine.get_state_str());

    `in_place_resource` has a wider scope of application than `in_place_arg` does, because it could carry all sort of constexpr data if the data is prior declared.
    For example, providing there is a struct carries some configuration:

struct Config {
    double EPS = 1e-8;
    int INF = 0x7fffffff;
    long long INFL = 0x7fffffff7fffffff;
} constexpr MATH_CONFIG;

    It is allowed to initialize a type with the resource by templates with `in_place_resource`, even if we are not sure about the concrete type of the resource:

class Calculator {
    public:
        template <class T, const T& CONFIG>
        explicit Calculator(std::in_place_resource_t<T, CONFIG>);
    }

Calculator calculator(std::in_place_resource<Config, MATH_CONFIG>);