Writing and using compile-time heterogeneous hash-table

Denis Panin, NVIDIA
Senior System Software Engineer

```
# create_ip 1.2.3.4 255.255.255.0
```

```
# create_ip 1.2.3.4 255.255.255.0

map<string, bool(*)(vector<string>)>
```

```
# create_ip 1.2.3.4 255.255.255.0

map<string, bool(*)(vector<string>)>

bool callback(vector<string> args) {



}
```

```
# create_ip 1.2.3.4 255.255.255.0

map<string, bool(*)(vector<string>)>

bool callback(vector<string> args) {
    if (args.size() != 2) return false;
    ipv4 ip = convert_ip(args[0]);
    if (some_error_check) return false;
    ipv4 mask = convert_ip(args[1]);
    if (some_error_check) return false;
    // do actual work
    return true;
}
```

```
# create_ip 1.2.3.4 255.255.255.0

map<string, bool(*)(vector<string>)>

bool callback(vector<string> args) {
    if (args.size() != 2) return false;
    ipv4 ip = convert_ip(args[0]);
    if (some_error_check) return false;
    ipv4 mask = convert_ip(args[1]);
    if (some_error_check) return false;
    // do actual work
    return true;
}
```

```
# create_ip 1.2.3.4 255.255.255.0

map<string, bool(*)(vector<string>)>

bool callback(vector<string> args) {
    if (args.size() != 2) return false;
    ipv4 ip = convert_ip(args[0]);
    if (some_error_check) return false;
    ipv4 mask = convert_ip(args[1]);
    if (some_error_check) return false;
    // do actual work
    return true;
}
```

```
# create_ip 1.2.3.4 255.255.255.0

map<string, bool(*)(vector<string>)>

bool callback(vector<string> args) {
    if (args.size() != 2) return false;
    ipv4 ip = convert_ip(args[0]);
    if (some_error_check) return false;
    ipv4 mask = convert_ip(args[1]);
    if (some_error_check) return false;
    // do actual work
    return true;
}
```

How it was done in our project?

How it was done in our project?

1. One function. 20K LOCs with million if's

How it was done in our project?

1. One function. 20K LOCs with million if's
2. Internal magical variables, meaning lost

How it was done in our project?

1. One function. 20K LOCs with million if's
2. Internal magical variables, meaning lost
3. Copy-paste driven development

How it was done in our project?

1. One function. 20K LOCs with million if's
2. Internal magical variables, meaning lost
3. Copy-paste driven development
4. Recursively calls itself :D

How it was done in our project?

1. One function. 20K LOCs with million if's
2. Internal magical variables, meaning lost
3. Copy-paste driven development
4. Recursively calls itself :D

IT WORKS AND NO ONE IS GOING TO REWRITE IT

But what do we want?

But what do we want?

1. Normal signature -> void create(ipv4 ip)

But what do we want?

1. Normal signature -> void create(ipv4 ip)
2. Automatic args count verification.

But what do we want?

1. Normal signature -> void create(ipv4 ip)
2. Automatic args count verification.
3. Automatic string arguments conversion.

But what do we want?

1. Normal signature -> void create(ipv4 ip)
2. Automatic args count verification.
3. Automatic string arguments conversion.

Adding new command should take 1 string without any boilerplate

```
void func(const T1& t1, T2 t2);
vector<string> args{ "t1_str", "t2_str" };
```

```
void func(const T1& t1, T2 t2);
vector<string> args{ "t1_str", "t2_str" };

if (get_arg_count(func) != args.size()) {
    return error;
}
```

```
void func(const T1& t1, T2 t2);
vector<string> args{ "t1_str", "t2_str" };

if (get_arg_count(func) != args.size()) {
    return error;
}

tuple<get_arg_types(func)> tpl;
```

```
void func(const T1& t1, T2 t2);
vector<string> args{ "t1_str", "t2_str" };

if (get_arg_count(func) != args.size()) {
    return error;
}

tuple<get_arg_types(func)> tpl;
for (size_t i = 0; i < args.size(); ++i) {
    using target_type = tuple_element_type<i>(tpl);


}
```

```
void func(const T1& t1, T2 t2);
vector<string> args{ "t1_str", "t2_str" };

if (get_arg_count(func) != args.size()) {
    return error;
}

tuple<get_arg_types(func)> tpl;
for (size_t i = 0; i < args.size(); ++i) {
    using target_type = tuple_element_type(tpl, i);
    tuple[i] = convert_to<target_type>(args[i]);
    if (invalid(tuple[i])) return error;
}
```

```
void func(const T1& t1, T2 t2);
vector<string> args{ "t1_str", "t2_str" };

if (get_arg_count(func) != args.size()) {
    return error;
}

tuple<get_arg_types(func)> tpl;
for (size_t i = 0; i < args.size(); ++i) {
    using target_type = tuple_element_type(tpl, i);
    tuple[i] = convert_to<target_type>(args[i]);
    if (invalid(tuple[i])) return error;
}

func(expand(tpl)...);
```

```
if (fname == "func1")
    execute(func1, vec_args);

if (fname == "func2")
    execute(func2, vec_args);

if (fname == "func3")
    execute(func3, vec_args);
```

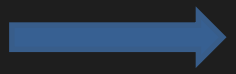
Please no

```
if (fname == "func1")
    execute(func1, vec_args);

if (fname == "func2")
    execute(func2, vec_args);

if (fname == "func3")
    execute(func3, vec_args);
```

We can do better



```
const auto ftbl = make_tuple(
    pair("func1", &func1),
    pair("func2", &func2),
    pair("func3", &func3)
);
lookup_exe(ftbl, "func1", vec_args);
```

Lets make compile-time heterogeneous hash-table with O(1) lookup!

```
size_t cthash(string_view s) {
    size_t hash{ 0 };
    for (char c : s) {
        hash += c * PRIME_1 ^ PRIME_2;
    }
    return hash;
}
```

```
constexpr // yes, really that's enough
size_t cthash(string_view s) {
    size_t hash{ 0 };
    for (char c : s) {
        hash += c * PRIME_1 ^ PRIME_2;
    }
    return hash;
}
```

```
constexpr // yes, really that's enough
size_t cthash(string_view s) {
    size_t hash{ 0 };
    for (char c : s) {
        hash += c * PRIME_1 ^ PRIME_2;
    }
    return hash;
}

struct_with_int<cthash("CSTRING")> si;
```

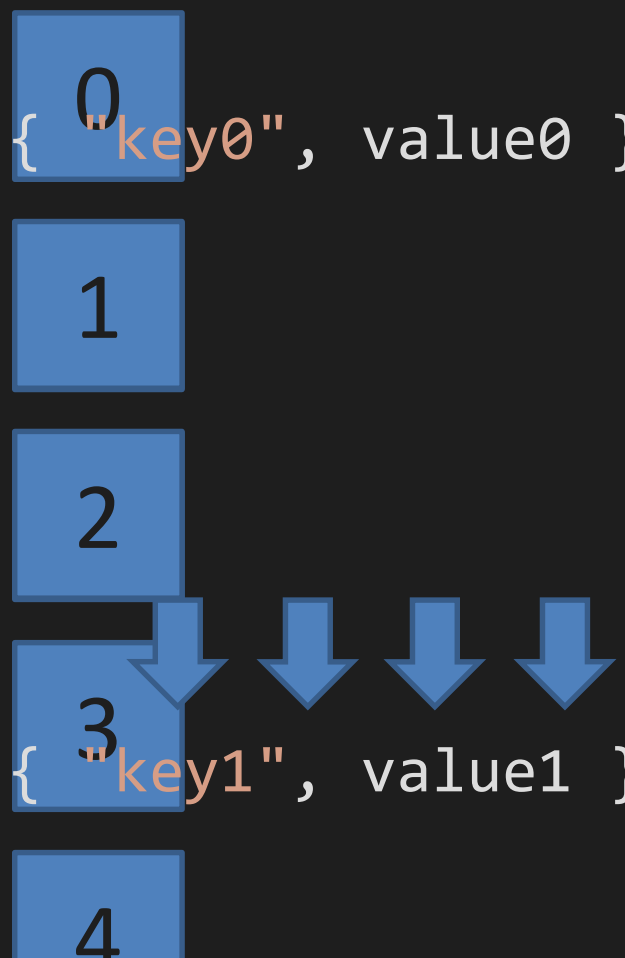
buckets = 5


{ "key4", value4 }
0
{ "key0", value0 }, { "key2", value2 }
1
2
3
{ "key1", value1 }, { "key3", value3 }
4

buckets = 5

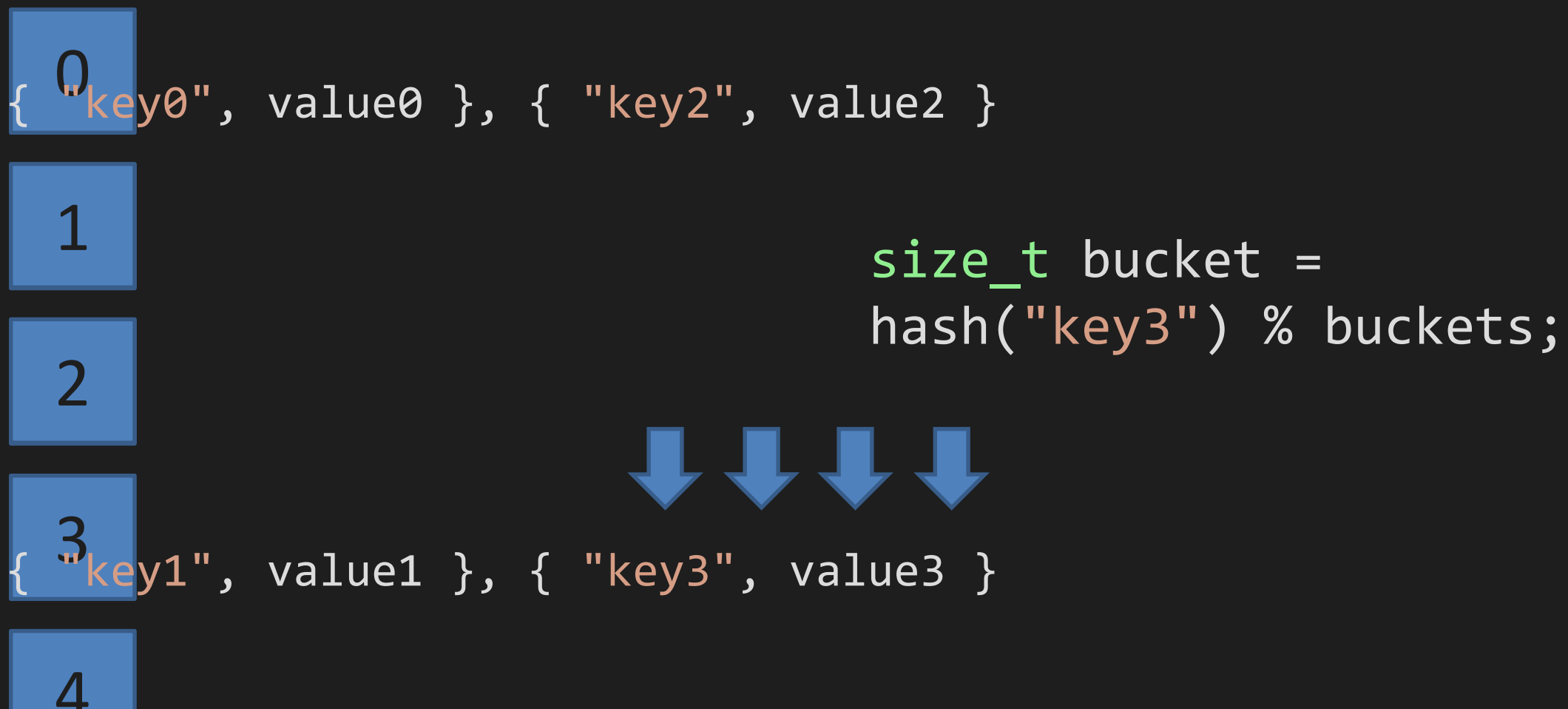

```
size_t bucket =
hash("key3") % buckets;
```

buckets = 5




```
size_t bucket =
hash("key3") % buckets;
```

buckets = 5




```
size_t bucket =
hash("key3") % buckets;
```

```
int main() {
    const auto htbl tbl(
        HASH_TABLE_ENTRY("key1", 1),
        HASH_TABLE_ENTRY("key2", nullptr)
    );









}
```

```
int main() {
    const auto htbl tbl(
        HASH_TABLE_ENTRY("key1", 1),
        HASH_TABLE_ENTRY("key2", nullptr)
    );

    tbl.execute("key1", func);
    tbl.execute("key2", func);
    if (!tbl.execute("key3", func))
        cout << "Not found :(" << endl;
}
```

```cpp
int main() {
    const auto htbl tbl(
        HASH_TABLE_ENTRY("key1", 1),
        HASH_TABLE_ENTRY("key2", nullptr)
    );

    const auto func = [](auto val) {
        cout << typeid(val).name() << endl;
    };

    tbl.execute("key1", func);
    tbl.execute("key2", func);
    if (!tbl.execute("key3", func))
        cout << "Not found :(" << endl;
}
```

```
int main() {
    const auto htbl tbl(
        HASH_TABLE_ENTRY("key1", 1),
        HASH_TABLE_ENTRY("key2", nullptr)
    );

    const auto func = [](auto val) {
        cout << typeid(val).name() << endl;
    };

    tbl.execute("key1", func);
    tbl.execute("key2", func);
    if (!tbl.execute("key3", func))
        cout << "Not found :(" << endl;
}
```

```cpp
int main() {
    const auto htbl tbl(
        HASH_TABLE_ENTRY("key1", 1),
        HASH_TABLE_ENTRY("key2", nullptr)
    );

    const auto func = [](auto val) {
        cout << typeid(val).name() << endl;
    };

    tbl.execute("key1", func);
    tbl.execute("key2", func);
    if (!tbl.execute("key3", func))
        cout << "Not found :(" << endl;
}
```

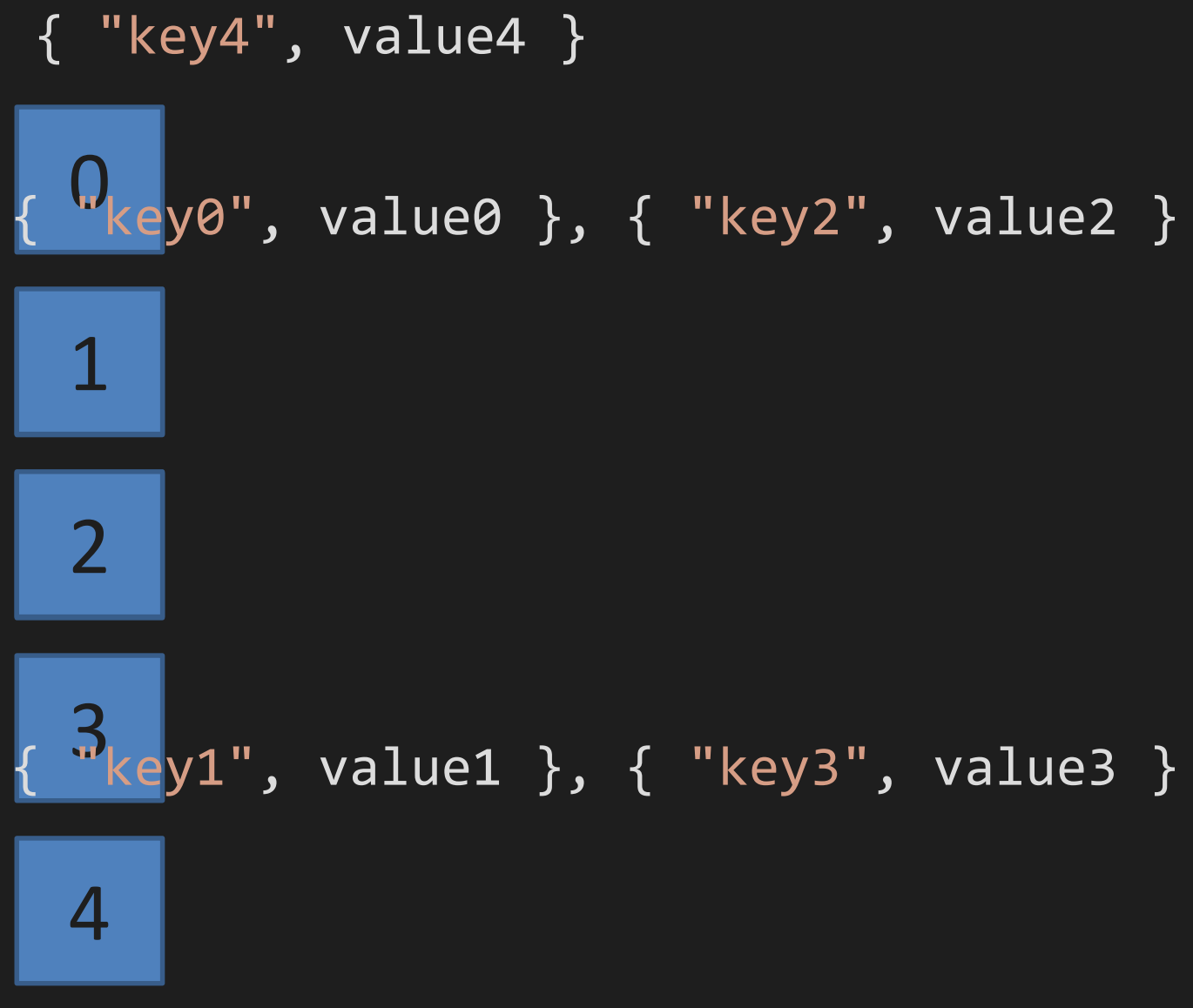
```cpp
int main() {
    const auto htbl tbl(
        HASH_TABLE_ENTRY("key1", 1),
        HASH_TABLE_ENTRY("key2", nullptr)
    );

    const auto func = [](auto val) {
        cout << typeid(val).name() << endl;
    };

    tbl.execute("key1", func);
    tbl.execute("key2", func);
    if (!tbl.execute("key3", func))
        cout << "Not found :(" << endl;
}
```

```
template <size_t HASH, typename VAL>
struct hash_table_entry {
  const string_view key;
  const VAL val;
  static constexpr auto hash = HASH;
};
```

```
#define HASH_TABLE_ENTRY(key, val) \
hash_table_entry<cthash(key), decltype(val)> (key, val)
```


{ "key4", value4 }
0
{ "key0", value0 }, { "key2", value2 }
1
2
3
{ "key1", value1 }, { "key3", value3 }
4

```
tuple(
        tuple( { "key4", value4 } ),

       tuple( { "key0", value0 }, { "key2", value2 } ),

        tuple(),

        tuple(),

      tuple( { "key1", value1 }, { "key3", value3 } )
);
```

```
tuple(
        tuple( { "key4", value4 } ),

       tuple( { "key0", value0 }, { "key2", value2 } ),

        tuple(),
        tuple(),

      tuple( { "key1", value1 }, { "key3", value3 } )
);
```

Don't try this at home,
unless you have >1TB RAM and
few spare weeks

```
create_table(
    pair0, // B1
    pair1, // B4
    pair2, // B1
    pair3, // B4
    pair4, // B0
)
```

```
array(
    { pair4 },
    { pair0, pair2 },
    {},
    { pair1, pair3 },
    {}
)
```

```
create_table(
    pair0, // B1
    pair1, // B4
    pair2, // B1
    pair3, // B4
    pair4, // B0
)
```

```
array(
    { pair4 },
    { pair0, pair2 },
    {},
    { pair1, pair3 },
    {}
)
```

Element count per bucket.
array(1, 2, 0, 2, 0);

```
template <typename TPL>
using hashmeta_t = array<size_t, tuple_size_v<TPL>>;
```

```cpp
template <typename TPL>
using hashmeta_t = array<size_t, tuple_size_v<TPL>>;

hashmeta_t<TPL> occurences = {}; // all zeroes
calc_occurences(tpl, occurences);
```

```cpp
template <typename TPL>
using hashmeta_t = array<size_t, tuple_size_v<TPL>>;

hashmeta_t<TPL> occurences = {}; // all zeroes
calc_occurences(tpl, occurences);

template <size_t IDX = 0, typename TPL>
constexpr void calc_occurences(const TPL& tpl, hashmeta_t<TPL>& arr) {
    static constexpr auto sz = tuple_size_v<TPL>;
    if constexpr (sz > IDX) {
        const auto bucket = tuple_element_t<IDX, TPL>::HASH % sz;
        ++arr[bucket];
        calc_occurences<IDX + 1>(tpl, arr);
    }
}
```

```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

```
array(
    { pair4 },
    { pair0, pair2 },
    {},
    { pair1, pair3 },
    {}
)
```

Element count per bucket.
array(1, 2, 0, 2, 0);

Offsets to bucket start.
array(0, 1, 3, 3, 5);

[1, 2, 0, 2, 0] -> [0, 1, 3, 3, 5]

```
template <typename TPL>
constexpr auto calc_offsets(const TPL& tpl) {
    hashmeta_t<TPL> occ = {};
    calc_occurences(tpl, occ);
    hashmeta_t<TPL> off = {};
    for (size_t i = 1; i < tuple_size_v<TPL>; ++i) {
        off[i] = off[i - 1] + occ[i - 1];
    }
    return offsets;
}
```

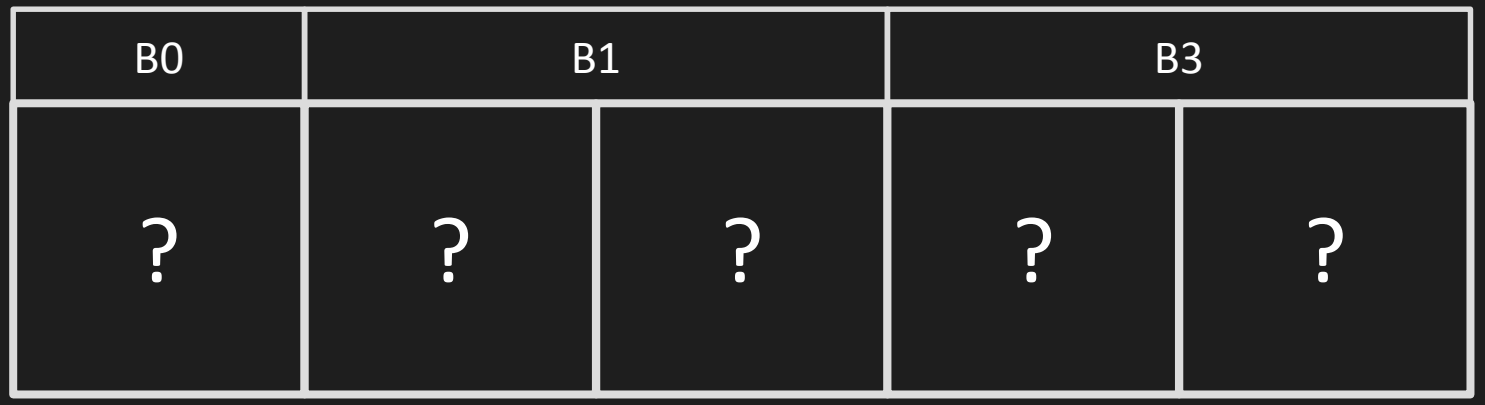
```
create_table(
   pair0, // B1
   pair1, // B3
   pair2, // B1
   pair3, // B3
   pair4, // B0
)
```

Copy of offsets
[0, 1, 3, 3, 5]

```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

Copy of offsets
[0, 1, 3, 3, 5]

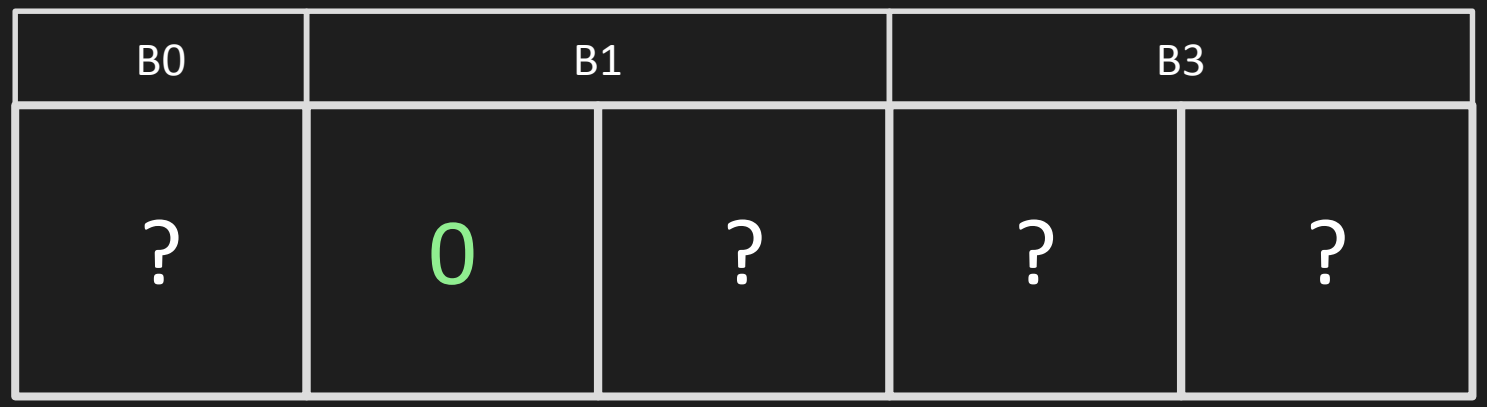
Array of indexes



```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

Copy of offsets
[0, 1, 3, 3, 5]

2



Array of indexes

```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

Copy of offsets
[0, 2, 3, 3, 5]

4

Array of indexes

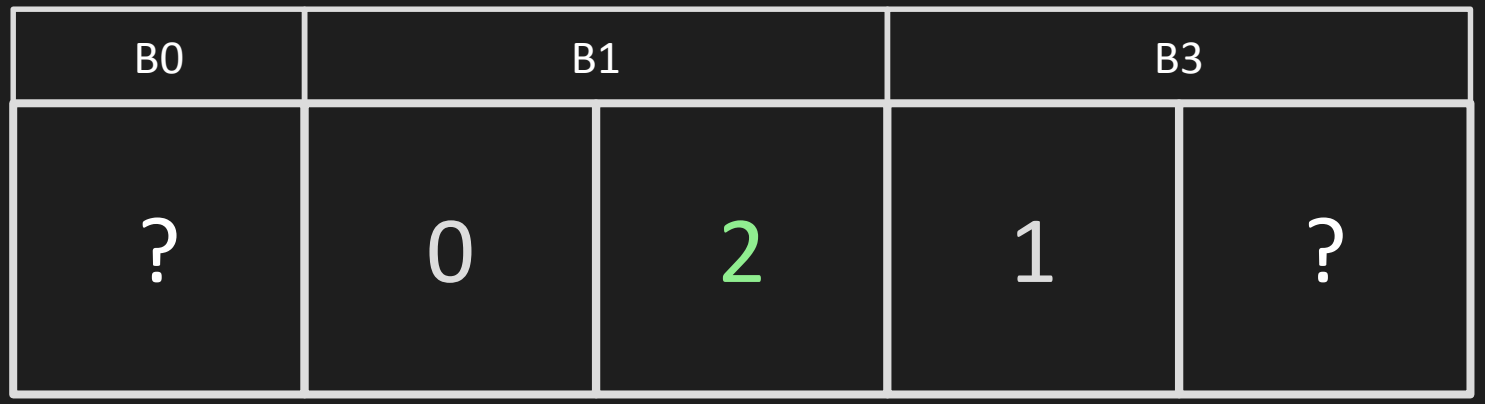
B0	B1		B3	
?	0	?	1	?

```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

Copy of offsets
[0, 2, 3, 4, 5]

3

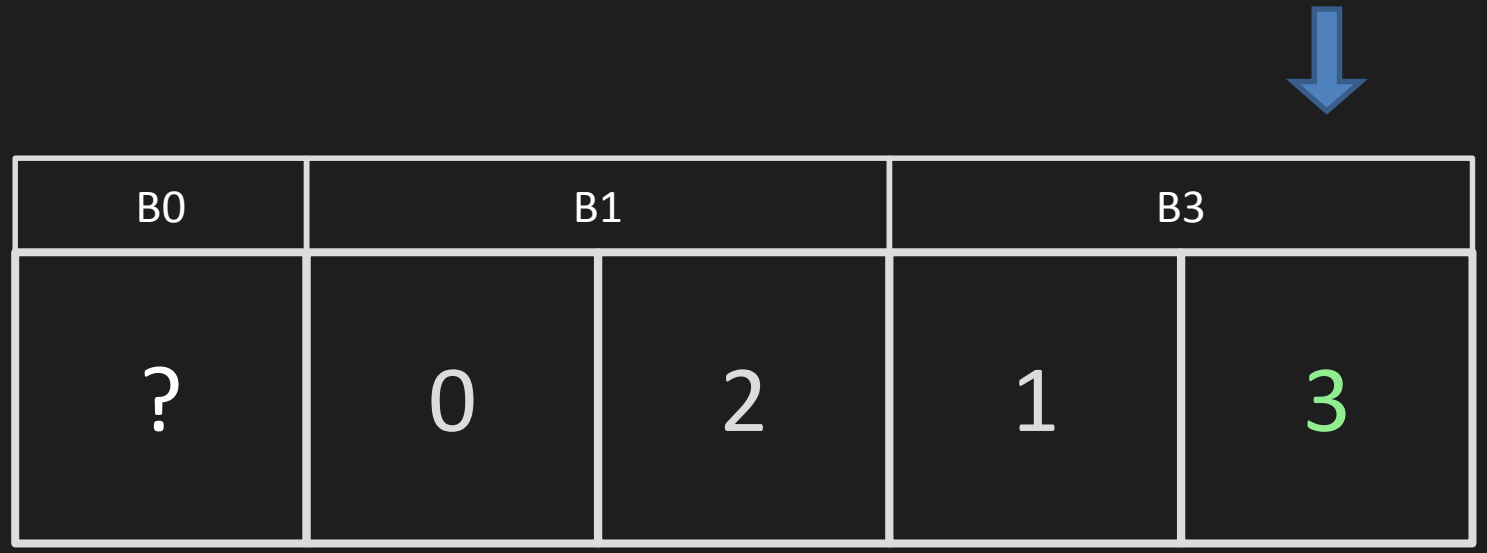
Array of indexes



```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

Copy of offsets
[0, 3, 3, 4, 5]

5

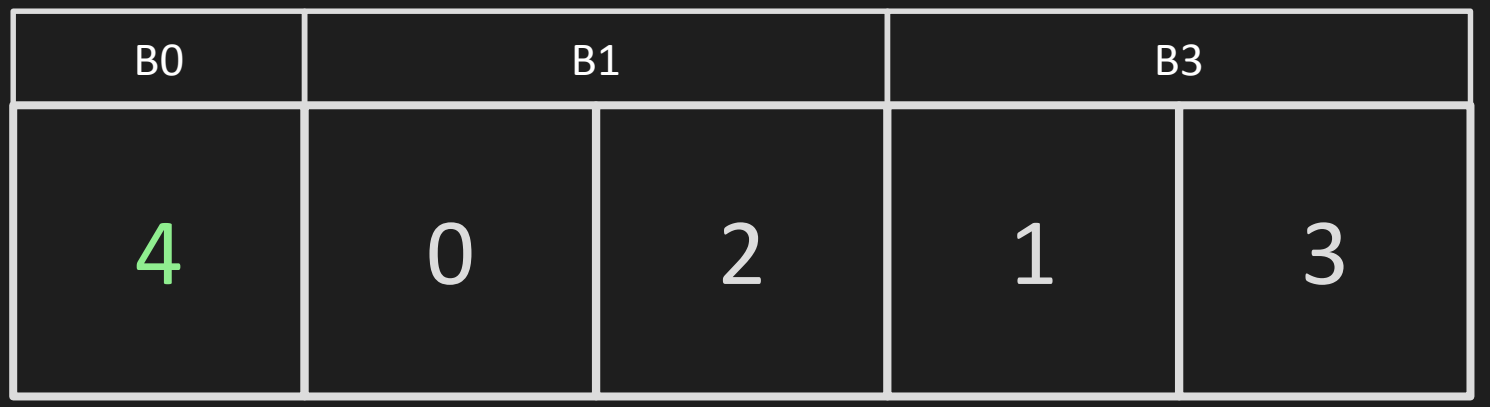


Array of indexes

```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

Copy of offsets
[0, 3, 3, 5, 5]

1



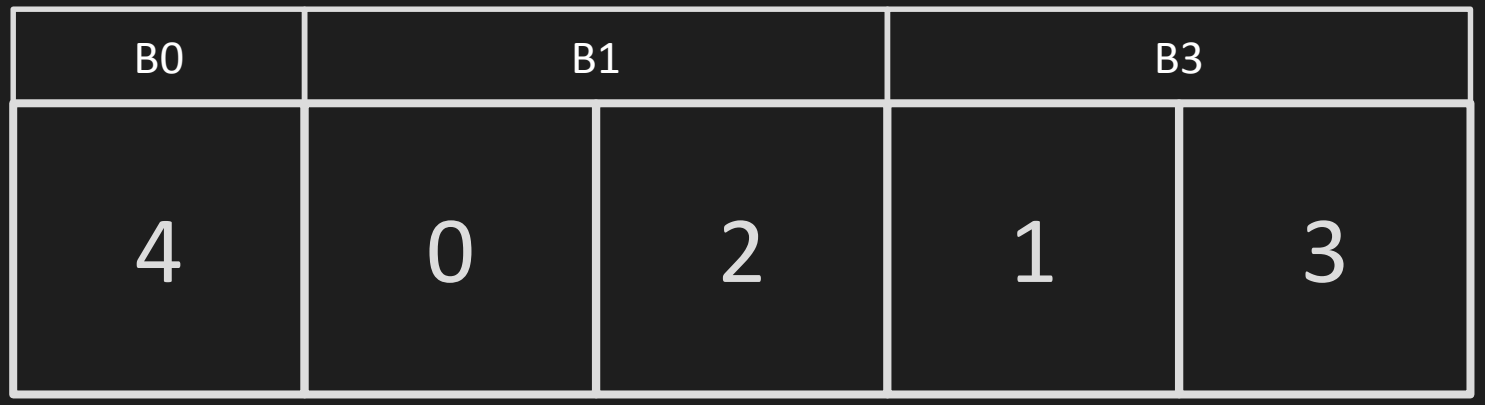
Array of indexes

```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

Original offsets
[0, 1, 3, 3, 5]

Copy of offsets
[1, 3, 3, 5, 5]

Array of indexes



```
create_table(
    pair0, // B1
    pair1, // B3
    pair2, // B1
    pair3, // B3
    pair4, // B0
)
```

Original offsets
[0, 1, 3, 3, 5]

Indexes
[4, 0, 2, 1, 3]

```
template <size_t IDX = 0, typename TPL>
constexpr void calc_indexes_impl(const TPL& tpl,
                hashmeta_t<TPL>& offs, hashmeta_t<TPL>& idxes) {
    if constexpr (tuple_size_v<TPL> > IDX) {
        const auto bucket = tuple_element_t<IDX, TPL>::HASH
                            % tuple_size_v<TPL>;
        const auto place = offs[bucket]++;
        idxes[place] = IDX;
        calc_indexes_impl<IDX + 1>(tpl, offs, idxes);
    }
}
```

```
template <typename ... ARGS>
class htbl {
public:
    constexpr htbl(ARGS&&... args) :
        m_tup(forward_as_tuple(args...)),
        m_off(calc_offsets(m_tup)),
        m_idx(calc_indexes(m_tup, m_off))
    {}

    template <typename FN>
    bool execute(const char* key, const FN& fn) const;

private:
    const tuple<ARGS...> m_tup;
    const array<size_t, sizeof...(ARGS)> m_off;
    const array<size_t, sizeof...(ARGS)> m_idx;
};
```

m_off [0, 1, 3, 3, 5] m_idx[4, 0, 2, 1, 3]

```
template <typename FN>
bool execute(const char* key, const FN& fn) const {
    const auto bucket = calc_hash(key) % tuple_size_v<TPL>;

    const auto from = m_off[bucket];
    const auto to = (bucket + 1 == tuple_size_v<TPL>) ?
                    tuple_size_v<TPL> :
                    m_off[bucket + 1];
    ...
```

m_off [0, 1, 3, 3, 5] m_idx[4, 0, 2, 1, 3]

```
template <typename FN>
bool execute(const char* key, const FN& fn) const {
    ...
    for (size_t i = from; i < to; ++i) {
        if (big_switch(m_tup, m_idx[i], key, fn) {
            return true;
        }
    }
    return false;
}
```

```
template <typename TPL, typename FN>
auto big_switch(const TPL& tpl, size_t idx, string_view key, const FN& func) {



    switch (idx) {
        case 0: if (get<0>.key == key) { func(get<0>.val); return true; }


        case 1: if (get<1>.key == key) { func(get<1>.val); return true; }


        case 2: if (get<2>.key == key) { func(get<2>.val); return true; }


        ...
    }
    return false;
}
```

```cpp
template <typename TPL, typename FN>
auto big_switch(const TPL& tpl, size_t idx, string_view key, const FN& func) {



    switch (idx) {
        case 0: if constexpr (0 < tuple_size_v<TPL>)
                if (get<0>.key == key) { func(get<0>.val); return true; }

        case 1: if constexpr (1 < tuple_size_v<TPL>)
                if (get<1>.key == key) { func(get<1>.val); return true; }

        case 2: if constexpr (2 < tuple_size_v<TPL>)
                if (get<2>.key == key) { func(get<2>.val); return true; }

        ...
    }
    return false;
}
```

```cpp
template <typename TPL, typename FN>
auto big_switch(const TPL& tpl, size_t idx, string_view key, const FN& func) {

    static_assert(tuple_size_v<TPL> > 3, "meh");

    switch (idx) {
        case 0: if constexpr (0 < tuple_size_v<TPL>)
                if (get<0>.key == key) { func(get<0>.val); return true; }

        case 1: if constexpr (1 < tuple_size_v<TPL>)
                if (get<1>.key == key) { func(get<1>.val); return true; }

        case 2: if constexpr (2 < tuple_size_v<TPL>)
                if (get<2>.key == key) { func(get<2>.val); return true; }

        ...
    }
    return false;
}
```

Token stream injection (Maybe C++23)

```
switch (idx) {
    case N: if constexpr (N < tuple_size_v<TPL>)
        if (get<N>.key == key) { ... };
    ...
}
```

Token stream injection (Maybe C++23)

```
switch (idx) {
    case N: if constexpr (N < tuple_size_v<TPL>)
        if (get<N>.key == key) { ... };
    ...
}

switch (idx) {
    constexpr {
        for (size_t N : tuple_size_v<TPL>) {
          -> { case N: ... }
        }
    }
    default: return false;
}
```

John Carmack

@ID_AA_Carmack

Following

Saw comment // NEW BOOST CODE, and had a moment of panic before realizing it was vehicle boost, not C++ boost

2:05 PM - 15 Jun 2011

136 Retweets **51** Likes

14 136 51

```cpp
template <typename TPL, typename FN>
auto big_switch(const TPL& tpl, size_t idx, string_view key, const FN& func) {

    static_assert(tuple_size_v<TPL> > 3, "meh");

    switch (idx) {
        case 0: if constexpr (0 < tuple_size_v<TPL>)
                if (get<0>.key == key) { func(get<0>.val); return true; }

        case 1: if constexpr (1 < tuple_size_v<TPL>)
                if (get<1>.key == key) { func(get<1>.val); return true; }

        case 2: if constexpr (2 < tuple_size_v<TPL>)
                if (get<2>.key == key) { func(get<2>.val); return true; }

        ...
    }
    return false;
}
```

```
template <typename TPL, typename FN>
auto big_switch(const TPL& tpl, size_t idx, string_view key, const FN& func) {

    #define MAX_SWITCHES 50
    static_assert(MAX_SWITCHES > tuple_size_v<TPL>, "booooost");

    switch (idx) {
        BOOST_PP_REPEAT(MAX_SWITCHES, NV_SWITCH, 1);
    }

    return false;
}

#define NV_SWITCH(unused, IDX, unused2)                                  \
case IDX: if constexpr (0 < tuple_size_v<TPL>)                           \
if (get<IDX>.key == key) { func(get<IDX>.val); return true; }
```

```cpp
template <size_t BEG = 0, typename TPL, typename FN>
auto big_switch(const TPL& tpl, size_t idx, string_view key, const FN& func) {
    switch (idx) {
        case BEG * 10 + 0: if constexpr (BEG * 10 + 0 < tuple_size_v<TPL>)
            { /* code for element BEG * 10 + 0; */ return true; }
        ...









    }

    return false;
}
```

```cpp
template <size_t BEG = 0, typename TPL, typename FN>
auto big_switch(const TPL& tpl, size_t idx, string_view key, const FN& func) {
    switch (idx) {
        case BEG * 10 + 0: if constexpr (BEG * 10 + 0 < tuple_size_v<TPL>)
            { /* code for element BEG * 10 + 0; */ return true; }
        ...
        // same for +1 to +8
        ...
        case BEG * 10 + 9: if constexpr (BEG * 10 + 9 < tuple_size_v<TPL>)
            { /* code for element BEG * 10 + 9; */ return true; }



    }

    return false;
}
```

```cpp
template <size_t BEG = 0, typename TPL, typename FN>
auto big_switch(const TPL& tpl, size_t idx, string_view key, const FN& func) {
    switch (idx) {
        case BEG * 10 + 0: if constexpr (BEG * 10 + 0 < tuple_size_v<TPL>)
            { /* code for element BEG * 10 + 0; */ return true; }
        ...
        // same for +1 to +8
        ...
        case BEG * 10 + 9: if constexpr (BEG * 10 + 9 < tuple_size_v<TPL>)
            { /* code for element BEG * 10 + 9; */ return true; }

        default:           if constexpr (BEG * 10 + 10 < tuple_size_v<TPL>)
            { return big_switch<BEG + 1>(tpl, idx, key, func);
    }

    return false;
}
```

Let's use heterogeneous hash-table for initial idea of string-to-func mapping!

```
void fn1(int, float);

const auto htbl tbl(
    HASH_TABLE_ENTRY("key1", &fn1),
);

vector<string> params{ "3", "5.7" };
const auto functor = [&params](auto val) {



};

tbl.execute("key1", functor);
```

```
void fn1(int, float);

const auto htbl tbl(
    HASH_TABLE_ENTRY("key1", &fn1),
);

vector<string> params{ "3", "5.7" };
const auto functor = [&params](auto val) {
    using TUP = get_fn_info<decltype(val)>::args;


};

tbl.execute("key1", functor);
```

```
template <typename RET, typename ... ARGS>
auto get_args(function<RET(ARGS...)>) {
    return tuple<decay_t<ARGS>...>{};
}
```

```cpp
template <typename RET, typename ... ARGS>
auto get_args(function<RET(ARGS...)>) {
    return tuple<decay_t<ARGS>...>{};
}

template <typename FN>
struct get_fn_info {
    using args = decltype(
        get_args(
            function{ std::declval<FN>() }
        )
    );
};
```

```
void fn1(int, float);

const auto htbl tbl(
    HASH_TABLE_ENTRY("key1", &fn1),
);

vector<string> params{ "3", "5.7" };
const auto functor = [&params](auto val) {
    using TUP = get_fn_info<decltype(val)>::args;


};

tbl.execute("key1", functor);
```

```
void fn1(int, float);

const auto htbl tbl(
    HASH_TABLE_ENTRY("key1", &fn1),
);

vector<string> params{ "3", "5.7" };
const auto functor = [&params](auto val) {
    using TUP = get_fn_info<decltype(val)>::args;
    const TUP conv = convert<TUP>(params);

};

tbl.execute("key1", functor);
```

```cpp
template <typename T>
T converter(const string& str) {
    static_assert(is_same_v<T, void>);
}

template <>
int converter<int>(const string& str) {
    return stoi(str);
}

template <>
float converter<float>(const string& str) {
    return stof(str);
}
```

```
std::make_index_sequence<N> is alias
std::index_sequence<0, 1, 2, ..., N - 1>

fn1(make_index_sequence<5>{});

template <size_t ... N>
void fn1(index_sequence<N...>) {
    accept_any(N...);
    // accept_any(0, 1, 2, ..., N - 1);
}
```

```cpp
template <typename ... ARGS>
void fn(tuple<ARGS...> tpl) {
    fn_helper(tpl,
      make_index_sequence<sizeof...(ARGS)>{});
}

template <typename TPL, size_t ... N>
void fn_helper(TPL tpl, index_sequence<N...>)  {
    accept_any(get<N>(tpl)...);
    // accept_any(get<0>(tpl), ..., get<N-1>(tpl));
}
```

```
template <typename TPL, typename CONT>
TPL converter(const CONT& cont) {
    return converter_impl<TPL>(cont,
        make_index_sequence<tuple_size_v<TPL>>{});
}
```

```
template <typename TPL, typename CONT, size_t ... N>
auto converter_impl(const CONT& cont,
                    index_sequence<N...>) {
    return make_tuple(
        convert<tuple_element_t<N, TPL>>(cont[N])...
    );
}
```

```
template <typename TPL, typename CONT, size_t ... N>
auto converter_impl(const CONT& cont,
                    index_sequence<N...>) {
    return make_tuple(
        convert<tuple_element_t<N, TPL>>(cont[N])...
    );
}

    return make_tuple(
        convert<tuple_element_t<0, TPL>>(cont[0]),
        convert<tuple_element_t<1, TPL>>(cont[1])
    )
```

```
template <typename TPL, typename CONT, size_t ... N>
auto converter_impl(const CONT& cont,
                    index_sequence<N...>) {
    return make_tuple(
        convert<tuple_element_t<N, TPL>>(cont[N])...
    );
}

    return make_tuple(
        convert<int>(cont[0]),
        convert<float>(cont[1])
    )
```

```
void fn1(int, float);

const auto htbl tbl(
    HASH_TABLE_ENTRY("key1", &fn1),
);

vector<string> params{ "3", "5.7" };
const auto functor = [&params](auto val) {
    using TUP = get_fn_info<decltype(val)>::args;
    const TUP conv = convert<TUP>(params);

};

tbl.execute("key1", functor);
```

```
void fn1(int, float);

const auto htbl tbl(
    HASH_TABLE_ENTRY("key1", &fn1),
);

vector<string> params{ "3", "5.7" };
const auto functor = [&params](auto val) {
    using TUP = get_fn_info<decltype(val)>::args;
    const TUP conv = convert<TUP>(params);
    apply(val, conv);
};

tbl.execute("key1", functor);
```

```
static const htbl tbl(
   HASH_TABLE_ENTRY("hello",
      htbl(HASH_TABLE_ENTRY("world", &func))
   )
);
```

Nested hash tables can be used for multi-word commands like "hello world arg1 arg2"

```
static const htbl tbl(
   HASH_TABLE_ENTRY("hello",
      htbl(HASH_TABLE_ENTRY("world", &func))
   )
);
```

Nested hash tables can be used for multi-word commands like "hello world arg1 arg2"

```
struct functor {
    template <typename ... ARGS>
    bool operator()(const htbl<ARGS...>& tbl) const ( ... );

    template <typename FN>
    bool operator()(const FN& fn) const ( ... );
}
```

THE END

github.com/starl1ght/italy18

starl1ght@mail.ru