www.italiancpp.org

Zero or More

Alberto Barbati

C++ Day 2025
October 25, Pavia

The Problem

Write a function that returns an arbitrary number of elements

We are interested in:

- how the function is declared
- how the declaration affects the function implementation and its usage
- how the C++ language, in its evolution, shapes the viable approaches and introduces new ones

Fine prints

- All returned elements have the same type
- The type, called `element_t`, is copiable and copies are cheap
- `element_t` has value semantic

In the examples, we will use this placeholder pseudo-function representing some kind of abstract processing we want to do with the obtained elements

```
void PROCESS_ELEMENT(element_t elem);
```

C90: Basic approaches

We'll start by looking a few different approaches that were available and used in C, thus before the advent of C++

Get one element at a time

.h
```
bool get_element(element_t* elem);
```

.c
```
element_t elem
while (get_element(&elem))
{
    PROCESS_ELEMENT(elem);
}
```

Considerations

- Most probably, we will need to pass additional arguments to `get_element` to provide context, those are not depicted for simplicity
- Similary, we might need to have start/cleanup functions
- `get_element` may be inefficient if the number of elements in very large
- The caller can stop looping as soon as it receieve enough elements, if the process doesn't need all of them
- Processing can start as soon as elements are available

Get elements using a caller-allocated buffer

```h
int get_elements(element_t* elems, int max_size);
```

```c
element_t elems[MAX_SIZE];

int num_elems = get_elements(elems, MAX_SIZE);

for (int i = 0; i < num_elems; ++i)
{
    PROCESS_ELEMENT(elems[i]);
}
```

Considerations

- `get_elements` is usually called in a loop, to handle the case wher the buffer it too small to accommodate all available elements
- The caller can stop looping as soon as it receieve enough elements, if the process doesn't need all of them
- Processing can start as soon as some elements are available
- It may be difficult to determine the optimal value for `MAX_SIZE`
- Bad thing may happen if you pass `get_element` the wrong size

Get all elements using a callee-allocated buffer

.h

```
element_t* get_all_elements(int* num);
void free_elements(element_t* elems);
```

.c

```
int num_elems;
element_t* elems = get_all_elements(&num_elems);

for (int i = 0; i < num_elems; ++i)
{
    PROCESS_ELEMENT(elems[i]);
}

free_elements(elems);
```

Considerations

- `get_all_elements` gets all elements with just one call + cleanup
- You need to remember to call the free function
- You always get all elements even if you wanted a few of them
- Processing can start only once all elements are retrieved
- Presumes you have enough memory to accommodate all elemens
- Dynamic allocation can be slow

"For each" approach

.h

```h
typedef void (*process_func_t)(element_t);
void foreach_element(process_func_t process_func);
```

.c

```c
void process_function(element_t elem)
{
    PROCESS_ELEMENT(elem);
}

/* ... */

foreach_element(process_function);
```

Considerations

- The iteration is provided by the function itself, not by the caller
- The API can get more complicated if you need to pass information or state to `process_function`
- The code that processes the elements has be put in a separate function, with that exact signature
- Processing can start as soon as elements are available
- Must process all elements (but...)

C++98 - A whole new language

C++98 introduces several features that changes everything

- classes, methods, constructors and destructors
- operator overload
- templates
- the C++ standard library

Get one element at a time

```cpp
generator_t gen(/* arguments */);

element_t elem;

while (gen(elem))
{
    PROCESS_ELEMENT(elem);
}
```

Possible implementation

.h

```
class generator_t
{
    // state

public:
    generator_t(/* parameters */);
    ~generator_t();

    bool operator()(element_t& elem);
}
```

Considerations

- There's a clear RAII API for starting and finishing the iteration
- No names, except the name of the class
- You can potentially use the API twice, even at the same time

Get all elements using a container

.h
```
void get_all_elements(std::vector<element_t>& elems);
```

.cpp
```
std::vector<element_t> elems;

get_all_elements(elems)

// memory freed by std::vector dtor
```

Possibile implementation

.cpp

```cpp
void get_all_elements(std::vector<element_t>& elements)
{
    // elements.clear();
    // elements.reserve(...);

    for (/* iteration */)
    {
        element_t elem = /* compute elem value */;
        elements.push_back(elem);
    }
}
```

Considerations

- `get_all_elements` can resize the container if the number of elements is known in advance, reducing the number of dynamic allocations
- You need to consider the previous state of the container: since both the "append" and the "clear+append" are reasonable, you have to make a choice and document it
- You lock `std::vector` as the only possible container

"For each" approach with templates

.h

```
template <typename Func>
void foreach_element(Func func)
{
    for (/* iteration */)
    {
        const element_t elem = /* compute elem value */;
        func(elem);
    }
}
```

Possible usage

.cpp

```cpp
struct processor_t
{
    void operator()(element_t elem)
    {
        PROCESS_ELEMENT(elem);
    }
};

// ...

foreach_element(processor_t());
```

Considerations

- Similar to the C90 "for each" approach, but now the processor can store state in its data members
- Code that process the elements is separated from the call to `foreach_element`
- It's a template!
- Compiler may be able to inline the call and provide efficient code

C++11: C++ becomes "Modern"

C++11 introduces a few new features:

- Move semantic
- Lambda expressions
- `std::function`
- Range-based for loop

"Get all elements" with move semantic

.h

```
std::vector<element_t> get_all_elements();
```

.cpp

```
// simple assignment
auto elems = get_all_elements();

// with range-based for
for (auto elem : get_all_elements())
{
    PROCESS_ELEMENT(elem);
}
```

Considerations

- There's no "previous state" of the container to worry about
- The new API is more terse and easier to combine with other constructs

"For each" with lambda expressions

Using the same `foreach_element` function we saw earlier, we can now write

```cpp
foreach_element([/* captures */](element_t elem)
{
    PROCESS_ELEMENT(elem);
});
```

Non-template "For each" with `std::function`

You can now write the `foreach_element` as a non-template function, by using `std::function` as the type of the argument

```
void foreach_element(std::function<void(element_t)> func);
```
.h

Considerations

- Implementation can be moved into a .cpp file
- `std::function` may introduce a little overhead, especially if it's used with a function object that has a state, since it may need to allocate dynamic memory
- The call to the payload function cannot be inlined

C++20 brings new tools

The C++20 features that are relevant to our problems are:

- `std::span`
- Concepts
- Ranges

"Get elements" with `std::span`

.h

```h
size_t get_elements(std::span<element_t> elems);
```

.cpp

```cpp
element_t c_array[42];
std::array<element_t, 42> array;
std::vector<element_t> vector{42};

get_elements(c_array);
get_elements(array);
get_elements(vector);
```

Considerations

- Safer API, since you cannot by mistake pass the wrong size
- One function works for several container types

"For each" with concepts and invoke

.h

```
template <std::invocable<element_t> Func>
void foreach_element(Func f)
{
    for (/* iteration */)
    {
        element_t elem = /* compute elem value */;
        std::invoke(f, elem);
    }
}
```

Considerations

- Improved diagnostics
- Works with pointer to member functions too

"Get all elements" by returning a range

.h

```
inline std::ranges::range<element_t> auto get_all_elements()
{
    /* ... */
    return /* any expression that satisfies the range concept,
        for example: any standard container (not just vector),
        or any standard range adaptor (view) */
}
```

Range example

```h
inline std::ranges::range<element_t> auto get_special_elements()
{
    return get_all_elements() | std::views::filter(is_special);
}
```

Considerations

- It may be more efficient if you already have elements stored in a container and you are just returning it as-is or applying some lazy range adapter, since you avoid creating an intermediate `std::vector` to hold the results

C++23 Generators

```cpp
std::generator<element_t> get_all_elements()
{
    for (/* iteration */)
    {
        element_t elem = /* compute elem value */;
        co_yield elem;
    }
}
```

Considerations

- It's NOT a template
- It's easy to write and easy to use
- There can be small overhead introduced by the coroutine
- The may potentially be one dynamic allocation, with a predictable size, regardless of the number of elements

Non-template "For each": new types

```
void foreach_elem(std::function<void(element_t)> f);
```
C++11

```
void foreach_elem(std::move_only_function<void(element_t)> f);
```
C++23

```
void foreach_elem(std::function_ref<void(element_t)> f);
```
C++26

Bonus slides: "For each" with early break in C++20

```
// process all elements
foreach_element([](elmement_t elem)
{
    PROCESS_ELEMENT(element)
});
```

```
// process some elements
foreach_element([](elmement_t elem)
{
    PROCESS_ELEMENT(element)
    return /* test */ ? iteration::go_on : iteration::stop;
});
```

Bonus slides: "For each" with early break in C++20

```
template <typename Func>
void foreach_element(Func f)
{
    for (/* iteration */)
    {
        element_t elem = /* compute elem value */;
        if (should_stop_after_invoke(f, elem))
        {
            break;
        }
    }
}
```

Bonus slides: "For each" with early break in C++20

```cpp
enum class iteration { go_on, stop };

template <typename Func, typename Elem> requires std::invocable<Func, Elem>
bool should_stop_after_invoke(Func&& f, Elem&& e)
{
    if constexpr (std::is_invocable_r_v<iteration, Func, Elem>)
    {
        return std::invoke_r<iteration>(
            std::forward<Func>(f), std::forward<Elem>(e)) == iteration::stop;
    }
    else
    {
        std::invoke(std::forward<Func>(f), std::forward<Elem>(e));
        return false;
    }
}
```

Thanks for your attention