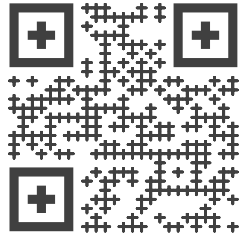
Italian C++ Community

www.italiancpp.org



Sharing memory

using non-virtual virtual inheritance

📧 mrcmnn@gmail.com | 🇮🇹 Italian C++ Conference | 🗓️ 10 June 2023 | 🏛️ Rome

Managing memory is hard

Managing memory is hard

memory is cheap

Managing memory is hard

memory is cheap

until it isn’t

Embedded systems

Embedded systems

- Embedded Template Library

Embedded systems

- Embedded Template Library
- mpack

Embedded systems

- Embedded Template Library
- mpack
- Arduino

```cpp
char buffer[FIXED_SIZE] = {};
std::array<char, FIXED_SIZE> buffer;
```

```cpp
char buffer[FIXED_SIZE] = {};
std::array<char, FIXED_SIZE> buffer;

char *buffer = (char*)malloc(size);
char *buffer = new char[size];
```

```cpp
char buffer[FIXED_SIZE] = {};
std::array<char, FIXED_SIZE> buffer;

char *buffer = (char*)malloc(size);
char *buffer = new char[size];

auto buffer = std::make_unique<char[]>(size);
auto buffer = std::make_shared<char[]>(size);
```

```cpp
char buffer[FIXED_SIZE] = {};
std::array<char, FIXED_SIZE> buffer;

char *buffer = (char*)malloc(size);
char *buffer = new char[size];

auto buffer = std::make_unique<char[]>(size);
auto buffer = std::make_shared<char[]>(size);

std::vector<char> buffer;
std::string buffer;
```

🌋 Too many ways 🌋

```
char buffer[FIXED_SIZE] = {};
std::array<char, FIXED_SIZE> buffer;

char *buffer = (char*)malloc(size);
char *buffer = new char[size];

auto buffer = std::make_unique<char[]>(size);
auto buffer = std::make_shared<char[]>(size);

std::vector<char> buffer;
std::string buffer;
```

 one type to rule them all 

💍 one type to rule them all 🧙

C++20's `std::span`

C++20's `std::span`

```
void whatever_algorithm(std::span<char> buffer);
```

C++20's `std::span`

```cpp
void whatever_algorithm(std::span<char> buffer);

char buffer[FIXED_SIZE] = {};
whatever_algorithm(buffer);
```

C++20's `std::span`

```cpp
void whatever_algorithm(std::span<char> buffer);

char buffer[FIXED_SIZE] = {};
whatever_algorithm(buffer);

std::array<char, FIXED_SIZE> buffer;
whatever_algorithm(buffer);
...
```

C++20's std::span

```
...
char *buffer = (char*)malloc(size);
whatever_algorithm({ buffer, size });
```

C++20's std::span

```
...
char *buffer = (char*)malloc(size);
whatever_algorithm({ buffer, size });

char *buffer = new char[size];
whatever_algorithm({ buffer, size });
```

C++20's std::span

```
...
char *buffer = (char*)malloc(size);
whatever_algorithm({ buffer, size });

char *buffer = new char[size];
whatever_algorithm({ buffer, size });

auto buffer = std::make_unique<char[]>(size);
whatever_algorithm({ buffer.get(), size });
```

C++20's std::span

```
...
char *buffer = (char*)malloc(size);
whatever_algorithm({ buffer, size });

char *buffer = new char[size];
whatever_algorithm({ buffer, size });

auto buffer = std::make_unique<char[]>(size);
whatever_algorithm({ buffer.get(), size });

auto buffer = std::make_shared<char[]>(size);
whatever_algorithm({ buffer.get(), size });
...
```

C++20's `std::span`

```
...
std::vector<char> buffer;
whatever_algorithm(buffer);
```

C++20's std::span

```
...
std::vector<char> buffer;
whatever_algorithm(buffer);

std::string buffer;
whatever_algorithm(buffer);
```

Recap

Recap

- Non-owning

Recap

- Non-owning
- Contiguous memory

Recap

- Non-owning
- Contiguous memory
- *Can be used in constexpr context*

Generating constants

Generating constants

```
constexpr auto constant_signal =
```

Generating constants

```
constexpr auto constant_signal = make_constexpr_array<float, 100>(                );
```

Generating constants

```
constexpr auto constant_signal = make_constexpr_array<float, 100>(whatever_function);
```

Generating constants

```
constexpr std::array<    > make_constexpr_array(     ) {



}

constexpr auto constant_signal = make_constexpr_array<float, 100>(whatever_function);
```

Generating constants

```
template <typename T, std::size_t S            >
constexpr std::array<    > make_constexpr_array(     ) {



}

constexpr auto constant_signal = make_constexpr_array<float, 100>(whatever_function);
```

Generating constants

```
template <typename T, std::size_t S            >
constexpr std::array<T, S> make_constexpr_array(     ) {



}

constexpr auto constant_signal = make_constexpr_array<float, 100>(whatever_function);
```

Generating constants

```
template <typename T, std::size_t S            >
constexpr std::array<T, S> make_constexpr_array(     ) {
    std::array<T, S> result;

    return result;
}

constexpr auto constant_signal = make_constexpr_array<float, 100>(whatever_function);
```

Generating constants

```
template <typename T, std::size_t S, typename F>
constexpr std::array<T, S> make_constexpr_array(     ) {
    std::array<T, S> result;

    return result;
}

constexpr auto constant_signal = make_constexpr_array<float, 100>(whatever_function);
```

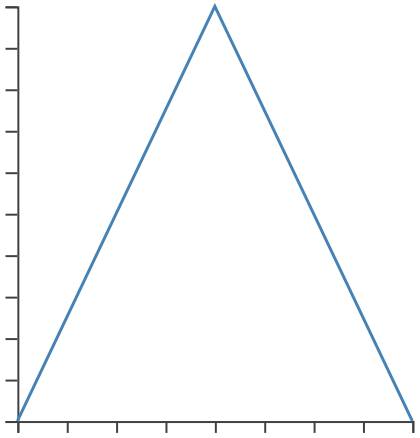
Generating constants

```
template <typename T, std::size_t S, typename F>
constexpr std::array<T, S> make_constexpr_array(F&& f) {
    std::array<T, S> result;

    return result;
}

constexpr auto constant_signal = make_constexpr_array<float, 100>(whatever_function);
```

Generating constants

```
template <typename T, std::size_t S, typename F>
constexpr std::array<T, S> make_constexpr_array(F&& f) {
    std::array<T, S> result;
    f(std::span(result));
    return result;
}

constexpr auto constant_signal = make_constexpr_array<float, 100>(whatever_function);
```

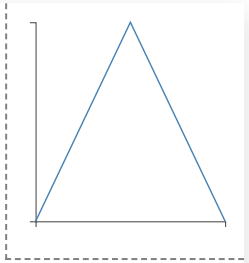
Generating constants



bartlett window

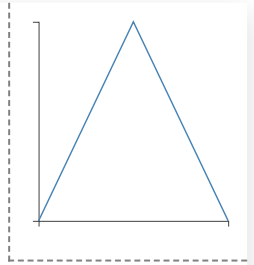
Generating constants

```
constexpr void bartlett(std::span<float> output) const {




}
```



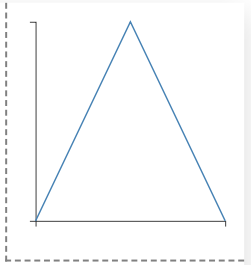
Generating constants

```
constexpr void bartlett(std::span<float> output) const {
    const auto N = output.size() / 2;




}
```



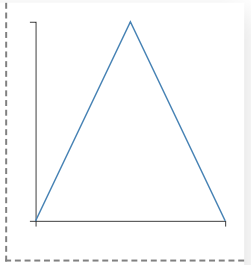
Generating constants

```
constexpr void bartlett(std::span<float> output) const {
    const auto N = output.size() / 2;
    for (std::size_t i = 0; i < N; i++) {
        output[i] = (float)i / N;
    }



}
```



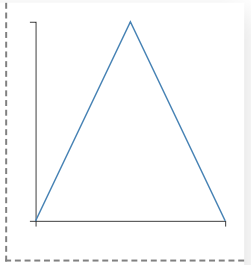
Generating constants

```
constexpr void bartlett(std::span<float> output) const {
    const auto N = output.size() / 2;
    for (std::size_t i = 0; i < N; i++) {
        output[i] = (float)i / N;
    }

    for (std::size_t i = N; i < output.size(); i++) {
        output[i] = (float)(output.size() - i - 1) / N;
    }
}
```



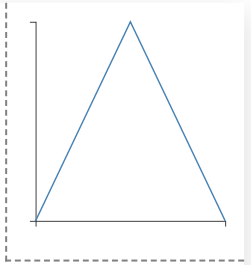
Generating constants

```
constexpr void bartlett(std::span<float> output) const {
    const auto N = output.size() / 2;
    for (std::size_t i = 0; i < N; i++) {
        output[i] = (float)i / N;
    }

    for (std::size_t i = N; i < output.size(); i++) {
        output[i] = (float)(output.size() - i - 1) / N;
    }
}
constexpr auto weights =
```

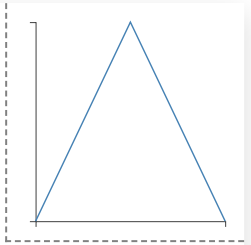


Generating constants

```
constexpr void bartlett(std::span<float> output) const {
    const auto N = output.size() / 2;
    for (std::size_t i = 0; i < N; i++) {
        output[i] = (float)i / N;
    }

    for (std::size_t i = N; i < output.size(); i++) {
        output[i] = (float)(output.size() - i - 1) / N;
    }
}
constexpr auto weights = make_constexpr_array<float, 10>(bartlett);
```



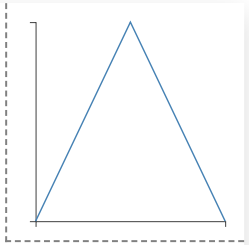
Generating constants



```
constexpr auto weights = make_constexpr_array<float, 10>(        );
```

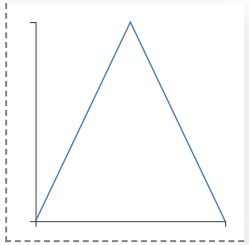
Generating constants

```
struct bartlett_t {









};


constexpr auto weights = make_constexpr_array<float, 10>(        );
```



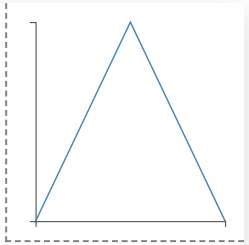
Generating constants

```
struct bartlett_t {

    constexpr void operator()(                          ) const {









    }
};


constexpr auto weights = make_constexpr_array<float, 10>(         );
```



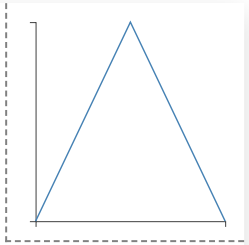
Generating constants

```
struct bartlett_t {
    template <typename T, std::size_t S>
    constexpr void operator()(                          ) const {









    }
};


constexpr auto weights = make_constexpr_array<float, 10>(         );
```



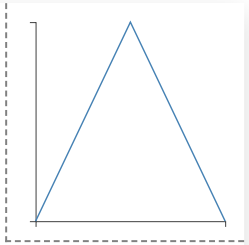
Generating constants

```
struct bartlett_t {
    template <typename T, std::size_t S>
    constexpr void operator()(std::span<T, S> output) const {









    }
};


constexpr auto weights = make_constexpr_array<float, 10>(        );
```



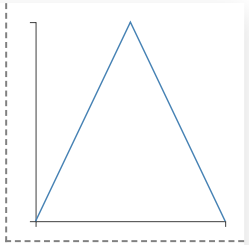
Generating constants

```
struct bartlett_t {
    template <typename T, std::size_t S>
    constexpr void operator()(std::span<T, S> output) const {
        const auto N = output.size() / 2;









    }
};


constexpr auto weights = make_constexpr_array<float, 10>(        );
```



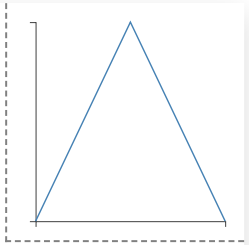
Generating constants

```
struct bartlett_t {
    template <typename T, std::size_t S>
    constexpr void operator()(std::span<T, S> output) const {
        const auto N = output.size() / 2;
        for (std::size_t i = 0; i < output.size() / 2; i++) {
            output[i] = T(i) / N;
        }




    }
};


constexpr auto weights = make_constexpr_array<float, 10>(        );
```



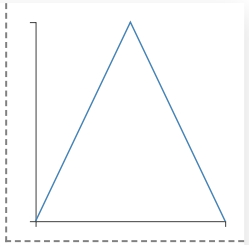
Generating constants

```
struct bartlett_t {
    template <typename T, std::size_t S>
    constexpr void operator()(std::span<T, S> output) const {
        const auto N = output.size() / 2;
        for (std::size_t i = 0; i < output.size() / 2; i++) {
            output[i] = T(i) / N;
        }

        for (std::size_t i = N; i < output.size(); i++) {
            output[i] = T(output.size() - i - 1) / N;
        }
    }
};


constexpr auto weights = make_constexpr_array<float, 10>(        );
```



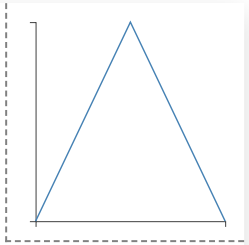
Generating constants

```
struct bartlett_t {
    template <typename T, std::size_t S>
    constexpr void operator()(std::span<T, S> output) const {
        const auto N = output.size() / 2;
        for (std::size_t i = 0; i < output.size() / 2; i++) {
            output[i] = T(i) / N;
        }

        for (std::size_t i = N; i < output.size(); i++) {
            output[i] = T(output.size() - i - 1) / N;
        }
    }
};
constexpr bartlett_t bartlett;

constexpr auto weights = make_constexpr_array<float, 10>(        );
```



Generating constants

```cpp
struct bartlett_t {
    template <typename T, std::size_t S>
    constexpr void operator()(std::span<T, S> output) const {
        const auto N = output.size() / 2;
        for (std::size_t i = 0; i < output.size() / 2; i++) {
            output[i] = T(i) / N;
        }

        for (std::size_t i = N; i < output.size(); i++) {
            output[i] = T(output.size() - i - 1) / N;
        }
    }
};
constexpr bartlett_t bartlett;

constexpr auto weights = make_constexpr_array<float, 10>(bartlett);
```



Ownership

Ownership

An example

Ownership

An example

```
bartlett_window(in, out);
```

Ownership

An example

```
constexpr void bartlett_window(
                                                ) {




}

bartlett_window(in, out);
```

Ownership

An example

```
template <typename T, std::size_t S>
constexpr void bartlett_window(
                                                ) {




}

bartlett_window(in, out);
```

Ownership

An example

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {




}

bartlett_window(in, out);
```

Ownership

An example

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {

    for (std::size_t i = 0; i < size; i++) {

    }
}

bartlett_window(in, out);
```

Ownership

An example

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {

    for (std::size_t i = 0; i < size; i++) {
        out[i] = weights[i] * in[i];
    }
}

bartlett_window(in, out);
```

Ownership

An example

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {
    auto weights = /* ??? */;
    for (std::size_t i = 0; i < size; i++) {
        out[i] = weights[i] * in[i];
    }
}

bartlett_window(in, out);
```

Forcing a choice

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {




    auto weights =                                      ;
    for (std::size_t i = 0; i < size; i++) {
        out[i] = weights[i] * in[i];
    }

}
```

Forcing a choice

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {

    T* storage = new T[    ]();


    auto weights =                                      ;
    for (std::size_t i = 0; i < size; i++) {
        out[i] = weights[i] * in[i];
    }
    delete[] storage;
}
```

Forcing a choice

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {
    constexpr auto size = in.size();
    T* storage = new T[size]();


    auto weights =                                      ;
    for (std::size_t i = 0; i < size; i++) {
        out[i] = weights[i] * in[i];
    }
    delete[] storage;
}
```

Forcing a choice

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {
    constexpr auto size = in.size();
    T* storage = new T[size]();
    bartlett(std::span<T, S>(storage, size));

    auto weights =                                      ;
    for (std::size_t i = 0; i < size; i++) {
        out[i] = weights[i] * in[i];
    }
    delete[] storage;
}
```

Forcing a choice

```
template <typename T, std::size_t S>
constexpr void bartlett_window(std::span<const T, S> in,
                               std::span<T, S> out) {
    constexpr auto size = in.size();
    T* storage = new T[size]();
    bartlett(std::span<T, S>(storage, size));

    auto weights = std::span<const T, S>(storage, size);
    for (std::size_t i = 0; i < size; i++) {
        out[i] = weights[i] * in[i];
    }
    delete[] storage;
}
```

Buck passing

```
template <typename T, std::size_t S>
constexpr void window(
                      std::span<const T, S> in,
                      std::span<T, S> out) {
    for (int i = 0; i < in.size(); i++) {
        out[i] = weights[i] * in[i];
    }
}
```

Buck passing

```
template <typename T, std::size_t S>
constexpr void window(std::span<const T, S> weights,
                      std::span<const T, S> in,
                      std::span<T, S> out) {
    for (int i = 0; i < in.size(); i++) {
        out[i] = weights[i] * in[i];
    }
}
```

Buck passing

```
window(          input, output);
```

Buck passing

```
constexpr auto weights =

window(         input, output);
```

Buck passing

```
constexpr auto weights = make_constexpr_array<float, 1024>(bartlett);

window(         input, output);
```

Buck passing

```
constexpr auto weights = make_constexpr_array<float, 1024>(bartlett);

window(weights, input, output);
```

Buck passing

```
void run_algorithm(std::size_t size) {



}
```

Buck passing

```
void run_algorithm(std::size_t size) {



    window(         input, output);
}
```

Buck passing

```
void run_algorithm(std::size_t size) {
    std::vector<float> weights(size);


    window(         input, output);
}
```

Buck passing

```
void run_algorithm(std::size_t size) {
    std::vector<float> weights(size);
    bartlett(weights);

    window(         input, output);
}
```

Buck passing

```
void run_algorithm(std::size_t size) {
    std::vector<float> weights(size);
    bartlett(weights);

    window(weights, input, output);
}
```

The user should not be dealing with implementation details

Hiding implementation

```
struct window_t {









};
```

Hiding implementation

```
struct window_t {




    constexpr void operator()(
                                                ) const {



    }
};
```

Hiding implementation

```
template <typename T, std::size_t S                          >
struct window_t {









    constexpr void operator()(
                                                  ) const {



    }
};
```

Hiding implementation

```
template <typename T, std::size_t S                          >
struct window_t {









    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) const {



    }
};
```

Hiding implementation

```
template <typename T, std::size_t S                          >
struct window_t {
    T weights[S];









    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) const {



    }
};
```

Hiding implementation

```
template <typename T, std::size_t S                          >
struct window_t {
    T weights[S];









    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) const {
        for (int i = 0; i < in.size(); i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T, std::size_t S                          >
struct window_t {
    T weights[S];

    constexpr window_t(      ) {

    }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) const {
        for (int i = 0; i < in.size(); i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T, std::size_t S                          >
struct window_t {
    T weights[S];

    constexpr window_t(auto f) {

    }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) const {
        for (int i = 0; i < in.size(); i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T, std::size_t S                          >
struct window_t {
    T weights[S];

    constexpr window_t(auto f) {
        f(                   );
    }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) const {
        for (int i = 0; i < in.size(); i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T, std::size_t S                          >
struct window_t {
    T weights[S];

    constexpr window_t(auto f) {
        f(std::span(weights));
    }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) const {
        for (int i = 0; i < in.size(); i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T, std::size_t S = std::dynamic_extent>
struct window_t {
    T weights[S];

    constexpr window_t(auto f) {
        f(std::span(weights));
    }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) const {
        for (int i = 0; i < in.size(); i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T>
struct window_t<T, std::dynamic_extent> {
                          weights;









    void operator()(std::span<const T> in, std::span<T> out) const {
        for (int i = 0; i < size; i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T>
struct window_t<T, std::dynamic_extent> {
    std::unique_ptr<T[]> weights;









    void operator()(std::span<const T> in, std::span<T> out) const {
        for (int i = 0; i < size; i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T>
struct window_t<T, std::dynamic_extent> {
    std::unique_ptr<T[]> weights;
    std::size_t size;









    void operator()(std::span<const T> in, std::span<T> out) const {
        for (int i = 0; i < size; i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T>
struct window_t<T, std::dynamic_extent> {
    std::unique_ptr<T[]> weights;
    std::size_t size;

    inline void reset(auto f, std::size_t new_size)  {



    }

    void operator()(std::span<const T> in, std::span<T> out) const {
        for (int i = 0; i < size; i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T>
struct window_t<T, std::dynamic_extent> {
    std::unique_ptr<T[]> weights;
    std::size_t size;

    inline void reset(auto f, std::size_t new_size)  {
        weights.reset(new T[size]);
        size = new_size;

    }

    void operator()(std::span<const T> in, std::span<T> out) const {
        for (int i = 0; i < size; i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
template <typename T>
struct window_t<T, std::dynamic_extent> {
    std::unique_ptr<T[]> weights;
    std::size_t size;

    inline void reset(auto f, std::size_t new_size)  {
        weights.reset(new T[size]);
        size = new_size;
        f(std::span(weights.get(), size));
    }

    void operator()(std::span<const T> in, std::span<T> out) const {
        for (int i = 0; i < size; i++) {
            out[i] = weights[i] * in[i];
        }
    }
};
```

Hiding implementation

```
void run() {




}
```

Hiding implementation

```
void run() {
    float input[1024],
        output[1024];




}
```

Hiding implementation

```
void run() {
    float input[1024],
        output[1024];

    for (;;) {



    }
}
```

Hiding implementation

```
void run() {
    float input[1024],
        output[1024];

    for (;;) {
        read_samples(input);

        write_samples(output);
    }
}
```

Hiding implementation

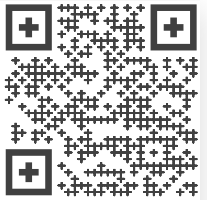
```
constexpr window_t<float, 1024> window(bartlett);

void run() {
    float input[1024],
        output[1024];

    for (;;) {
        read_samples(input);

        write_samples(output);
    }
}
```

Hiding implementation

```
constexpr window_t<float, 1024> window(bartlett);

void run() {
    float input[1024],
        output[1024];

    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```

Hiding implementation

```
constexpr window_t<float, 1024> window(bartlett);

void run() {
    float input[1024],
        output[1024];

    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```



```
window:
    .long   0
    .long   989855744
    ...
run():
    ...
.L2:
    movaps  xmm1, XMMWORD PTR [rsp+16+rax]
    movaps  xmm0, XMMWORD PTR window[rax]
    mulps   xmm0, xmm1
    movaps  XMMWORD PTR [rsp], xmm1
    movaps  XMMWORD PTR [rsp+4112+rax], xmm0
    add     rax, 16
    cmp     rax, 4096
    jne     .L2
    ...
```

Hiding implementation

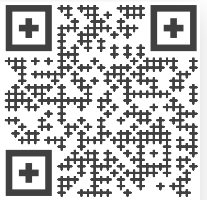
```
void run() {


    float input[1024],
        output[1024];
    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```

Hiding implementation

```
void run() {
    window_t<float, 1024> window(bartlett);

    float input[1024],
        output[1024];
    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```

Hiding implementation

```
void run() {
    window_t<float, 1024> window(bartlett);

    float input[1024],
        output[1024];
    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```



```
run():
    sub     rsp, 12312
    lea     rdi, [rsp+16]
    call    void bartlett_t::operator()<float, 1024>
    ...
void bartlett_t::operator()<float, 1024>:
    ...
```

Hiding implementation

```
void run(size_t size) {




    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```

Hiding implementation

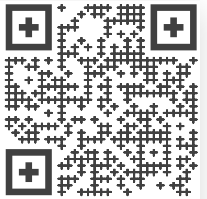
```
void run(size_t size) {



    std::vector<float> input(size),
        output(size);
    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```

Hiding implementation

```
void run(size_t size) {
    window_t<float> window;
    window.reset(bartlett, size);

    std::vector<float> input(size),
        output(size);
    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```

Hiding implementation

```
void run(size_t size) {
    window_t<float> window;
    window.reset(bartlett, size);

    std::vector<float> input(size),
        output(size);
    for (;;) {
        read_samples(input);
        window(input, output);
        write_samples(output);
    }
}
```



```
run(unsigned long):
    ...
    call    operator new[](unsigned long)
    mov     rdi, rax
    mov     rsi, r14
    mov     r12, rax
    call    void bartlett_t::operator()<float, ...>
    ...
```

Combining algorithms

Combining algorithms

```
struct windowed_spectrum_t {




};
```

Combining algorithms

```
struct windowed_spectrum_t {




    constexpr void operator()(
                                                ) {


    }
};
```

Combining algorithms

```
template <typename T, std::size_t S>
struct windowed_spectrum_t {









    constexpr void operator()(
                                                ) {



    }
};
```

Combining algorithms

```
template <typename T, std::size_t S>
struct windowed_spectrum_t {









    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {


    }
};
```

Combining algorithms

```
template <typename T, std::size_t S>
struct windowed_spectrum_t {
    window_t<T, S> window;









    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {


    }
};
```

Combining algorithms

```
template <typename T, std::size_t S>
struct windowed_spectrum_t {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;




    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {


    }
};
```

Combining algorithms

```
template <typename T, std::size_t S>
struct windowed_spectrum_t {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;


    windowed_spectrum_t(auto fn) : window(fn) { }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {



    }
};
```

Combining algorithms

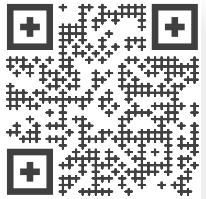
```
template <typename T, std::size_t S>
struct windowed_spectrum_t {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;


    windowed_spectrum_t(auto fn) : window(fn) { }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {
        window(in,     );

    }
};
```

Combining algorithms

```
template <typename T, std::size_t S>
struct windowed_spectrum_t {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;


    windowed_spectrum_t(auto fn) : window(fn) { }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {
        window(in,     );
        spectrum(    , out);
    }
};
```

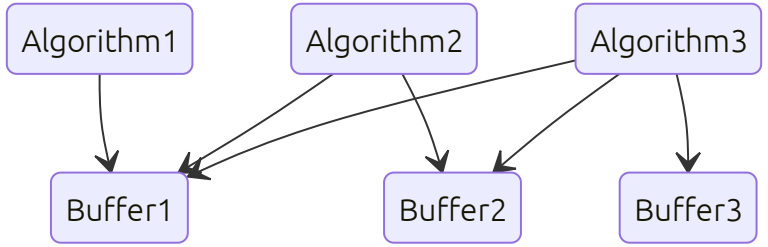
Combining algorithms

```cpp
template <typename T, std::size_t S>
struct windowed_spectrum_t {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;
    T temp[S]; // <-- A temporary buffer is used as a bridge!

    windowed_spectrum_t(auto fn) : window(fn) { }

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {
        window(in, temp);
        spectrum(temp, out);
    }
};
```



Temporary buffers should be shared

Temporary buffers should be shared



Another algorithm

Another algorithm

```
struct dct_t {




};
```

Another algorithm

```
struct dct_t {

    constexpr void operator()(
                                                    ) {


    }
};
```

Another algorithm

```
template <typename T, std::size_t S>
struct dct_t {


    constexpr void operator()(
                                                    ) {



    }
};
```

Another algorithm

```
template <typename T, std::size_t S>
struct dct_t {


    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {


    }
};
```

Another algorithm

```
template <typename T>
static constexpr void dct_transform(std::span<T> out,
                                    std::span<T> temp);

template <typename T, std::size_t S>
struct dct_t {


    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {



    }
};
```

Another algorithm

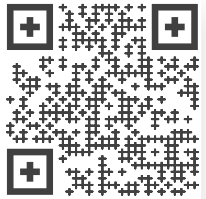
```
template <typename T>
static constexpr void dct_transform(std::span<T> out,
                                    std::span<T> temp);

template <typename T, std::size_t S>
struct dct_t {


    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {

        dct_transform<T>(         );
    }
};
```

Another algorithm

```
template <typename T>
static constexpr void dct_transform(std::span<T> out,
                                    std::span<T> temp);

template <typename T, std::size_t S>
struct dct_t {


    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {
        std::copy(in.begin(), in.end(), out.begin());
        dct_transform<T>(out,     );
    }
};
```

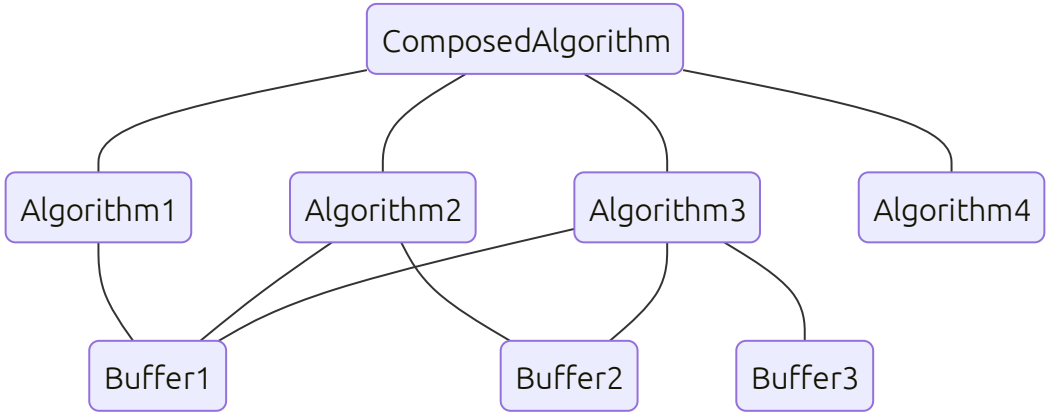
Another algorithm

```cpp
template <typename T>
static constexpr void dct_transform(std::span<T> out,
                                    std::span<T> temp);

template <typename T, std::size_t S>
struct dct_t {
    T temp[S]; // <-- another buffer!

    constexpr void operator()(std::span<const T, S> in,
                              std::span<T, S> out) {
        std::copy(in.begin(), in.end(), out.begin());
        dct_transform<T>(out, temp);
    }
};
```

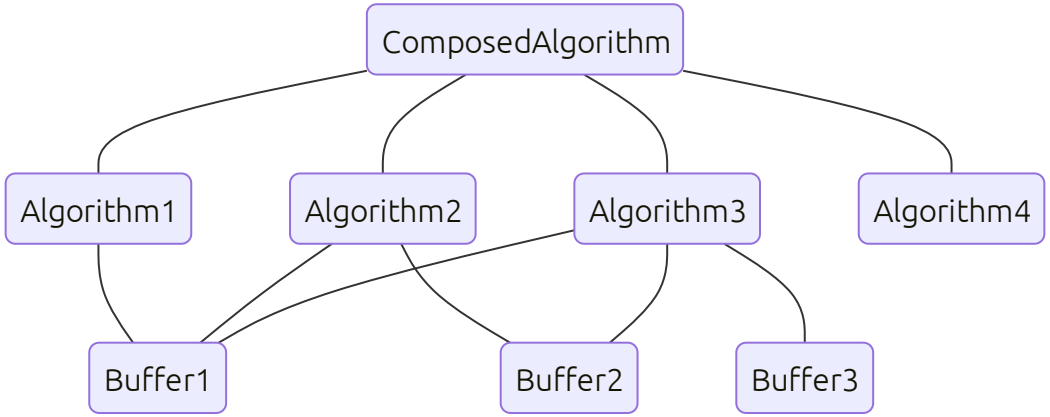


The idea

The idea


ComposedAlgorithm
Algorithm1
Algorithm2
Algorithm3
Algorithm4
Buffer1
Buffer2
Buffer3

The idea



Using *virtual inheritance* to share buffers

Modelling a buffer

```
struct buffer_t {



};
```

Modelling a buffer

```
template <          typename T, std::size_t S                    >
struct buffer_t {



};
```

Modelling a buffer

```
template <          typename T, std::size_t S                    >
struct buffer_t {

private:
    T storage[S];
};
```

Modelling a buffer

```
template <          typename T, std::size_t S                    >
struct buffer_t {
    inline operator std::span<T, S>() { return storage; }
private:
    T storage[S];
};
```

Modelling a buffer

```
template <auto tag, typename T, std::size_t S                    >
struct buffer_t {
    inline operator std::span<T, S>() { return storage; }
private:
    T storage[S];
};
```

Modelling a buffer

```
template <auto tag, typename T, std::size_t S = std::dynamic_extent>
struct buffer_t {
    inline operator std::span<T, S>() { return storage; }
private:
    T storage[S];
};
```

Modelling a buffer

```
template <auto tag, typename T>
struct buffer_t<tag, T, std::dynamic_extent> {




};
```

Modelling a buffer

```
template <auto tag, typename T>
struct buffer_t<tag, T, std::dynamic_extent> {









private:
    std::unique_ptr<T[]> storage;
    std::size_t size;
};
```

Modelling a buffer

```
template <auto tag, typename T>
struct buffer_t<tag, T, std::dynamic_extent> {









    inline operator std::span<T>() { return { storage.get(), size }; }
private:
    std::unique_ptr<T[]> storage;
    std::size_t size;
};
```

Modelling a buffer

```
template <auto tag, typename T>
struct buffer_t<tag, T, std::dynamic_extent> {
    inline void reset(std::size_t new_size) {




    }

    inline operator std::span<T>() { return { storage.get(), size }; }
private:
    std::unique_ptr<T[]> storage;
    std::size_t size;
};
```

Modelling a buffer

```
template <auto tag, typename T>
struct buffer_t<tag, T, std::dynamic_extent> {
    inline void reset(std::size_t new_size) {
        if (new_size != size) {


        }
    }

    inline operator std::span<T>() { return { storage.get(), size }; }
private:
    std::unique_ptr<T[]> storage;
    std::size_t size;
};
```

Modelling a buffer

```
template <auto tag, typename T>
struct buffer_t<tag, T, std::dynamic_extent> {
    inline void reset(std::size_t new_size) {
        if (new_size != size) {
            storage.reset(new T[new_size]);
            size = new_size;
        }
    }

    inline operator std::span<T>() { return { storage.get(), size }; }
private:
    std::unique_ptr<T[]> storage;
    std::size_t size;
};
```

Modelling a buffer

A little helper

```
auto whatever_buffer = get_buffer<whatever_tag>(this);
```

Modelling a buffer

A little helper

```
auto whatever_buffer = get_buffer<whatever_tag>(this);


std::span<    > get_buffer(buffer_t<tag,     >* buffer) {

}
```

Modelling a buffer

A little helper

```
auto whatever_buffer = get_buffer<whatever_tag>(this);

template <auto tag                              >
std::span<    > get_buffer(buffer_t<tag,     >* buffer) {

}
```

Modelling a buffer

A little helper

```
auto whatever_buffer = get_buffer<whatever_tag>(this);

template <auto tag, typename T, std::size_t S>
std::span<T, S> get_buffer(buffer_t<tag, T, S>* buffer) {

}
```

Modelling a buffer

A little helper

```
auto whatever_buffer = get_buffer<whatever_tag>(this);

template <auto tag, typename T, std::size_t S>
std::span<T, S> get_buffer(buffer_t<tag, T, S>* buffer) {
    return *buffer;
}
```

non-virtual virtual inheritance

non-virtual virtual inheritance

```
template <typename T, std::size_t S = std::dynamic_extent>
struct dct_t                                          {
    constexpr void dct(std::span<const T, S> in,
                       std::span<T, S> out) {
        std::copy(in.begin(), in.end(), out.begin());
        dct_transform<T>(out                                  );
    }
};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S = std::dynamic_extent>
struct dct_t :         buffer_t<                T, S> {
    constexpr void dct(std::span<const T, S> in,
                       std::span<T, S> out) {
        std::copy(in.begin(), in.end(), out.begin());
        dct_transform<T>(out                                  );
    }
};
```

non-virtual virtual inheritance

```
constexpr struct {} transform_tag;

template <typename T, std::size_t S = std::dynamic_extent>
struct dct_t :         buffer_t<transform_tag, T, S> {
    constexpr void dct(std::span<const T, S> in,
                       std::span<T, S> out) {
        std::copy(in.begin(), in.end(), out.begin());
        dct_transform<T>(out                                  );
    }
};
```

non-virtual virtual inheritance

```
constexpr struct {} transform_tag;

template <typename T, std::size_t S = std::dynamic_extent>
struct dct_t :         buffer_t<transform_tag, T, S> {
    constexpr void dct(std::span<const T, S> in,
                       std::span<T, S> out) {
        std::copy(in.begin(), in.end(), out.begin());
        dct_transform<T>(out, get_buffer<transform_tag>(this));
    }
};
```

non-virtual virtual inheritance

```
constexpr struct {} transform_tag;

template <typename T, std::size_t S = std::dynamic_extent>
struct dct_t : virtual buffer_t<transform_tag, T, S> {
    constexpr void dct(std::span<const T, S> in,
                       std::span<T, S> out) {
        std::copy(in.begin(), in.end(), out.begin());
        dct_transform<T>(out, get_buffer<transform_tag>(this));
    }
};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct windowed_spectrum_t                                          {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;









    constexpr void windowed_spectrum(std::span<const T, S> in,
                                     std::span<T, S> out) {



    }
};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct windowed_spectrum_t                                          {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;









    constexpr void windowed_spectrum(std::span<const T, S> in,
                                     std::span<T, S> out) {
        window(in,                                  );
        spectrum(                                  , out);
    }
};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct windowed_spectrum_t :         buffer_t<transform_tag, T, S>  {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;




    constexpr void windowed_spectrum(std::span<const T, S> in,
                                     std::span<T, S> out) {
        window(in,                                  );
        spectrum(                                  , out);
    }
};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct windowed_spectrum_t :          buffer_t<transform_tag, T, S>  {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;




    constexpr void windowed_spectrum(std::span<const T, S> in,
                                     std::span<T, S> out) {
        window(in, get_buffer<transform_tag>(this));
        spectrum(get_buffer<transform_tag>(this), out);
    }
};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct windowed_spectrum_t :          buffer_t<transform_tag, T, S>  {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;


    constexpr windowed_spectrum_t(auto fn) requires(S != std::dynamic_extent):
        window(fn) { }

    constexpr void windowed_spectrum(std::span<const T, S> in,
                                     std::span<T, S> out) {
        window(in, get_buffer<transform_tag>(this));
        spectrum(get_buffer<transform_tag>(this), out);
    }
};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct windowed_spectrum_t :         buffer_t<transform_tag, T, S>  {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;

    constexpr windowed_spectrum_t() = default;
    constexpr windowed_spectrum_t(auto fn) requires(S != std::dynamic_extent):
        window(fn) { }

    constexpr void windowed_spectrum(std::span<const T, S> in,
                                     std::span<T, S> out) {
        window(in, get_buffer<transform_tag>(this));
        spectrum(get_buffer<transform_tag>(this), out);
    }
};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct windowed_spectrum_t : virtual buffer_t<transform_tag, T, S>  {
    window_t<T, S> window;
    spectrum_t<T, S> spectrum;

    constexpr windowed_spectrum_t() = default;
    constexpr windowed_spectrum_t(auto fn) requires(S != std::dynamic_extent):
        window(fn) { }

    constexpr void windowed_spectrum(std::span<const T, S> in,
                                     std::span<T, S> out) {
        window(in, get_buffer<transform_tag>(this));
        spectrum(get_buffer<transform_tag>(this), out);
    }
};
```

non-virtual virtual inheritance

```
struct algorithms_t

{









};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct algorithms_t


{




};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct algorithms_t:
    windowed_spectrum_t<T, S>

{









};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct algorithms_t:
    windowed_spectrum_t<T, S>,
    dct_t<T, S>
{









};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct algorithms_t:
    windowed_spectrum_t<T, S>,
    dct_t<T, S>
{

    constexpr algorithms_t(auto fn) requires (S != std::dynamic_extent):
        windowed_spectrum_t<T, S>(fn) { }




};
```

non-virtual virtual inheritance

```
template <typename T, std::size_t S>
struct algorithms_t:
    windowed_spectrum_t<T, S>,
    dct_t<T, S>
{
    constexpr algorithms_t() = default;
    constexpr algorithms_t(auto fn) requires (S != std::dynamic_extent):
        windowed_spectrum_t<T, S>(fn) { }

    inline void reset(auto fn, std::size_t size) requires (S == std::dynamic_extent) {
        windowed_spectrum_t<T, S>::reset(fn, size);
        dct_t<T, S>::reset(size);
    }
};
```

non-virtual virtual inheritance

```
void run_static() {

    for (;;) {









    }
}
```

non-virtual virtual inheritance

```
void run_static() {
    float input[1024], output[1024];

    for (;;) {









    }
}
```

non-virtual virtual inheritance

```
void run_static() {
    float input[1024], output[1024];

    for (;;) {
        read_samples(input);


        write_usart(output);


        write_leds(output);
    }
}
```

non-virtual virtual inheritance

```
algorithms_t<float, 1024> algorithms(bartlett);

void run_static() {
    float input[1024], output[1024];

    for (;;) {
        read_samples(input);


        write_usart(output);


        write_leds(output);
    }
}
```

non-virtual virtual inheritance

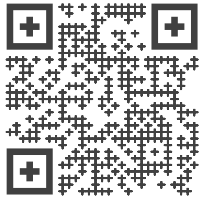
```
algorithms_t<float, 1024> algorithms(bartlett);

void run_static() {
    float input[1024], output[1024];

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);


        write_leds(output);
    }
}
```

non-virtual virtual inheritance

```
algorithms_t<float, 1024> algorithms(bartlett);

void run_static() {
    float input[1024], output[1024];

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```

non-virtual virtual inheritance

```
algorithms_t<float, 1024> algorithms(bartlett);

void run_static() {
    float input[1024], output[1024];

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```



```
_GLOBAL__sub_I_run_static():
    mov     edi, OFFSET FLAT:algorithms+8
    call    void bartlett_t::operator()<float, 1024ul>
    ...
vtable for spectrum_t<float, 1024ul>:
    ...
```

non-virtual virtual inheritance

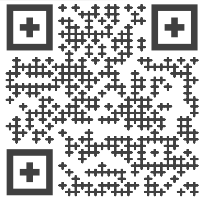
```
void run_stack() {

    float input[1024], output[1024];

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```

non-virtual virtual inheritance

```
void run_stack() {
    algorithms_t<float, 1024> algorithms(bartlett);
    float input[1024], output[1024];

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```

non-virtual virtual inheritance

```cpp
void run_stack() {
    algorithms_t<float, 1024> algorithms(bartlett);
    float input[1024], output[1024];

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```



```
run_stack():
    ...
    sub     rsp, 32856
    lea     rdi, [rsp+8248]
    lea     r15, [rsp+48]
    call    void bartlett_t::operator()<float, 1024ul>
    ...
vtable for spectrum_t<float, 1024ul>:
    ...
```

non-virtual virtual inheritance

```
void run_dynamic(std::size_t size) {




    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```

non-virtual virtual inheritance

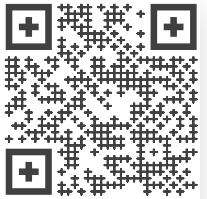
```
void run_dynamic(std::size_t size) {


    std::vector<float>
        input(1024), output(1024);

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```

non-virtual virtual inheritance

```
void run_dynamic(std::size_t size) {
    algorithms_t<float> algorithms;
    algorithms.reset(bartlett, size);
    std::vector<float>
        input(1024), output(1024);

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```

non-virtual virtual inheritance

```cpp
void run_dynamic(std::size_t size) {
    algorithms_t<float> algorithms;
    algorithms.reset(bartlett, size);
    std::vector<float>
        input(1024), output(1024);

    for (;;) {
        read_samples(input);

        algorithms.dct(input, output);
        write_usart(output);

        algorithms.windowed_spectrum(input, output);
        write_leds(output);
    }
}
```



```
run_dynamic(unsigned long):
    ...
    lea     rdi, [0+rbp*4]
    call    operator new[](unsigned long)
    mov     QWORD PTR [rsp+168], rax
    mov     rdi, rax
.L129:
    mov     rsi, rbp
    call    void bartlett_t::operator()
    ...
vtable for spectrum_t<float, 1024ul>:
    ...
```

non-virtual virtual inheritance

non-virtual virtual inheritance

- easy to implement

non-virtual virtual inheritance

- easy to implement
- no templates

non-virtual virtual inheritance

- easy to implement
- no templates
- some drawbacks

Thank you!