Core C++ 2025

19 Oct. 2025 :: Tel-Aviv

# Less Boilerplate, More Business Logic

# Features In C++ 20/23 That Makes Our Code Better

Dr. Miri (Kopel) Ben-Nissan

# About Me

- Dr. Miri (Kopel) Ben-Nissan.

- Ph.D. in Computer Science from Bar-Ilan University.

- 20+ years of experience in C++ programming in Algorithms, Real Time and Backend environments. Team Leader and System Architect.

- Lecturer at Bar-Ilan University, owner of MKBN Consultancy. More than 20 years experience in teaching C++, advanced programming and design courses.

- [miri@mkbn.co.il](mailto:miri@mkbn.co.il)


I ♥ OOP


challenge

massive growth of the code bases

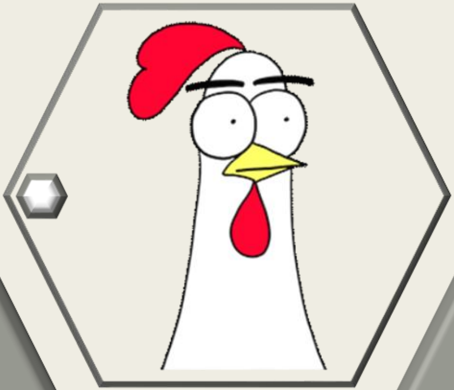
minimum amount of noise


correct

concise

clean

focused



templates

technical syntax

definitions

code reuse

auto generated code

frameworks


# Boilerplate Code

- repetitive
- must be included in many different places in a program

# Example

Can refer to necessary code for running tasks, although not doing much logic.

```cpp
#include <iostream>

int main(){
    std::cout<<"Hello World"<<std::endl;
    return 0;
}
```

# Example

```
for(auto it=vec.begin(); itr!=vec.end(); ++itr){
    //do something with *itr
}
```

```
for(auto x : vec){
    //do something with x
}
```

```
std::for_each(v.begin(), v.end(), [](int &n) { n++; });
```


# IS IT BAD?!

Makes software development more efficient

Saves time and accelerate the coding process

## Example

A “string utils” library which implements common string actions like concatenation, lowercase, substring etc’.


## Example

Code structure like **code generation tool**, that generates code for a class definition with ctor, dtor, public, private etc'.

The programmer needs just to fill in the rest…

# what can go wrong?

# When two boilerplate codes depends on each other…

## Examples

- Operators `new`, `new[]` and `delete`, `delete[]`
- STL containers with the need to add comparators.

**The code is becoming contextually redundant.**

In general, programmers spend more time in reading code than in writing it.

So, make sure code is focusing as possible on the business logic

# Features that allows focusing in logic

- Data List Processing – `std::ranges` and `std::views`
- Data Structure Initialization – designated initializers
- Error Handling – `std::expected`

# Data Sequence Processing

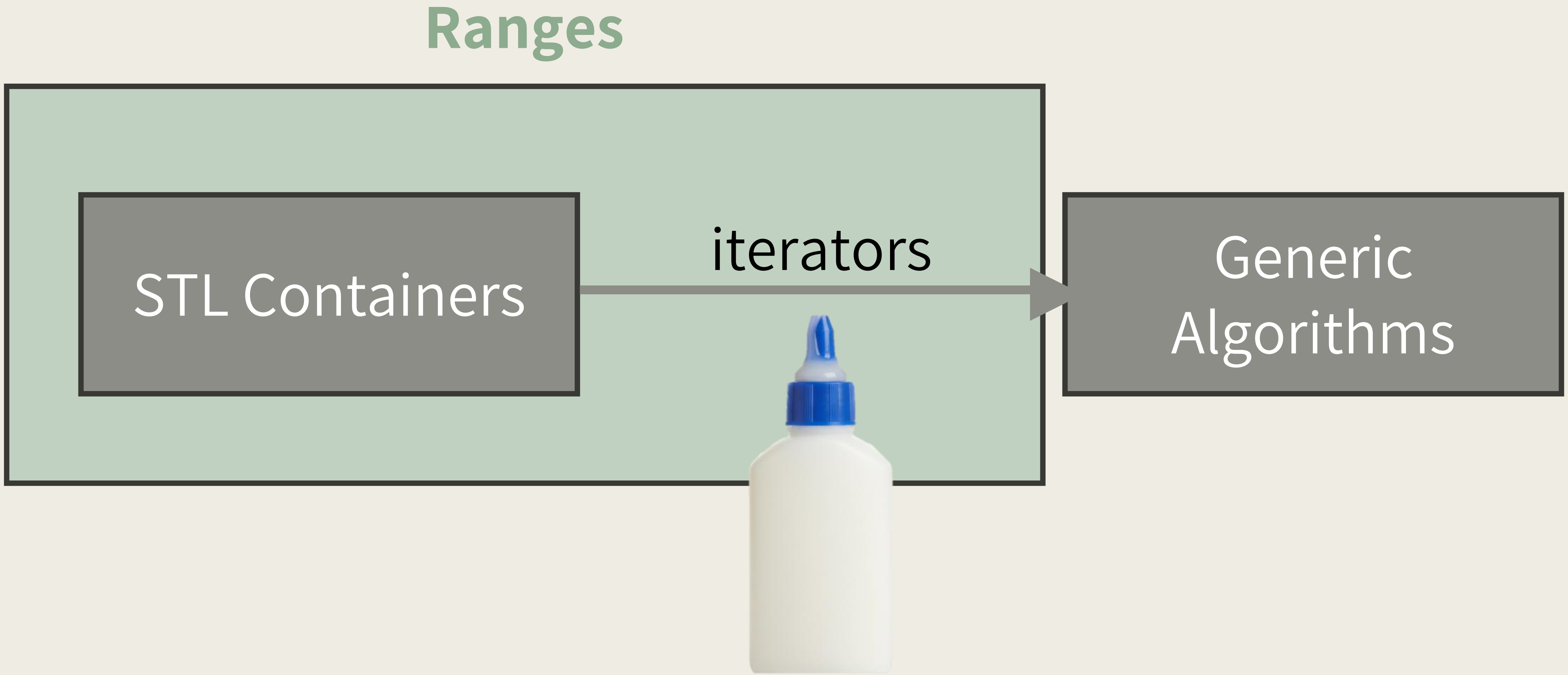
# Example

```
int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5, 6};
    std::vector<int> evensSquared;

    for (const auto& x : numbers) {
        if (x % 2 == 0) {
            evensSquared.push_back(x * x);
        }
    }

    for (const auto& n : evensSquared) {
        std::cout << n << " ";
    }
}
```

# C++20: Ranges and Views



*std::ranges and vies(Cont…)*

- **A range** is any type that can be iterated over, typically by providing a pair of begin() and end() functions (or an iterator and a sentinel).

- **Until C++20:** a range is a pair of iterators.

```
std::sort(vec.begin(), vec.end());
```

**usually extra work**

# *std::ranges and vies(Cont…)*

- **Ranges Algorithms**: operate directly on range objects instead of iterator pairs, simplifying code.
  - for example: `std::ranges::sort`, `std::ranges::find`

*std::ranges and vies(Cont…)*

- **Since C++20:** a sequence is treated as a unified object.

```
std::ranges::sort(vec);
```


- *Note: ranges are not about skipping iterators!*

- A **view** is a special kind of range that is **non-owning, cheap to copy, and lazy**.
  - *except ownin_view*

- The `std::views` namespace (an alias for `std::ranges::views`) contains the **range adaptors** used to create views.

- **Examples of View Adaptors:**
  - **`std::views::filter`** (filters elements based on a predicate)
  - **`std::views::transform`** (applies a function to each element)
  - **`std::views::reverse`** (iterates over a range backward)

| Feature | std::ranges | std::views |
|---|---|---|
| Category | General concept and library framework | Specific type of range (conceptually) |
| Scope | Any iterable sequence (containers, views, C-arrays, etc.) | Non-owning, lightweight ranges |
| Ownership | Can be owning (std::vector) or non-owning (std::string_view) | Primarily non-owning |
| Copy/Move Cost | Can be proportional to size (O(N)) for containers | Constant Time (O(1)) |
| Evaluation | Eager (containers), or determined by the type of range/algorithm | Lazy (computes elements only as requested) |
| Usage | Used as input for range algorithms (std::ranges::sort(vec)) | Used to create transformation and filtering pipelines |

*std::ranges and vies(Cont…)*

- **Ranges library** supports a set of standard algorithms for data filtering, transformation, and sorting, in a condensed and clear manner.

- **Range algorithms** = applied to ranges **eagerly**.

- **Range adaptors** = applied to views **lazily**.
  Adaptors can be composed into pipelines, so that their actions take place as the view is iterated.

```cpp
//a functor
struct SquareAndLog {
    SquareAndLog() {
        std::cout<<"[CTOR] SquareAndLog functor created"<<std::endl;
    }

    ~SquareAndLog() {
        std::cout<<"[DTOR] SquareAndLog functor destroyed"<<std::endl;
    }

    int operator()(int n) const {
        std::cout<<"[LOG] Processing element: " << n << std::endl;
        return n * n;
    }
};
```

```cpp
void run_demonstration() {
    std::vector<int> numbers{10, 20, 30, 40, 50};
    std::vector<int> eager_result(numbers.size());

    std::ranges::transform(numbers,
                           eager_result.begin(),
                           SquareAndLog{});

    for (int val : eager_result) {
        std::cout << "[OUTPUT] Stored value received: "
                  << val
                  << std::endl;
    }
}
```

## *std::ranges and vies(Cont…)*

- Accessing elements in the view is done “lazily”. This makes it very cheap to combine or compose views.

```cpp
void run_demonstration() {
    std::vector<int> numbers{10, 20, 30, 40, 50};

    auto squared_view = numbers
                      | std::views::transform(SquareAndLog{});

    // ONLY the work for the first element runs, proving laziness.
    if (!squared_view.empty()) {
        std::cout << "Requesting the first element" << std::endl;

        int first_square = *squared_view.begin();

        std::cout <<"[OUTPUT] Result of first element: "
                << first_square << std::endl;
    }
}
```

## go back to our example...

```
#include <vector>
#include <ranges>
#include <iostream>
#include <print>

int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5, 6};

    auto evensSquared = numbers
        | std::views::filter([](int n){ return n % 2 == 0; })
        | std::views::transform([](int n){ return n * n; });

   std::print("{}",evensSquared);
}
```

# C++20: convert output to std::list

```
int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5, 6};

    auto evensSquared = numbers
        | std::views::filter([](int n){ return n % 2 == 0; })
        | std::views::transform([](int n){ return n * n; });

    // Collect the filtered and transformed range into a std::list
    std::list<int> evensSquaredList{
                    evensSquared.begin(), evensSquared.end()};


    std::print("{}",evensSquaredList);
}
```

# C++23: std::ranges::to

- `std::ranges::to` is a C++23 utility that takes a range (like a view or a container) and constructs a new non-view object (a container) from it, storing the elements of the input range into the new container.

# C++23: convert output to std::list

```
#include <vector>
#include <ranges>
#include <iostream>
#include <print>

int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5, 6};

    auto evensSquared = numbers
        | std::views::filter([](int n){ return n % 2 == 0; })
        | std::views::transform([](int n){ return n * n; })
        | std::ranges::to<std::list>();

    std::print("{}",evensSquared);
}
```

# Advantages

Clean and Continuous code

Less Error Prone

Support for Various Containers

Readability

# Data Structure Initialization

# Example

```cpp
// C++17: Boilerplate for handling optional parameters
struct ConnectionOptions {
    int timeout_ms{1000};
    int retries{3};
    std::string protocol{"TCP"};
    int buffer_size{4096};

    // BOILERPLATE 1: Full constructor
    ConnectionOptions(int t, int r, const std::string& p, int b)
        : timeout_ms{t}, retries{r}, protocol{p}, buffer_size{b} {}

    // BOILERPLATE 2: Constructor for timeout and retries only
    ConnectionOptions(int t, int r)
        : timeout_ms{t}, retries{r} {}

    // BOILERPLATE 3: Constructor for just protocol
    ConnectionOptions(const std::string& p)
        : protocol{p} {}

    // ... you would need many more constructors...
};
```

# Solution may be: Builder Pattern

```
struct ConnectionOptions {
public:
    void print() const {
        std::cout   <<"time out: "<<timeout_ms<<" | retries: "<<retries
                    <<" | protocol: "<<protocol<<" | buffer size: “
                    <<buffer_size<<std::endl;
    }

private:
    ConnectionOptions(size_t t, size_t r, const std::string& p, size_t b)
        : timeout_ms(t), retries(r), protocol(p), buffer_size(b) {}

    friend struct ConnectionOptBuilder;

    size_t timeout_ms{1000};
    size_t retries{3};
    std::string protocol{"TCP"};
    size_t buffer_size{4096};
};
```

```cpp
struct ConnectionOptBuilder{
    ConnectionOptBuilder() = default;

    ConnectionOptBuilder & timeout(size_t t) {_timeout_ms = t; return *this;}
    ConnectionOptBuilder & retries(size_t r) {_retries = r; return *this;}
    ConnectionOptBuilder & protocol(std::string p) {_protocol=p; return *this;}
    ConnectionOptBuilder & buffSize(size_t bs) {_buffer_size = bs; return *this;}

    ConnectionOptions build() {
      return ConnectionOptions{_timeout_ms, _retries, _protocol, _buffer_size};
    }

private:
    size_t _timeout_ms{1000};
    size_t _retries{3};
    std::string _protocol{"TCP"};
    size_t _buffer_size{4096};
};
```

```cpp
int main(){
    auto co{ConnectionOptBuilder().timeout(5).buffSize(10).build()};
    co.print();
}
```

# C++20 Designated Initializers

- A form of Aggregate Initialization.
- Increases readability.

```
struct Date {
    int year;
    int month;
    int day;
};

Date(int,int,int);
```

```
struct Date {
    int year;
    int month;
    int day;
};


Date(int,int,int);


Date d{ .year = 2055, .month = 3, .day = 10 };
```

# *C++ Designated Initializers (Cont…)*

- Designated initializers are **only valid for aggregates**.
- **Designators must appear in the same order** as the corresponding data members are declared.
- You **cannot mix designated initializers with positional initializers** (the regular, ordered way).
- Designators can only be used for **non-static data members**.
- You **cannot use a designator more than once** for the same data member.
- **Nesting is disallowed:** Designators cannot be used to initialize members of nested anonymous structs or unions.
- It is **not mandatory** to specify a value for every member.

```cpp
// C++20: No constructors needed!
struct ConnectionOptions {
    int timeout_ms = 1000;
    int retries = 3;
    std::string protocol = "TCP";
    int buffer_size = 4096;
    // No explicit constructors are written.
};


ConnectionOptions opts1{ .retries = 10 };


ConnectionOptions opts2{ .timeout_ms = 500, .protocol = "UDP" };
```

# Advantages

- **Implicit Constructor Boilerplate** – in C++17 style forces you to memorize the exact declaration order of members. Hard to add/remove data members.

- **Safety Boilerplate** – designated initializers act as self-documenting code.

# But…

- Fields cannot be private.
- No logic can be performed on constructor (when you think it is a good idea…)

# C++26 reflection - Keyword Arguments

- C++26 proposal for reflection can help with enforce encapsulation of the fields, since designated initializers are implemented only for aggregate types.

- Instead of manually add all ctor permutations and get/set, the compiler will be able to do it if we will ask for it, and will handle adding/removing/renaming fields in the class.

# Error Handling

# Example

```cpp
#include <optional>
#include <string>
#include <iostream>

std::optional<int> parseInt(const std::string& s) {
    try {
        return std::stoi(s);
    } catch (...) {
        return std::nullopt;
    }
}

int main() {
    auto val = parseInt("42");
    if (val) {
        std::cout << "Value: " << *val;
    } else {
        std::cout << "Error parsing integer";
    }
}
```

# Suggestion 1: extra parameter for error msg

```
std::optional<int> parseInt2( const std::string& s,
                              std::string& errormsg)
{
    try {
        return std::stoi(s);
    } catch (...) {
        errormsg = "error parsing string to integer";
        return std::nullopt;
    }
}
```

```cpp
void test2(){
    std::string errMsg;

    auto val = parseInt2("42",errMsg);

    if (val) {
        std::cout << "Value: " << *val;
    } else {
        std::cout << "Error parsing integer: "<<errMsg;
    }
}
```

# Suggestion 2: return pair/variant/struct

```
std::pair<int,std::string> parseInt3(const std::string& s)
{
    try {
        return std::make_pair(std::stoi(s),"");
    } catch (...) {
        return std::make_pair(-1,
              "error parsing string to integer");
    }
}
```

```cpp
void test3(){
    std::string errMsg;

    auto [val,errMsg] = parseInt3("42");

    if (not errMsg.empty()) {
        std::cout << "Value: " << val;
    } else {
        std::cout << "Error parsing integer: "<<errMsg;
    }
}
```

# c++23: expected

- The <expected> header in C++23 introduces a new way to handle errors and expected values.

- The class template `std::expected` provides a way to represent either of two values: an **expected** value of type **T**, or an **unexpected** value of type **E**.

- The <expected> header introduces two main class templates:
  - *std::expected*
  - *std::unexpected*

# Member functions of the std::expected class

| | |
|---|---|
| **value()** | retrieve the stored value of type T. |
| **error()** | gives access to the stored error of type E. |
| **has_value()** | returns true if the `std::expected` holds a value and false if it holds an error. |
| **error_code()** | When applicable, this member function is employed to convert the stored error into an error code. |

# std::unexpected template class

```
std::expected<T, E> function_name {
        // statements
        return std::unexpected< E >(some_value);
}
```

- The C++ type system ensures that you cannot inadvertently mix up expected values and errors.

- More efficient than using exceptions.

```cpp
// Function now returns an int on success or a specific error
(std::string) on failure
std::expected<int, std::string> parseInt(const std::string& s) {
    try {
        return std::stoi(s);
    } catch (...) {
        return std::unexpected("Invalid integer");
    }
}
int main() {
    auto val = parseInt("42");
    if (val) {
        std::cout << "Value: " << *val;
    } else {
        // Accesses the specific error value, eliminating ambiguity
        std::cout << "Error: " << val.error();
    }
}
```

# So, did we REALLY get rid from the boilerplate code here?

```cpp
#include <iostream>
#include <expected>

class A{
public:
    A() {std::cout<<"A()\n";}
    explicit A(int x) : _x{x} {std::cout<<"A("<<_x<<")\n";}
    A(const A&) {std::cout<<"A(const A&)\n";}
    A(A&&) {std::cout<<"A(A&&)\n";}
    ~A(){std::cout<<"~A()\n";}
private:
    int _x;
};

std::expected<A,int> func(){
    return A{5};
}
```

```cpp
#include <iostream>
#include <expected>

class A{
public:
    A() {std::cout<<"A()\n";}
    explicit A(int x) : _x{x} {std::cout<<"A("<<_x<<")\n";}
    A(const A&) {std::cout<<"A(const A&)\n";}
    A(A&&) {std::cout<<"A(A&&)\n";}
    ~A(){std::cout<<"~A()\n";}
private:
    int _x;
};

std::expected<A,int> func(){
    return std::expected<A,int>{std::in_place,5};
    // return A{5};
}
```

# CONCLUSION

# Conclusion

- We set out to explore C++20 and C++23 features that directly enable us to write less infrastructural code and focus squarely on the problem domain.

- **The compiler is now doing more of the low-level work, freeing us to write what truly matters: our business logic.**

- To master modern C++, we must embrace this new paradigm of expressiveness, safety, and efficiency.

# THANK YOU!