

Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Part 4 – Stream Iterators and Exceptions



IFS

INSTITUTE FOR
SOFTWARE

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 11.10.2017
HS2017



HSR

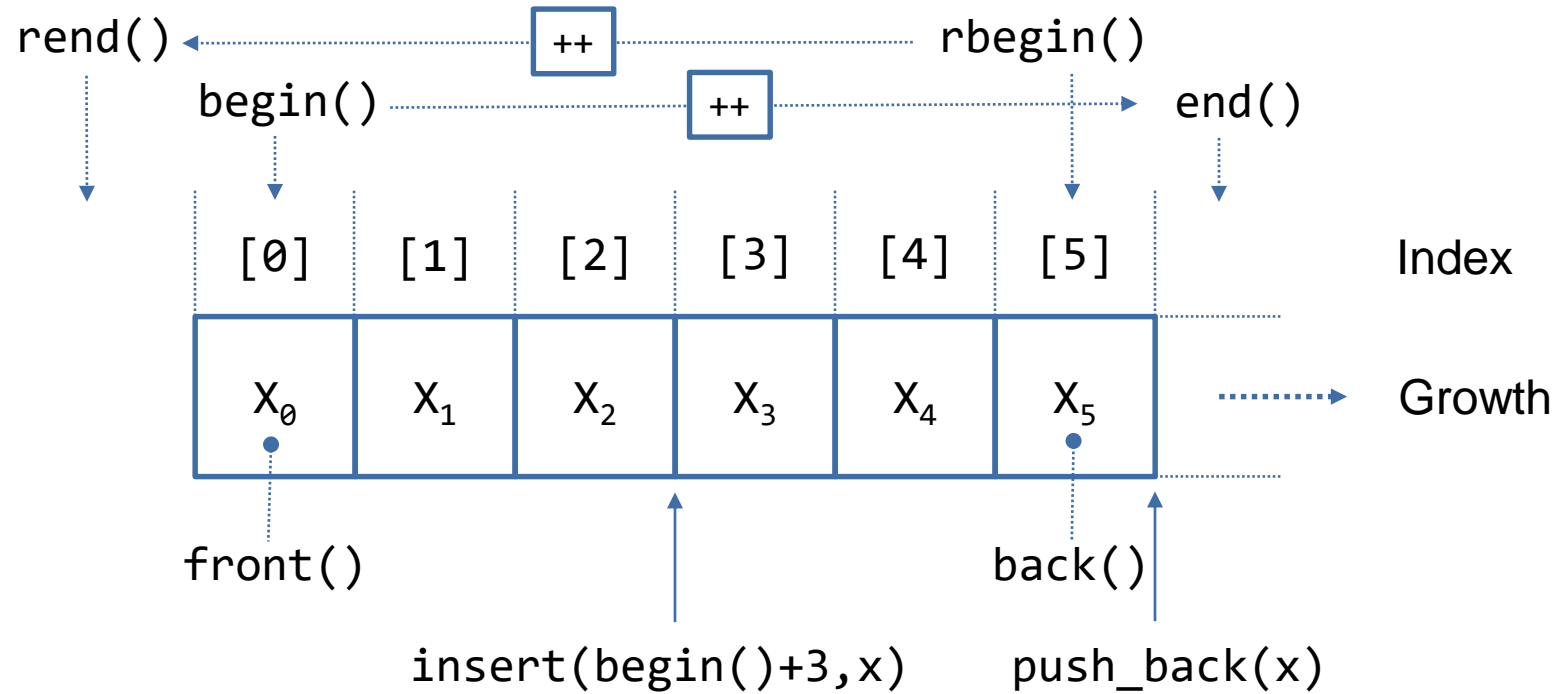
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Recap Week 3

```
std::vector<int>(6);
```

Iterators:



```
void print(int x) {
    std::cout << "print: " << x << ' ';
}

void printAll(std::vector<int> v) {
    std::for_each(std::cbegin(v), std::cend(v), print);
}

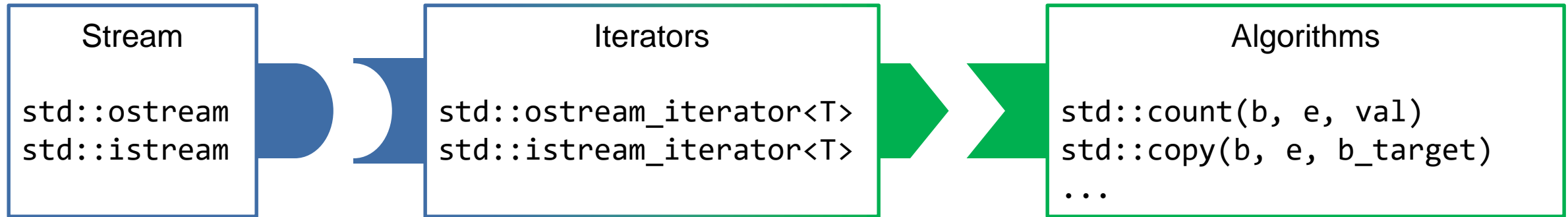
void printAllReverse(std::vector<int> v) {
    std::for_each(std::crbegin(v), std::crend(v), print);
}

int main() {
    std::vector<int> values{1, 2, 3, 4, 5};
    printAll(values);           //prints 1 2 3 4 5
    printAllReverse(values);    //prints 5 4 3 2 1
}
```

Iterators for I/O

Goals:

- You know how to create iterators for `std::istream` and `std::ostream`
- You can specify ranges on streams with stream iterators



- Streams (`std::istream` and `std::ostream`) cannot be used with algorithms directly

```
std::copy(std::begin(v), std::end(v), std::ostream_iterator<int>{std::cout, ", "});
```

- `std::ostream_iterator<T>` outputs values of type `T` to the given `std::ostream`
 - No `end()` marker needed for output, it ends when the input range ends
- `std::istream_iterator<T>` reads values of type `T` from the given `std::istream`
 - End iterator is the default constructed `std::istream_iterator<T>{}`
 - It ends when the stream is no longer good()

- The (stream) iterators have a very unpleasant name length, even with auto-completion
- A type alias can help to abbreviate that

```
using <alias-name> = <type>;
```

- Useful if long type names occur more than once
- Example
 - Copy strings from standard input to standard output

```
using input = std::istream_iterator<std::string>;  
input eof{};  
input in{std::cin};  
std::ostream_iterator<std::string> out{std::cout, " "};  
std::copy(in, eof, out);
```

- **std::istream_iterator uses operator >> for input**
 - Disadvantage: It skips white space
- **For an exact copy, we also need the rest**
- **std::istreambuf_iterator<char> uses std::istream::get() to get every character**
 - This only works with char-like types

```
using input = std::istreambuf_iterator<char>;  
input eof{};  
input in{std::cin};  
std::ostream_iterator<char> out{std::cout, " "};  
std::copy(in, eof, out);
```


- To fill a vector from a stream you can either use `copy` with `std::back_inserter(v)`
 - It uses `v.push_back()` internally

```
using input = std::istream_iterator<int>;
input eof{};
std::vector<int> v{};
std::copy(input{std::cin}, eof, std::back_inserter(v));
```

- Or, construct the `std::vector<T>` directly from two iterators

```
using input = std::istream_iterator<int>;
input eof{};
std::vector<int> const v{input{std::cin}, eof};
```

References

Goals:

- You know the difference between a value and a reference

- **A reference is an alias for a variable or value**
 - The original must exist as long as it is referred to!
- **Mostly useful for function parameters**
 - Avoids copying large objects (const reference)
 - Provides side-effects on call-side variable (non-const reference)
- **Sometimes useful for member/local variables**
 - Caution: Dangling References
- **Rarely useful as return type**



- For demonstration purposes only!

```
int i{42};  
int & ri{i};           // must initialize ref  
int const & cri{ i }; // const alias  
int const & cr{6*7};  // extend lifetime of 6*7  
ri = 43;              // changes i, ri only an alias  
//--cri;             // doesn't work -> const
```

- Ampersand (&) prefix for lvalue references

- Must always be initialized!
- No null-references

- const & prefix for const references

- const reference to non-const value is possible

- Rvalue references (&&) are a topic in C++ Advanced

- **Scoping rules are similar to other languages**
- **Parameters are visible in function block**
- **In C++: Lifetime is equally important**
 - When block scope ends, lifetime is over (} "collects garbage")
 - Dependability on destruction is a key feature of C++
- **Beware of redefining the same name**
 - Shadowing happens and is not an error as in Java

```
void showScopingRules(int i, double d) {
    unsigned j{1};
    // cannot use name i instead of j
    std::cout << i << "\n";
    {
        char i{'d'}; // shadows parameter i
        // parameter i not accessible but d is
        std::cout << i << " " << d << "\n";
    }
    ++i; // that is the parameter i
        // no longer shadowed
    for (auto i = 0u; i < j; ++i) {
        // another i
        std::cout << i << "\n";
    }
    std::cout << i << "\n";
    // parameter i again
}
```

Functions (One More Time)

Goals:

- You know the different ways of passing parameters to functions
- You know how to pass default arguments to functions
- You know how to write a lambda

	const: <ul style="list-style-type: none">Parameter cannot be changed	non-const: <ul style="list-style-type: none">Parameter can be changed
reference: <ul style="list-style-type: none">Argument on call-site is accessed	<pre>void f(std::string const & s) { //no modification //efficient for large objects }</pre>	<pre>void f(std::string & s) { //modification possible //side-effect also at call-site }</pre>
copy: <ul style="list-style-type: none">Function has its own copy of the parameter	<pre>void f(std::string const s) { //no modification //rarely used }</pre>	<pre>void f(std::string s) { //modification possible //side-effect only locally }</pre>

■ Call-site always looks the same

- You cannot pass a const argument to a non-const reference

```
std::string name{"John"};  
f(name);
```

- **Value Parameter:** `void f(type par);`
 - Default for parameters
- **Reference Parameter:** `void f(type & par);`
 - When side-effect is required at call-site
- **Const-Reference Parameter:** `void f(type const & par);`
 - Possible optimization, when type is large (costly to copy) and no side-effects desired at call-site
 - For non-copyable objects
- **Const Value Parameter:** `void f(type const par);`
 - Hardly ever
 - Could prevent changing the parameter in the function inadvertently

- **By Value:** `type f();`

- Default for return types

- **By Reference:** `type & f();` or `type const & f();`

- Only return a reference parameter (or a class member variable from a member function)

```
std::ostream & sayHello(std::ostream & out) {  
    return out << "Hello";  
}
```

- Never return a reference to a local variable!

```
std::string & create() {  
    std::string name{"John"};  
    return name;  
}
```



- **By Const Value:** `type const f();`

- Don't do this, it just annoys the caller

```
void incr(int & var);  
void incr(int & var, unsigned delta);
```


- **The same function name can be used for different functions if parameter number or types differ**
 - Functions cannot be overloaded just by their return type
 - If only the parameter type is different there might be ambiguities
- **Resolution of overloads happens at compile-time = Ad hoc polymorphism**
- **Internal name of a function also contains its parameter types as significant information (not the parameter names)**
 - Information required by the linker

```
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    }
    return 1;
}

double factorial(double n) {
    double result = 1;
    if (n < 15) {
        return factorial(static_cast<int>(n));
    }
    while (n > 1) {
        result *= n;
        --n;
    }
    return result;
}
```

```
void demoAmbiguity() {
    std::cout << factorial(3) << '\n';
    std::cout << factorial(1e2) << '\n';

    std::cout << factorial(10u) << '\n';
    std::cout << factorial(1e1L) << '\n';
}
```



```
void incr(int & var, unsigned delta = 1);
```

- **A function declaration can provide default arguments for its parameters from the right**

- Definition doesn't need to/shouldn't repeat
- = is required as part of the parameter declaration

```
void incr(int & var, unsigned delta) {  
    var += delta;  
}
```

- **Implicit overload of the function with fewer parameters**

- If n default arguments are provided, n+1 versions of the function are declared

- **Default arguments can be omitted when calling the function**

```
int counter{0};  
incr(counter);    //uses default for delta  
incr(counter, 5);
```

```
void applyAndPrint(double x, double f(double)) {  
    std::cout << "f(" << x << ") = " << f(x) << '\n';  
}
```

■ Functions are "first class" objects in C++

- You can pass them as argument
- You can keep them in references variables
- In C you could do that with function pointers (only)

■ Syntax for function reference types

```
double (&h)(double)
```

- Later more with generic parameters in template functions, which also allow passing lambdas

■ Defining inline functions

```
auto const g = [](char c) -> char {  
    return std::toupper(c);  
};  
g('a');
```

■ auto const for function variable from Lambda

■ [] introduces a Lambda function

- Can contain captures: [=] or [&] to access variables from scope

■ (Parameters) as with other functions, but optional if empty

- Parameters can be `auto` (Generic Lambda)

■ Body block with statements

■ Capturing a local variable by value

- Local copy lives as long as the lambda lives
- Local copy is immutable, unless lambda is declared mutable

```
int x = 5;
auto l = [x]() mutable {
    std::cout << x;
};
```

■ Capturing a local variable by reference

- Allows modification of the captured variable
- Side-effect is visible in the surrounding scope, but referenced variable must live at least as long as the lambda lives

```
int x = 5;
auto const l = [&x]() {
    std::cout << ++x;
};
```

■ Capturing all local variables by value or reference

```
int x = 5;
auto l = [=]() mutable {
    std::cout << x;
};
```

```
int x = 5;
auto const l = [&]() {
    std::cout << ++x;
};
```

■ Capturing this pointer

- Accessing and modify members of the class is possible

```
struct S {  
    void foo() {  
        auto square = [this] {  
            member *= 2;  
        };  
    }  
private:  
    int member{};  
};
```

■ New local variable can be specified in capture

- New variable in capture has type `auto`
- Can be modified if lambda is `mutable`

```
auto squares = [x=1]() mutable {  
    std::cout << x *= 2;  
};
```

-
- In capture above examples can be combined and separated with commas (,)

Failing Functions

Goals:

- You know 5 different ways to react to errors in functions
- You know how to throw, catch and test exceptions

- **Precondition (Assumption) is violated**
 - Negative index
 - Divisor is zero
 - Usually caller provided wrong arguments
- **A function without preconditions has a so-called "wide contract" as opposed to "narrow contract"**
- **Postcondition could not be satisfied**
 - Resources for computation are not available
 - Can not open a file

Contract cannot be fulfilled

- **What should you do, if a function cannot fulfill its purpose?**
 1. Ignore the error and provide potentially **undefined behavior**
 2. Return a **standard result** to cover the error
 3. Return an **error code** or error value
 4. Provide an **error status** as a side-effect
 5. Throw an **exception**

- **But first you need to know, if it can fail at all!**



```
std::vector<int>{1, 2, 3, 4, 5};  
v[5] = 7;
```



- Relies on the caller to satisfy all preconditions
- Viable only if not dependent on other resources
- Most efficient implementation
 - No unnecessary checks
- Simple for the implementer but harder for the caller
- Should be done consciously and consistently!

```
std::string inputName(std::istream & in) {  
    std::string name{};  
    in >> name;  
    return name.size() ? name : "anonymous";  
}
```

- Reliefs the caller from the need to care if it can continue with the default value
- Can hide underlying problems
 - Debugging can give you nightmares
- Often better if caller can specify its own default value

```
std::string inputNameWithDefault(std::istream & in,  
                                std::string const & def = "anonymous") {  
    std::string name{};  
    in >> name;  
    return name.size() ? name : def;  
}
```

■ Only feasible if result domain is smaller than return type

- There exists a value that can be used
- Sometimes invented artificially: `std::string::npos`

```
bool contains(std::string const & s, int number) {  
    auto substring = std::to_string(number);  
    return s.find(substring) != std::string::npos;  
}
```

- POSIX defines -1 to mark failure of many system calls

■ Burden on the caller to check the result

- Danger of ignoring significant errors if result is otherwise insignificant
- In C++17 we get the `std::optional<T>` type

```
std::optional<std::string> inputName(std::istream & in) {  
    std::string name{};  
    if (in >> name) return name;  
    return {};  
}
```

- **Requires reference parameter**

- Can be this object in member functions
- Annoying when error variable must be provided

```
int connect(std::string url, bool & error) {  
    //set error when an error occurred  
}
```

- **(Bad!) Alternative: Global variable**

- POSIX' errno is the glorious example of that

- **Example: std::istream's state (good(), fail()) is changed as a side-effect of input**

```
std::string name{};  
in >> name;  
if (in.fail()) { //Member variable  
    //Handle error case  
}
```

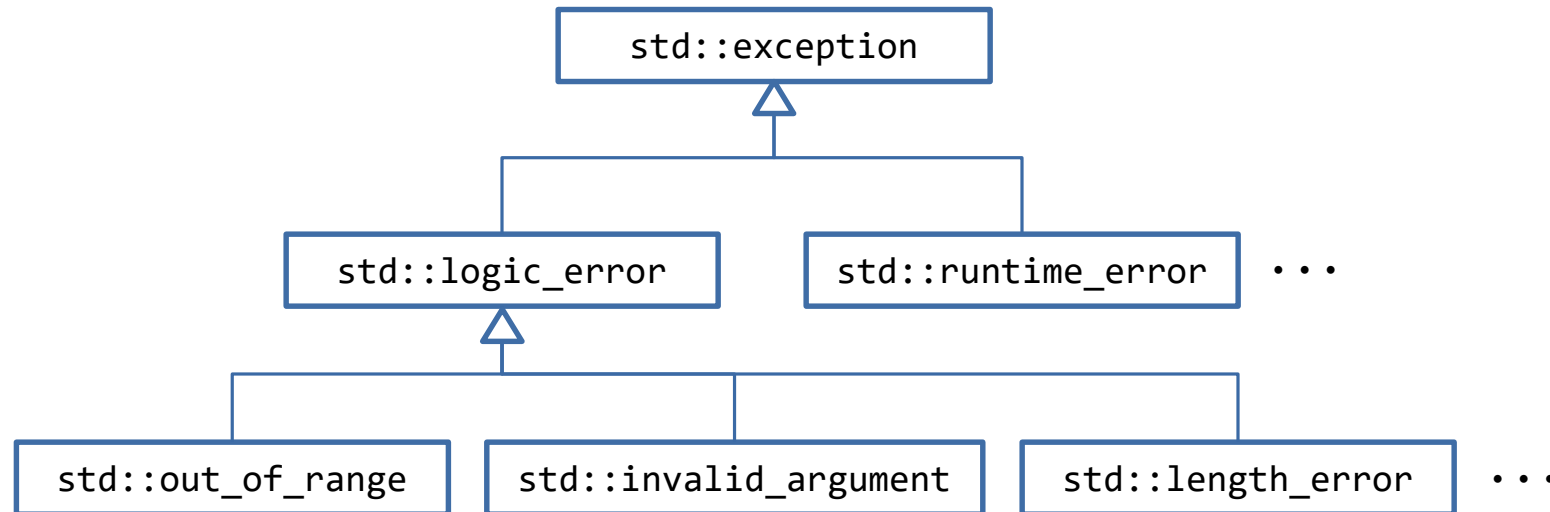
```
throw std::invalid_argument{"Description"};  
throw 15;
```

- **To throw an exception:** `throw value;`
 - Any (copyable) type can be thrown
- **No means to specify what could be thrown**
 - No checks if you catch an exception that might be thrown at call-site
- **No meta-information is available as part of the exception**
 - No stack trace, no source position of throw
- **Exception thrown while exception is propagated results in program abort (not while caught)**

- **Try-catch block as in Java**
 - Can be the whole function body
- **Principle: Throw by value, catch by const reference**
 - Avoids unnecessary copying
 - Allows dynamic polymorphism for class types
- **Sequence of catches is significant**
 - First match wins
- **Catch all with ellipsis (...):**
 - Must be last catch
- **Caught exceptions can be rethrown with throw;**

```
try {  
    throwingCall();  
} catch (type const & e) {  
    //Handle type exception  
} catch (type2 const & e) {  
    //Handle type2 exception  
} catch (...) {  
    //Handle other exception types  
}
```

- The Standard Library has some pre-defined exception types that you can also use in <stdexcept>



- All have a constructor parameter for the "reason" of type `std::string`
- `std::exception` is the base class
 - It provides the `what()` member function to obtain the "reason"

- **Functions that have a precondition on their caller**
 - When not all possible argument values are useful for the function

```
double square_root(double x) {  
    if (x < 0) {  
        throw std::invalid_argument{"square_root imaginary"};  
    }  
    return std::sqrt(x);  
}
```

- **Do NOT use exceptions as a second means to return values**
 - Catch then becomes a "come from" and throw a "go to"

- CUTE provides **ASSERT_THROWS(code, exception)**

```
void testSquareRootNegativeThrows() {  
    ASSERT_THROWS(square_root(-1.0), std::invalid_argument);  
}
```

```
void testEmptyVectorAtThrows() {  
    std::vector<int> empty_vector{};  
    ASSERT_THROWS(empty_vector.at(0), std::out_of_range);  
}
```

- You can also use **try-FAILM()-catch**

```
void testForExceptionTryCatch() {  
    std::vector<int> empty_vector{};  
    try {  
        empty_vector.at(1);  
        FAILM("expected Exception");  
    } catch (std::out_of_range const &) {  
        // expected  
    }  
}
```

■ A good function

- Does one thing well and is named after that ("High Cohesion")
- Has only few parameters (≤ 3 , max 5)
- Consists of only a few lines without deeply nested control structure
- Provides guarantees about its result (aka its contract)
- Is easy to use with all possible argument values its parameter types allow or provides consistent error reporting if argument values prohibit delivering its result (exception)

■ Pass parameters and return results by value (unless there is a good reason not to)