Programovací jazyky a výkonnost

Programovací jazyky pro numerické aplikace

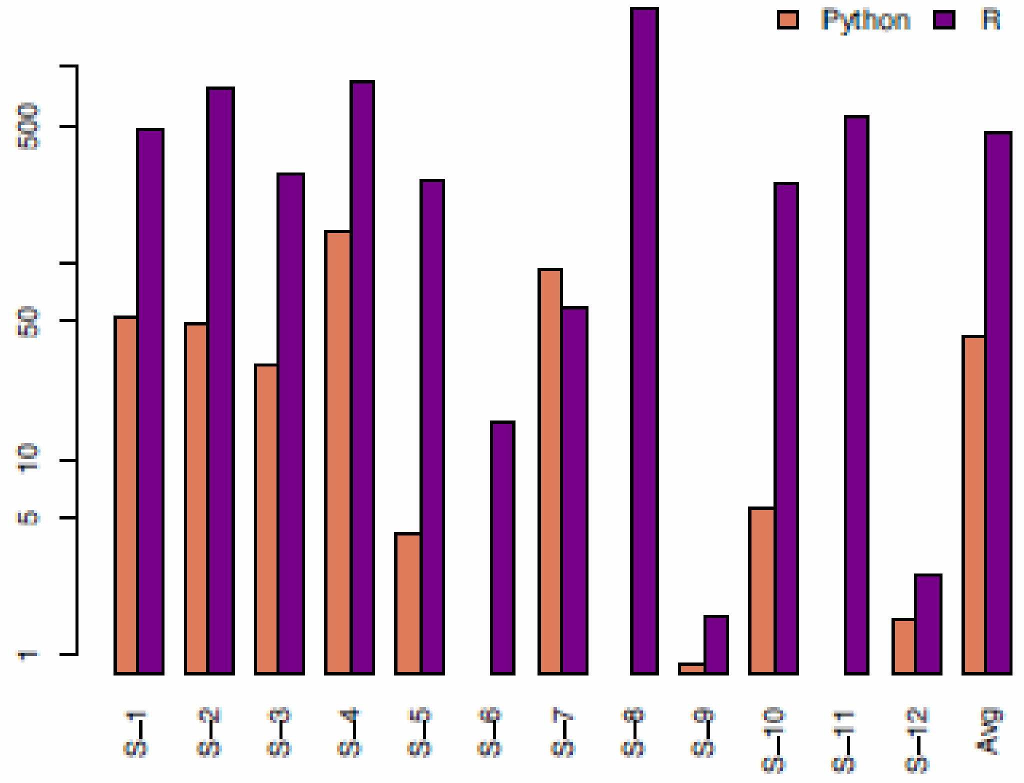
- Numerické a příbuzné aplikace
  - FORTRAN
    - Velká konkurence mezi překladači
    - Relativně slabý jazyk dovolující agresivní optimalizace
  - C/C++
    - Téměř stejně kvalitní překladače
    - Úspěšnější standardizace
    - Agresivní optimalizace vyžadují rozšíření jazyka
      - restrict/__restrict
      - #pragma omp/#pragma acc
  - Oba jazyky jsou relativně obtížné pro začátečníky
    - Ne-informatici žádají snadný start

Existují jiné jazyky než FORTRAN a C/C++?

	Fortran	Julia	Python	R	Matlab	Octave	Mathe-matica	Java-Script	Go	LuaJIT	Java
fib	0.70	2.11	77.76	533.52	26.89	9324.35	118.53	3.36	1.86	1.71	1.21
parse_int	5.05	1.45	17.02	45.73	802.52	9581.44	15.02	6.06	1.20	5.77	3.35
quicksort	1.31	1.15	32.89	264.54	4.92	1866.01	43.23	2.70	1.29	2.03	2.60
mandel	0.81	0.79	15.32	53.16	7.58	451.81	5.13	0.66	1.11	0.67	1.35
pi_sum	1.00	1.00	21.99	9.56	1.00	299.31	1.69	1.01	1.00	1.00	1.00
rand_mat_stat	1.45	1.66	17.93	14.56	14.52	30.93	5.95	2.30	2.96	3.27	3.92
rand_mat_mul	3.48	1.02	1.14	1.57	1.12	1.12	1.30	15.07	1.42	1.16	2.36

jednovláknový kód, implementace v C = 1.00
zdroj: MIT CSAIL, julialang.org

Existují jiné jazyky než FORTRAN a C/C++?



	Name	Input
S-1	Binary trees	16
S-2	Fankuch redux	10
S-3	Fasta	2.5M
S-4	Fasta redux	2.5M
S-5	K-nucleotide	50K
S-6	Mandelbrot	4K
S-7	N-body	500K
S-8	Pidigits	500
S-9	Regex-dna	2.5K
S-10	Rev. complement	5M
S-11	Spectral norm	640
S-12	Spectral norm alt	11K

Fig. 7. Slowdown of Python and R, normalized to C for the Shootout benchmarks

Zdroj: Morandat et al., Evaluating the Design of the R Language, ECOOP 2012

Existují jiné jazyky než FORTRAN a C/C++?

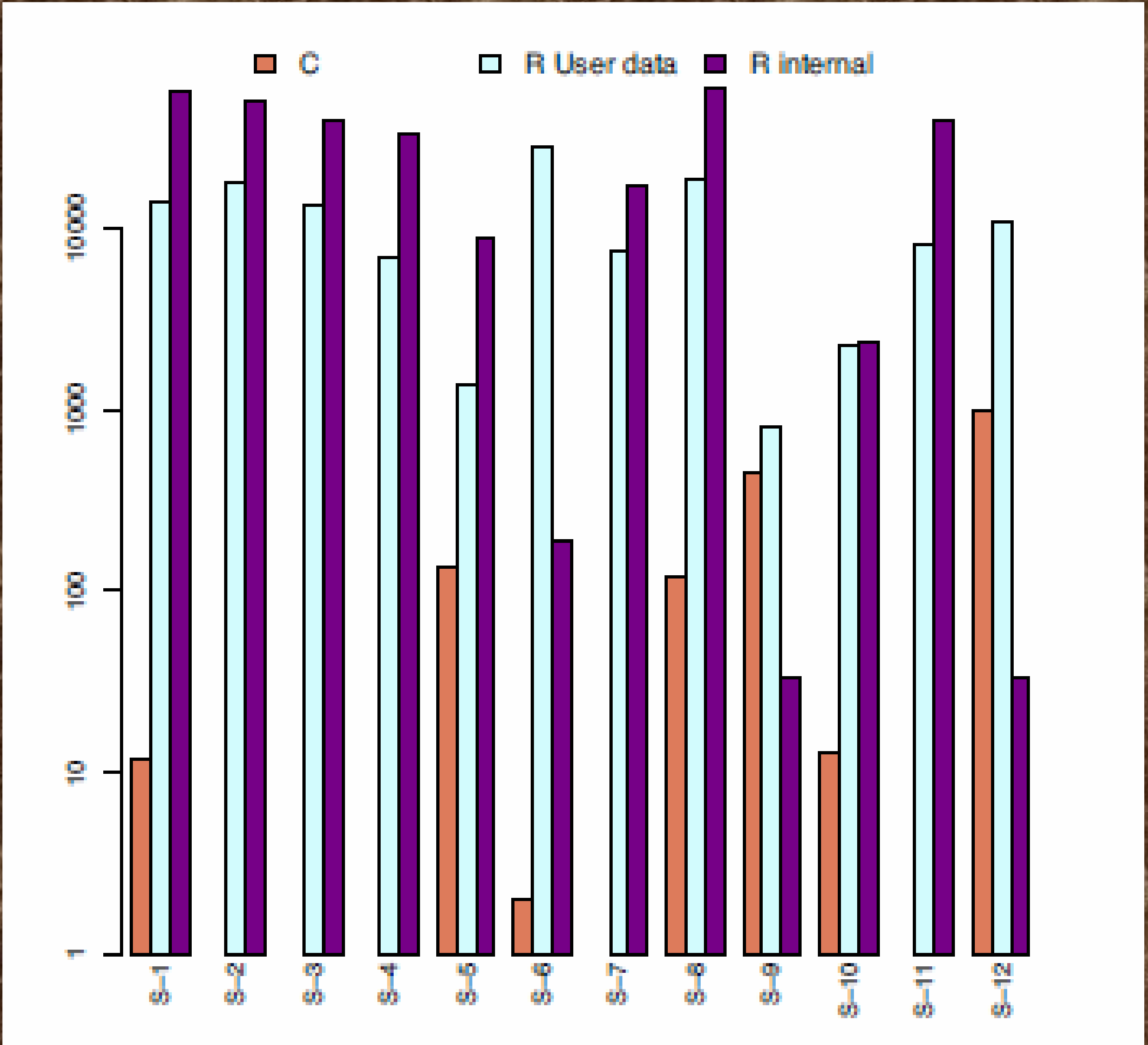


Fig. 9. Heap allocated memory (MB log scale). C vs. R.

Existují jiné jazyky než FORTRAN a C/C++?

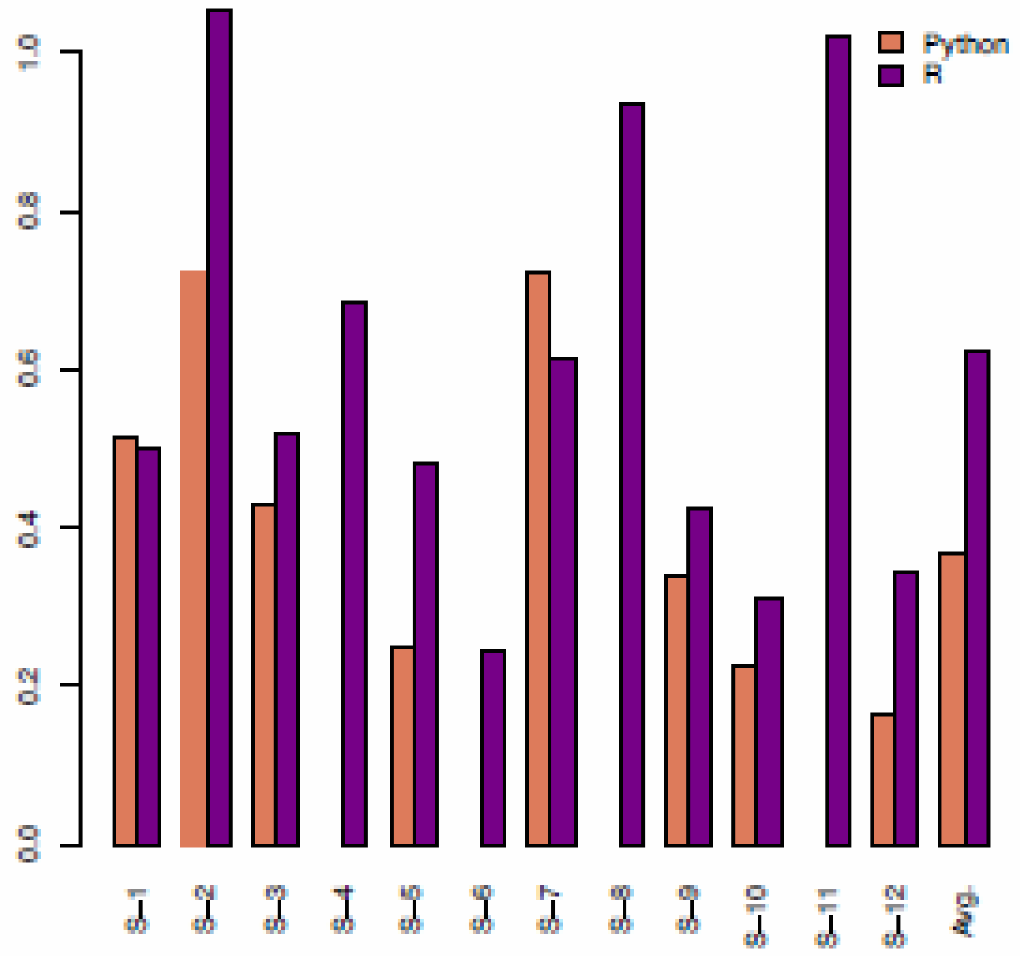


Fig. 10. Shootout Python and R code size, normalized to C

Příklad: (Ne-)výkonnost jazyka R

- Jazyk R by mohl být překladači optimalizován snadněji než C
  - Převládá funkcionální charakter
  - Lazy evaluation
  - Žádné ukazatele, aliasy etc.
  - Imutabilní datové struktury teoreticky umožňují snadnou (de)alokaci
- Jiné části R jsou však pro překladače neřešitelné
  - dynamické typy
  - netypický mechanismus tříd
  - eval
  - ohyzdnosti jako super-assignment
- Některé úlohy se v R programují velmi obtížně
  - Mutabilní datové struktury
    - Hashovací tabulky, AVL stromy, ...
  - Nevýpočetní části: I/O, GUI, ...

Rozšíření C++ pro řízení optimalizací (pro jednovláknový kód)

Integrace procedur

- Intel
    - #pragma inline/forceinline/noinline [recursive]
  - The forceinline pragma indicates that the calls in question should be inlined whenever the compiler is capable of doing so.
  - The inline pragma is a hint to the compiler that the user prefers that the calls in question be inlined, but expects the compiler not to inline them if its heuristics determine that the inlining would be overly aggressive and might slow down the compilation of the source code excessively, create too large of an executable, or degrade performance.
  - The noinline pragma indicates that the calls in question should not be inlined.
- gcc
    - void f() __attribute__((always_inline))
  - For functions declared inline, this attribute inlines the function independent of any restrictions that otherwise apply to inlining. Failure to inline such a function is diagnosed as an error.
    - void f() __attribute__((flatten))
  - Generally, inlining into a function is limited. For a function marked with this attribute, every call inside this function is inlined, if possible. Whether the function itself is considered for inlining depends on its size and the current inlining parameters.
    - void f() __attribute__((noinline))

Ignorování potenciálních závislostí

- pragma ivdep
  - Intel
    - #pragma ivdep
  - gcc
    - #pragma GCC ivdep
  - The ivdep pragma instructs the compiler to ignore assumed vector dependencies.
  - The proven dependencies that prevent vectorization are not ignored, only assumed dependencies are ignored.
  - In addition to the ivdep pragma, the vector pragma can be used to override the efficiency heuristics of the vectorizer.

```
void example(int *a, int k, int c, int m) {
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

Ignorování potenciálních aliasů

- SolarisCC (Oracle)
    - #pragma noalias (pointer, pointer [, pointer]…)
    - #pragma may_not_point_to (pointer, variable [, variable]…)
- C99 (gcc, Intel, MSVC)
    - __restrict (C++ MSVC)

```
void copy(int * restrict a, int * restrict b, int m) {
  for (int i = 0; i < m; i++)
    a[i] = b[i];
}
```

Vynucená vektorizace

- Intel
    - #pragma vector always
    - #pragma simd
  - Asks the compiler to vectorize the loop if it is safe to do so, whether or not the compiler thinks that will improve performance.
- OpenMP
    - #pragma omp declare simd
- gcc
    - void f() __attribute__((simd))
  - This attribute enables creation of one or more function versions that can process multiple arguments using SIMD instructions from a single invocation.

Parallelization/vectorization/optimization in C++

C++17 execution control

- C++17
    - already in TS 19570:2015 as experimental
  - <execution> - namespace std::execution
    - Types (tags)
      - class sequenced_policy { /* unspecified */ };
      - class parallel_policy { /* unspecified */ };
      - class parallel_unsequenced_policy { /* unspecified */ };
    - Constants – instances of tags used as arguments to functions
      - inline constexpr sequenced_policy seq { /* unspecified */ };
      - inline constexpr parallel_policy par { /* unspecified */ };
      - inline constexpr parallel_unsequenced_policy par_unseq { /* unspecified */ };
  - Additional execution policies may be provided by a standard library implementation (possible future additions may include std::parallel::cuda and std::parallel::opencl)

C++17 execution control

- Example – for_each
  - Old version (C++98, semantics changed in C++11)
    - template< class InputIt, class UnaryFunction >
      UnaryFunction for_each(InputIt first, InputIt last,
      UnaryFunction f);
    - Applies the given function object f to the result of dereferencing every iterator in the range [first, last), in order.
    - UnaryFunction must meet the requirements of MoveConstructible. Does not have to be CopyConstructible
  - C++17
    - template< class ExecutionPolicy, class ForwardIt, class
      UnaryFunction2 >
      void for_each(ExecutionPolicy&& policy,
        ForwardIt first, ForwardIt last, UnaryFunction2 f);
    - Applies the given function object f to the result of dereferencing every iterator in the range [first, last) (not necessarily in order).
    - The algorithm is executed according to policy.
    - UnaryFunction2 must meet the requirements of CopyConstructible.

C++17 execution control

- Effect of the tags
  - sequenced_policy
    - Parallel algorithm's execution may not be parallelized.
    - The invocations of element access functions in parallel algorithms invoked with this policy (usually specified as std::execution::seq) are indeterminately sequenced in the calling thread.
  - parallel_policy
    - Parallel algorithm's execution may be parallelized.
    - The invocations of element access functions in parallel algorithms invoked with this policy (usually specified as std::execution::par) are permitted to execute in either the invoking thread or in a thread implicitly created by the library to support parallel algorithm execution.
    - Any such invocations executing in the same thread are indeterminately sequenced with respect to each other.
  - parallel_unsequenced_policy
    - Parallel algorithm's execution may be parallelized, vectorized, or migrated across threads (such as by a parent-stealing scheduler).
    - The invocations of element access functions in parallel algorithms invoked with this policy are permitted to execute in an unordered fashion in unspecified threads, and unsequenced with respect to one another within each thread.
    - Unsequenced execution policy is the only case where function calls are unsequenced with respect to each other, meaning they can be interleaved. In all other situations in C++, they are indeterminately-sequenced (cannot interleave).
      - Because of that, users are not allowed to allocate or deallocate memory, acquire mutexes or perform any other vectorization-unsafe operations when using this policy (vectorization-unsafe functions are the ones that synchronize-with another function, e.g. std::mutex::unlock synchronizes-with the next std::mutex::lock)
  - During the execution of a parallel algorithm with any of these three execution policies, if the invocation of an element access function exits via an uncaught exception, std::terminate is called, but the implementations may define additional execution policies that handle exceptions differently.

C++17 execution control

- Parallel versions of algorithms
  - execution_policy argument added
    - adjacent_difference adjacent_find all_of any_of copy copy_if copy_n count count_if equal fill fill_n find find_end find_first_of find_if find_if_not generate generate_n includes inner_product inplace_merge is_heap is_heap_until is_partitioned is_sorted is_sorted_until lexicographical_compare max_element merge min_element minmax_element mismatch move none_of nth_element partial_sort partial_sort_copy partition partition_copy remove remove_copy remove_copy_if remove_if replace replace_copy replace_copy_if replace_if reverse reverse_copy rotate rotate_copy search search_n set_difference set_intersection set_symmetric_difference set_union sort stable_partition stable_sort swap_ranges transform uninitialized_copy uninitialized_copy_n uninitialized_fill uninitialized_fill_n unique unique_copy
- New parallel algorithms
  - for_each - similar to std::for_each except returns void
  - for_each_n - applies a function object to the first n elements of a sequence
  - reduce - similar to std::accumulate, except out of order
  - exclusive_scan - similar to std::partial_sum, excludes the ith input element from the ith sum
  - inclusive_scan - similar to std::partial_sum, includes the ith input element in the ith sum
  - transform_reduce - applies a functor, then reduces out of order
  - transform_exclusive_scan - applies a functor, then calculates exclusive scan
  - transform_inclusive_scan - applies a functor, then calculates inclusive scan

Sequenced-before

- value computation: calculation of the value that is returned by the expression. This may involve determination of the identity of the object (glvalue evaluation, e.g. if the expression returns a reference to some object) or reading the value previously assigned to an object (prvalue evaluation, e.g. if the expression returns a number, or some other value)
- side effect: access (read or write) to an object designated by a volatile glvalue, modification (writing) to an object, calling a library I/O function, or calling a function that does any of those operations.

- Sequenced-before rules (since C++11)
  - "sequenced-before" is an asymmetric, transitive, pair-wise relationship between evaluations within the same thread.
    - If A is sequenced before B, then evaluation of A will be complete before evaluation of B begins.
    - If A is not sequenced before B and B is not sequenced before A, then two possibilities exist:
      - evaluations of A and B are unsequenced: they may be performed in any order and may overlap (within a single thread of execution, the compiler may interleave the CPU instructions that comprise A and B)
      - evaluations of A and B are indeterminately sequenced: they may be performed in any order but may not overlap: either A will be complete before B, or B will be complete before A. The order may be the opposite the next time the same expression is evaluated.
  - Undefined behavior
    - If a side effect on a scalar object is unsequenced relative to another side effect on the same scalar object, the behavior is undefined.
    - If a side effect on a scalar object is unsequenced relative to a value computation using the value of the same scalar object, the behavior is undefined.

Sequenced-before

- Each value computation and side effect of a full expression, that is
    - unevaluated operand
    - constant expression
    - an entire initializer, including any comma-separated constituent expressions
    - the destructor call generated at the end of the lifetime of a non-temporary object
    - an expression that is not part of another full-expression (such as the entire expression statement, controlling expression of a for/while loop, conditional expression of if/switch, the expression in a return statement, etc),
    - including implicit conversions applied to the result of the expression, destructor calls to the temporaries, default member initializers (when initializing aggregates), and every other language construct that involves a function call,
  - is sequenced before each value computation and side effect of the next full expression.
- The value computations (but not the side-effects) of the operands to any operator are sequenced before the value computation of the result of the operator (but not its side-effects).
- When calling a function (whether or not the function is inline, and whether or not explicit function call syntax is used), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function.

Sequenced-before

- The value computation of the built-in post-increment and post-decrement operators is sequenced before its side-effect.
- The side effect of the built-in pre-increment and pre-decrement operators is sequenced before its value computation (implicit rule due to definition as compound assignment)
- Every value computation and side effect of the first (left) argument of the built-in logical AND operator && and the built-in logical OR operator || is sequenced before every value computation and side effect of the second (right) argument.
- Every value computation and side effect associated with the first expression in the conditional operator ?: is sequenced before every value computation and side effect associated with the second or third expression.
- The side effect (modification of the left argument) of the built-in assignment operator and of all built-in compound assignment operators is sequenced after the value computation (but not the side effects) of both left and right arguments, and is sequenced before the value computation of the assignment expression (that is, before returning the reference to the modified object)
- Every value computation and side effect of the first (left) argument of the built-in comma operator , is sequenced before every value computation and side effect of the second (right) argument.
- In list-initialization, every value computation and side effect of a given initializer clause is sequenced before every value computation and side effect associated with any initializer clause that follows it in the brace-enclosed comma-separated list of initalizers.
- When returning from a function, copy-initialization of the temporary that is the result of evaluating the function call is sequenced-before the destruction of all temporaries at the end of the operand of the return statement, which, in turn, is sequenced-before the destruction of local variables of the block enclosing the return statement.

- A function call that is not sequenced before or sequenced after another function call is indeterminately sequenced (the program must behave as if the CPU instructions that constitute different function calls were not interleaved, even if the functions were inlined).
  - The rule has one exception: a function calls made by a standard library algorithm executing under std::par_unseq execution policy are unsequenced and may be arbitrarily interleaved.

Sequenced-before

- **Additional rules - C++17**
  - In a function-call expression, the expression that names the function is sequenced before every argument expression and every default argument.
  - In a function call, value computations and side effects of the initialization of every parameter are indeterminately sequenced with respect to value computations and side effects of any other parameter.
  - In a subscript expression E1[E2], every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2
  - In a pointer-to-member expression E1.*E2 or E1->*E2, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2 (unless the dynamic type of E1 does not contain the member to which E2 refers)
  - In a shift operator expression E1<<E2 and E1>>E2, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2
  - The call to the allocation function (operator new) is sequenced before (since C++17) the evaluation of the constructor arguments in a new-expression
  - Every expression in a comma-separated list of expressions in a parenthesized initializer is evaluated as if for a function call (indeterminately-sequenced)
- Every overloaded operator obeys the sequencing rules of the built-in operator it overloads when called using operator notation.
- In every simple assignment expression E1=E2 and every compound assignment expression E1@=E2, every value computation and side-effect of E2 is sequenced before every value computation and side effect of E1
  - i = ++i + 2;   // undefined behavior until C++11
  - i = i++ + 2;   // undefined behavior until C++17
  - f(i = -2, i = -2); // undefined behavior until C++17
  - f(++i, ++i);   // undefined behavior until C++17, unspecified after C++17
  - i = ++i + i++;   // undefined behavior