```
--------------------------------------------------------------------------------
================================================================================
HACKING C++
https://hackingcpp.com/cpp/beginners_guide.html
================================================================================

--------------------------------------------------------------------------------
FIRST STEPS
--------------------------------------------------------------------------------

    ----------------------------------------------------------------------------
    Hello World
    ----------------------------------------------------------------------------

        $ vi hello.cpp

            #include <iostream>                         // (1)
            // our first program
            int main()                                  // (2)
            {
                std::cout << "Hello World\n";           // (3)
            }

        - Parts

            (1) #include <iostream>
                - Replaced with the content of the file iostream
                - iostream
                    - Header file in the compiler directory
                    - Provides input and output functionalities
                - Difference
                    (a) #include "path/to/file"
                            - Inserts the content of a file
                    (b) #include <filename>
                            - Same, but searches for file in all 'include dirs'


            (2) int main()
                - Defines function called "main"
                - Every program starts by executing the main function
                - int
                    - The only allowed return type for the main function
                - ()
                    - Empty parameter list

            (3) std::cout << "Hello World\n";
                - std
                    - Namespace of the standard library
                - cout
                    - Character out
                    - Refers to the standard (console) output

    ----------------------------------------------------------------------------
    Compliling "hello.cpp"
    ----------------------------------------------------------------------------

        $ g++ hello.cpp -o sayhello
        $./sayhello

    - Process
        (1) Source Code
        (2) Preprocessor
        (3) Compiler
        (4) Linker
        (5) Binary Executable

    - Terminology
        - Compile Error
```

```
        - Compiler Warning
        - static
            - fixed at compile time
        - dynamic
            - changeable at runtime

    - Compiler Flags
        - Recommended for g++

                -std=c++20      ..set compiler standard
                -Wall           ..warnings
                -Wpedantic
                -Wextra
                -Wshadow
                -o <filename>   ..name of output file

    - Don't use 'using namespace std;'

            using namespace std;
            int main() {
                cout << "bla\n";
            }

        - Polluting the global namespace with symbols from other namespaces
            - Serious liability
            - Can lead to name conflicts and ambiguities

    ----------------------------------------------------------------------------
    Input & Output (Basics)
    ----------------------------------------------------------------------------

    - I/O Streams

            #include <iostream>

            int main() {
                int i;
                std::cin >> i;
                std::cout << i << '\n';
            }

        - Sources and Targets

            std::cin    ..chars from stdin:
                            - Reads from buffer
            std::cout   ..chars to stdout
                            - Writes to buffer first
                            - Output to console when buffer full
            std::clog   ..chars to stderr
                            - Writes to buffer first
                            - Output to console when buffer full
            std::cerr   ..chars to stderr
                            - Immediatly writes to console

    - Stream Operators

            >>      "get from"      source >> target
            <<      "put to"        target << source

        - Chaining

            int i = 0;
            double d =0.0;

            // read to values of different type:
            std:: cin >> i >> d;

            // print two values and some text:
```

```
        std: cout << "your input:\n"
                  << i << " and " << d << '\n';
```

- Avoid std::endl

```
        std::cout << "some text" << std::endl;
        std::cout << "more text" << std::endl;
```

  - Each call to std:endl
    - Flushes the output buffer
    - Writes the output immediately
  - Can lead to serious performance degradation
  - Only usecase
    - Making sure that some output needs to materialize immediately

- C++'s I/O streams
  - Use buffers
    - To mitigate the performance impact of system input or output operations
    - Output
      - Is collected until a minimum number of chars can be written

- Do this!

```
        std::cout << "some text\nmore text\n";
```

  - Using line break
  - No premature flush of output buffer
  - Only one call to operator <<
    - Each additional call creates a small overhead

Fundamental Types

- Variable Declarations

```
        type variable = value;

        type variable {value};  // C++11
```

  - Example

```
        char c = 18

            Variable: named Object
            Object: piece of memory that holds a value of some type
            Value: set of bits interpreted according to a type

                        18dec   00010010bin

            Type: possible values and operations

                        -128 ... 127
                        + * - / %
```

  - Variables should almost always be initialized when declared to prevent bugs

- Quick Overview

  Booleans

```
            bool b1 = true;
            bool b2 = false;
```

  Characters

```
        - smallest integer; usually 1 byte
        - on x86/x86_64 signed -> values e [-128, 127]

            char c = 'A';       // character literal
            char a = 65;        // same as above

    Signed Integers

        - n bits -> values e [-2^(n-1), 2^(n-1) -1]

            short s  = 7;
            int   i  = 12347;
            long  l1 = -7856974990L;
            long long l2 = 89565656974990LL;

            // ' digit separator C++14
            long l3 = 512'323'697'499;

    Unsigned Integers

        - n bits -> values e [0, 2^n -1]

            unsigned u1 = 12347U;
            unsigned long u2 = 123478912345UL;
            unsigned long long u3 = 123478912345ULL;

            // non-decimal literals
            unsigned x = 0x4A;          // hexadecimal
            unsigned b = 0b10110101;    // binary C++14

    Floating Point Types

        - float usually IEEE 754 32 bit
        - double usually IEEE 754 64 bit
        - long double usually 80-bit on x86/x86_64

            float       f  = 1.88f;
            double      d1 = 3.5e38;
            long double d2 = 3.5e38L;   // C++11

            // ' digit separator C++14
            double d3 = 512'232'697'499.052;

- Arithmetic Operations

        a (op) b        ..returns result of operation (op) applied to the
                          values of a and b
        a (op)= b       ..stores result of operation (op) in a

    - Examples

        int a = 4;          // variable a is set to value 4
        int b = 3;          // variable b is set to value 3

        a = a + b;          // a: 7     add
        a += b;             // a: 10

        a = a - b;          // a: 7     subtract
        a -= b;             // a: 4

        a = a * b;          // a: 12    multiply
        a *= b;             // a: 36

        a = a / b;          // a: 12    divide
        a /= b;             // a: 4

        a = a % b;          // a: 1     remainder of division (modulo)
```

```
    - Increment/Decrement

        - Changes value by +/- 1
        - ++x / --x
            - Prefix expression
            - Returns new (incremented/decremented) value
        - x++ / x--
            - Postfix expression
            - Increments/decrements value, but returns old value
        - Examples

            int a = 4;          // a: 4
            int b = 3;          //          b:3

            b = a++;            // a: 5     b: 4
            b = ++a;            // a: 6     b: 6

            b = --a;            // a: 5     b: 5
            b = a--;            // a: 4     b: 5

    - Comparisons

        2-way Compariosons

            - Result of comparison is either true or false
            - Examples

                int x = 10;
                int y = 5;
                                        // result   operator
                bool b1 = x == 5;       //  false    equals
                bool b2 = (x != 6);     //  true     not equal

                bool b3 = x > y;        //  true     greater
                bool b4 = x < y;        //  false    smaller
                bool b5 = y >= 5;       //  true     greater/equal
                bool b6 = x <= 30;      //  true     smaller/equal

        3-way Comparisons With <=>      // C++20

            - Determines the relative ordering of 2 objects

                (a <=> b) < 0           if a < b
                (a <=> b) > 0           if a > b
                (a <=> b) == 0          if a and b are equal/equivalent

            - Return a comparison category value that can be compared to
              literal 0
                - The returned value comes from one of three possible categories

                    (1) std::strong_ordering
                    (2) std::weak_ordering
                    (3) std::partial_ordering

            - Examples

                4 <=> 6     -> std::strong_ordering::less
                5 <=> 5     -> std::strong_ordering::equal
                8 <=> 1     -> std::strong_ordering::greater

    - Boolean Logic

        Operators

            bool a = true;
            bool b = false;

            bool c = a && b;    // false    logical AND
```

```
            bool d = a || b;    // true     logical OR
            bool e = !a;        // false    logical NOT

            // Alternative Spellings:
            bool x = a and b;   // false
            bool y = a or b;    // true
            bool z = not a;     // false

        Conversion to bool

            - 0 is always false;
            - Everything else is true;
            - Examples

                bool f = 12;    // true     (int -> bool)
                bool g = 0;     // false    (int -> bool)
                bool h = 1.2;   // true     (double -> bool)

        Short-circuit Evaluation

            - The second operand of a boolean comparison is not evaluated if the
              result is already known after evaluating the first operand
            - Examples

                int i = 2;
                int k = 8;
                bool b1 = (i >0) || (k < 3);    // i > 0 is true;
                                                // k < 3 is not evaluated
                                                // because  result of logical OR
                                                // is already true

    - Memory Sizes of Fundamental Types

        - All type sizes are a multiple of sizeof(char)

                cout << sizeof(char);   // 1
                cout << sizeof(bool);   // 1
                cout << sizeof(short);  // 2
                cout << sizeof(int);    // 4
                cout << sizeof(long);   // 8

                // number of bits in a char
                cout << CHAR_BIT;       // 8

                char    c = 'A';
                bool    b = true;
                int     i = 1234;
                long    l = 12;
                short   s = 8;


                 ---+----------+                     +----------+
                    | 00000000 | 0x21                |          | 0x2F
                    +----------+                     +----------+
                    | 00000000 | 0x20                |          | 0x2E
                  i +----------+                     +----------+
                    | 00000100 | 0x1F                |          | 0x2D
                    +----------+                     +----------+
                    | 11010010 | 0x1E                |          | 0x2C
                 ---+----------+                  ---+----------+
                    |          | 0x1D                | 00000000 | 0x2B
                    +----------+                   s +----------+
                    |          | 0x1C                | 00001000 | 0x2A
                    +----------+                  ---+----------+
                    |          | 0x1B                | 00000000 | 0x29
                 ---+----------+                     +----------+
                  b | 00000001 | 0x1A                | 00000000 | 0x28
                 ---+----------+                     +----------+
```

```
                |          | 0x19                 | 00000000 | 0x27
                +----------+                      +----------+
                |          | 0x18                 | 00000000 | 0x26
                +----------+                    l +----------+
                |          | 0x17                 | 00000000 | 0x25
             ---+----------+                      +----------+
              c | 01000001 | 0x16                 | 00000000 | 0x24
             ---+----------+                      +----------+
                |          | 0x15                 | 00000000 | 0x23
                +----------+                      +----------+
                |          | 0x14                 | 00001100 | 0x22
                +----------+                   ---+----------+
```

- Sizes Are Platform Dependent
    - Only basic guarantees

```
sizeof(short) >= sizeof(char)
sizeof(int)   >= sizeof(short)
sizeof(long)  >= sizeof(int)
```

    - Example
        - On some 32-bit platforms: int = long

- Integer Size Guarantees C++11

```
#include <cstdint>
```

    - Exact size (not available on some platforms)

```
int8_t, int16_t, int32_t, int64_t, uint8_t, ...
```

    - Guaranteed minimum size

```
int_least8_t, uint_least8_t, ...
```

    - Fastest with guaranteed minimum size

```
int_fast8_t, uint_fast8_t, ...
```

- Fixed-Width Floating Point Type Guarantees C++23

```
# include <stdfloat>

// storage bits: sign + exponent + mantissa
std::float16_t  a = 12.3f16;        // 1 +  5 +  10 =  16 bits = 2 B
std::float32_t  b = 12.3f32;        // 1 +  8 +  23 =  32 bits = 4 B
std::float64_t  c = 12.3f64;        // 1 + 11 +  52 =  64 bits = 8 B
std::float128_t d = 12.3f128;       // 1 + 15 + 112 = 128 bits = 16B
std::bfloat16_t e = 12.3b16;        // 1 +  8 +   7 =  16 bits = 2 B
```

- std::numeric_limits<type>

```
#include <limits>

// smallest negative value:
std::cout << std::numeric_limits<double>::lowest();

// float/double: smallest value > 0
// integers: smallest value
std::cout << std::numeric_limits<double>::min();

// largest positive value:
std::cout << std::numeric_limits<double>::max();

// smallest difference btw. 1 and next value:
std::cout << std::numeric_limits<double>::epsilon();
```

- Reference

```
            https://en.cppreference.com/w/cpp/types/numeric_limits

    - Type Narrowing

        - Conversion from type that can represent more values to one that can
          represent less
        - May result in loss of information
        - In general no compiler warning
        - Potential source of subtle runtime bugs
        - Examples

            double   d = 1.23456;
            float    f = 2.53f;
            unsigned u = 120u;

            double e = f;           // OK float -> double

            int i = 2.5;            // NARROWING double -> int
            int j = u;              // NARROWING unsigned int -> int
            int k = f;              // NARROWING float -> int

        - Braced Initialization C++11

                type variable { value };

            - Works for all fundamental types
            - Narrowing conversion -> compiler warning
            - Example

                double   d {1.23456};   // OK
                float    f {2.53f};     // OK
                unsigned u {120u};      // OK

                double e {f};           // OK

                int i {2.5};            // COMPILER WARNING: double       -> int
                int j {u};              // COMPILER WARNING: unsigned int -> int
                int k {f};              // COMPILER WARNING: float        -> int

            - Prevent silent type conversions!
                - Especially narrowing unsigned to signed integer conversions
                    - Hard-to-find runtime bugs!

    - Bitwise Operations

        Bitwise Logic

                a & b   bitwise AND
                a | b   bitwise OR
                a ^ b   bitwise XOR
                ~a      bitwise NOT (one's complement)

            - Example

                #include <cstdint>                        memory bits:

                std::uint8_t a = 6;                       0000 0110
                std::uint8_t b = 0b00001011;              0000 1011

                std::uint8_t c1 = (a & b);      // 2      0000 0010
                std::uint8_t c2 = (a | b);      // 15     0000 1111
                std::uint8_t c3 = (a ^ b);      // 13     0000 1101

                std::uint8_t c4 = ~a;           // 249    1111 1001
                std::uint8_t c5 = ~b;           // 244    1111 0100

                // test if int is even/odd:               result:
```

```
                bool a_odd  = a & 1;                    0 -> false
                bool a_even = !(a & 1);                 1 -> true

        Bitwise Shifts

                x << n      ..retruns x’s value with its bits shifted by
                              n places to the left
                x >> n      ..retruns x’s value with its bits shifted by
                              n places to the right
                x <<= n     ..modifies x by shifting bits by n places (left)
                x >>= n     ..modifies x by shifting bits by n places (right)

            - Example

                #include <cstdint>                      memory bits:

                std::uint8_t a = 1;                     0000 0001
                a <<= 6;                // 64           0100 0000
                a >>= 4;                //  4           0000 0100

                std:uint8_t b1 = (1 << 1);  //  2       0000 0010
                std:uint8_t b2 = (1 << 2);  //  4       0000 0100
                std:uint8_t b3 = (1 << 4);  // 16       0001 0000

            - Warning:
                - Shifting the bits of an object whose type has N bits by N or
                  more than N places is undefined behavior!

                    std::uint32_t i = 1;        // 32 bit type
                    i <<= 32;                   // UNDEFINED BEHAVIOR!
                    std::uint64_t j = 1;        // 64 bit type
                    j <<= 70;                   // UNDEFINED BEHAVIOR!

    - Arithmetic Conversions & Promotions

        - Lot of rules (go back to C)
            - Purpose:
                - Determine a common type for both operands and the result of a
                  binary operation

                        Operand A (op) Operand B -> Result

        A Simplified Summary

            - Operations Invloving At Least One Floating-Point Type

                    long double (op) any other type -> long double
                    double (op) float -> double
                    double (op) any integer -> double
                    float (op) any integer -> float

            - Operations On Two Integer Types

                (1) Integer Promotion
                        - Applied first to both operands
                        - Basically everything smaller than int gets promoted to
                          either int or unsigned int

                (2) Integer conversion
                        - Applied if both operand types are different
                        - Both signed:
                            - Smaller type converted to larger
                        - Both unsigned:
                            - Smaller type converted to larger
                        - Signed (op) unsigned:
                            (a) signed converted to unsigned if both have same
                                width
                            (b) otherwise unsigned converted to signed if that
```

```
                              can represent all values
                          (c) otherwise both converted to unsigned


    ---------------------------------------------------------------------------
    Introduction to std::vector
    ---------------------------------------------------------------------------

    - array
        - Can hold different values/objects of same type
    - dynamic
        - Size can be changed at runtime
    - Guidelines, best practices and common mistakes:

            https://hackingcpp.com/cpp/std/vector.html

    - Initialization / Access

            #include <vector>

            // Initialization with 3 elements
            std::vector<int> v {2, 7, 9};
            // Number of elements
            std::cout << v.size() << '\n';

            std::cout << v[0] << '\n';              // 2
            std::cout << v[1] << '\n';              // 7

            // Assign new value
            v[1] = 4;
            std::cout << v[1] << '\n';              // 4

            std::cout << v.front() << '\n';         // 2 (first element)
            std::cout << v.back() << '\n';          // 9 (last element)

        - CAREFUL!

            vector<int> v1 {5,2};   ->  5 2
            vector<int> v2 (5,2);   ->  2 2 2 2 2
                            / \
          number of elements   default element value

    - Appending Elements

            vector<T>::push_back(Element)

        - Example

            std::vector<int> v;
            cout << v.size() << '\n';           // 0

            v.push_back(2);                     // 2
            cout << v.size() << '\n';           // 1

            v.push_back(7);                     // 2 7
            cout << v.size() << '\n';           // 2

            v.push_back(9);                     // 2 7 9
            cout << v.size() << '\n';           // 3

    - Resizing

            vector<T>::resize(new_number_of_elements, filler_value=T{})

        - Example

            std::vector<int> v {1,2};           // 1 2
            v.push_back(3);                     // 1 2 3
```

```
        cout << v.size() << '\n';          // 3 elements

        v.resize(6, 0);                    // 1 2 3 0 0 0
        cout << v.size() << '\n';          // 6 elements
```

- Erasing Elements (at the end)

```
        vector<T>::pop_back()
        vector<T>::clear()
```

  - Example

```
        std::vector<int> v {1,2,3,4,5,6};  // 1 2 3 4 5 6
        cout << v.size() << '\n';          // 6

        v.pop_back();                      // 1 2 3 4 5
        cout << v.size() << '\n';          // 5

        v.pop_back();                      // 1 2 3 4
        cout << v.size() << '\n';          // 4

        v.clear();
        cout << v.size() << '\n';          // 0
```

- Copies Are Always Deep!

  - vector
    - A so-called regular type
      - i.e., it "behaves like int" in the following ways:
        (1) deep copying
              - Copying creates a new vector object and copies all contained objects
        (2) deep assignment
              - All contained objects are copied from source to assignment target
        (3) deep comparison
              - Comparing two vectors compares the values of the contained objects
        (4) deep ownership
              - Destroying a vector destroys all contained objects
      - Most types in the C++ standard library and ecosystem are regular
  - Example

```
        std::vector<int> a {1,2,3,4};      // a: 1 2 3 4
        std::vector<int> b = a;            // b: 1 2 3 4

        if (a == b) cout << "equal\n";     // equal

        a[0] = 9;                          // a: 9 2 3 4
                                           // b: 1 2 3 4
        cout << b[0] << '\n';              // 1
        if (a != b) cout << "different\n"; // different
```

  - WARNING
    - Copying vectors can be quite expensive (= take a long time) if
        (a) containing many elements
        (b) contained type is expensive to copy

Enumerations

- Scoped Enumerations C++11

```
        enum class name { enumerator1, enumerator2, ... enumeratorN };
```

```
        - Default: each enumerator is mapped to a whole number from 0 to N-1
        - Example

            enum class day { mon, tue, wed, thu, fri, sat, sun };
            day d = day::mon;
            d = day::tue;       // works
            d = wed;        // COMPILER ERROR: 'wed' only known in day's scope

    - Unscoped Enumerations

            enum name { enumerator1, enumerator2, ... enumeratorN };

        - Note: the absence of the keyword "class"
        - Example

            enum day { mon, tue, wed, thu, fri, sat, sun };
            day d = mon;    // OK!, enumerator "mon" unscoped
            int i = wed;    // OK!, i=2
            enum stars { sun, ... };    // COMPILER ERROR: name collision

        - Enumerators not confined to a scope -> name collisions
        - Dangerous implicit conversion to underlying type
        - Cannot query properties of enumeration as in some other languages
        - AVOID UNscoped enumerations

    - Underlying Type Of Enumerations

        - Must be an integer type (char, short, long, ...)
            - int is the default
        - Example

            // 7 values -> char should be enough
            enum class day : char {
                mon, tue, wed, thu, fri, sat, sun
            };
            // less than 10,000 -> short should be enough
            enum class language_ISO639 : short {
                abk, aar, afr, aka, amh, ara, arg, ...
            };

    - Custom Enumerator Mapping

        - Enumerator values can be set explicitly
        - Need not start with 0
        - Some values can be left out
        - Can be partial (only some enumerators with expl. value)
        - TIP
            - If you set enumerator values explicitly, do it for ALL enumerators
        - Examples

            enum class month {
                jan = 1, feb = 2, mar = 3, apr = 4, ... dec = 12
            };
            enum class flag {
                A = 2, B = 8, C = 5, D, E, F = 25
            };

    - Conversions To/From Underlying Type

            enum class month {
                jan = 1, feb = 2, mar = 3, apr = 4, ... dec = 12
            };

        - Enum -> Integer

            int i = static_cast<int>(month::mar);
            // i: 3
```

```
        - Integer -> Enum

            int i = 0;
            cin >> i;
            // make sure i >= 1 and <= 12
            month m1 =  static_cast<month>(i);


    ----------------------------------------------------------------------------
    Control Flow (Basics)
    ----------------------------------------------------------------------------

    - Terminology
        - Expressions
            - Series of computations (operators + operands)
            - May produce a result
        - Statements
            - Program fragments that are evaluated in sequence
            - Do not produce a result
            - Can contain one or multiple expressions
            - Delimited by ; and grouped by { }

    - Conditional Branching

            if (condition1) {
                // do this if condition1 is true
            }
            else if (condition2) {
                // else this if condition2 is true
            }
            else {
                // otherwise do this
            }

        - Code is (not) executed based on result of condition
        - Result of condition expression must be (convertible to) a boolean val
        - Conditions will be checked from to to bottom
        - Examples

            if (true)   { cout << "yes\n";} // yes
            if (false)  { cout << "yes\n";} // -
            if (2)      { cout << "yes\n";} // yes  (23 -> true)
            if (0)      { cout << "yes\n";} // -    ( 0 -> false)

        - Example

            int i = 0;
            cin >> i;
            if (i < 0) {
                cout << "negative\n";
            } else if (i == 0) {
                cout << "zero\n";
            } else {
                cout << "positive\n";
            }

        - if(statement; condition) { ... }  C++17

            - Useful for limiting the scope of temporary variables
            - Example

                int i = 0;
                std::cin >> i;
                if (int x = 2*i; x > 10) { cout << x; }

    - Switching: Value-Based Branching

        - Over values of integer types (char, int, long, enums, ...)
```

```
        - Checked & executed from top to bottom
        - Executes everything between matching case and next break (or the
          closing "}")
        - Example

                int i = 0;
                cin >> i;
                int m = i % 5;
                switch (m) {
                    case 0:     // do this if m is 0
                        break;
                    case 1:     // do this if m is 1
                    case 3:     // do this (also) if m is 1 or 3
                        break;
                    default:    // do this if m is NOT 0, 1 or 3
                }

        - switch (statement; variable) { ... }  C++17
            - Useful for limiting the scope of temporary variables
            - Example

                int i = 0;
                std::cin >> i;
                switch (int k = 2*i; k) { ... }

    - Ternary Conditional Operator

            Result = Condition ? If-Expression : Else-Expression

        - Examples

            int i = 8;
            int j = i > 10 ? 1 : 2;         // j: 2

            int k = 20;
            int l = (k > 10) ? 1 : 2;       // l: 1

            int b = true;
            double d =  b ? 2.0 : 0.5;      // d: 2.0
            double e = !b ? 2.0 : 0.5;      // e: 0.5

    - Loop Iteration

        Range-Based Loops   C++11

            for (variable : range) { ... }

                - range = object with standard iterator interface
                    - e.g., std::vector
                - Example

                    std::vector<int> v {1,2,3,4,5};
                    // print all elements of vector to console
                    for (int x : v) { std::cout << x << '\n'; }

        for (initialization; condition; step) { ... }

            - Example

                    // prints 0 1 2 3 4
                    for (int i = 0; i < 5; ++i) {
                        std::cout << i << ' ';
                    }

        while (condition) { ... }

            - first check of condition: before first loop iteration
            - Example
```

```
                int j = 5;
                while (j < 10) {
                    std::cout << j << ' ';
                    ++j;
                }

    do { ... } while (condition);

        - first check of condition: after first loop iteration
        - Example

                int j = 10;
                do {
                    std::cout << j << ' ';
                    --j;
                } while (j > 0);

    TIPS
        - Only write loops if there is no (standard) library funciton/
          algorithm for what needs to be done
        - Prefer range-based loops over all other types of loops!
            - No indexing/condition bugs possible
        - Use (do) while loops only, if the number of iterations is not
          known beforehand!


--------------------------------------------------------------------------------
Type System Basics
--------------------------------------------------------------------------------

- Declare Constants With const

        Type const variable_name = value;

    - Value can't be changed once assigned
    - Initial value can be dynamic (= set at runtime)
    - Example

        int i = 0;
        cin >> i;
        int const k = i;    // "int constant"

        k = 6;      // COMPILER ERROR: k is const!

    - TIP
        - Always declare variables as const if their values does not need to
          be changed after the initial assignment
            - Avoids bugs: does not compile if accidentally changed later
            - Helps understanding the code
            - Can improve performance

- Type Aliases

        using NewType = OldType;    C++11

        typedef OldType NewType;    C++98

    - Examples

        using real = double;
        using ullim = std::numeric_limits<unsigned long>;
        using index_vector = std::vector<std::uint_least64_t>;

    - TIP
        - Prefer 'using' over 'typedef'

- Type Deduction: auto      C++11
```

```
        auto variable = expression;

    - Varriable type deduced from right hand side of assignment
    - Often more convenient, safer and future proof
    - Important for generic (type independent) programming
    - Examples

        auto i = 2;                 int
        auto u = 56u;               unsigned int
        auto d = 2.023;             double
        auto f = 4.01f;             float
        auto l = -78787879797878l;  long int

        auto x = 0 * i;             x: int
        auto y = i + d;             y: double
        auto z = f * d;             z: double

- Constant Expressions: constexpr   C++11

    - Must be computable at compile time
    - Can be computed at runtime if not  invoked in a constexpr context
    - Expressions inside a constexpr context must be constexpr themselves
    - constexpr functions may contain
        - C++11 nothing but one return statement
        - C++14 multiple statements
    - Examples

        // two simple functions:
        constexpr int cxf (int i) { return i*2; }
                  int foo (int i) { return i*2; }

        constexpr int i = 2;        // OK '2' is a literal

        constexpr int j = cxf(5);   // OK, cxf is constexpr
        constexpr int k = cxf(i);   // OK, cxf and i are constexpr

                  int x = 0;        // not constexpr
                  int l = cxf(x);   // OK, not a constexpr context

        // constexpr contexts:
        constexpr int m = cxf( x );
        constexpr int n = foo( 5 );


----------------------------------------------------------------------------
Functions (Basics)
----------------------------------------------------------------------------

        return_type name (parameters) { body }

        // "call" at "call site"
        auto result = name( arguments )

Inputs & Outputs

- First Example
    - Function that computes mean of 2 numbers

        double mean (double a, double b) {
            return (a + b) / 2;
        }
        int main () {
            std::cout << mean (2, 6) << '\n';   // prints 4

- Return Types
    - Either one value: int, double, ...
        - Example
```

```
            double square (double x) {
                return (x * x);
            }

        - Example

            int max (int x, int y) {
                if (x > y) return x; else return y;
            }

    - Or nothing: void
        - Example

            void print_squares (int n) {
                for (int i = 1; i <= n; ++i)
                    cout << square(i) << '\n';
            }

    - Full Return Type Deduction    C++14
        - Deduction = compiler determines type automatically
        - Example

            auto foo (int i, double d) {
                ...
                return i;
            }

        - ERROR: Inconsistent return types!

            auto foo (int i, double d) {
                return i;   // int
                ..
                return d;   // double
            }

- Parameters
    - None:

            f()

    - One or many:

            g(int a, double b, int c, ...)

    - Parameter names have to be unique within list
    - const Parameters
        - Example

            int foo (int a, int const b) {
                a += 5;      // Correct
                b += 10;     // COMPILER ERROR: can't modify const parameter
                return (a + b);
            }

            // calling foo:
            foo(2,9);       // const has no effect here

            // Any 2nd argument passed to foo will be copied into the local
            // variable b -> the fact that b is const has no effect outside
            // of foo

        - If a value of a parameter inside a function does not need to or
          must not be changed make it const!

- Defaulted Parameters
    - Example
```

```
                double f (double a, double b = 1.5) {
                    return (a * b);
                }

                int main () {
                    cout << f(2);        // 1 argument  -> 3.0
                    cout << f(2, 3);     // 2 arguments -> 6.0
                }
```

- IMPORTANT
    - Each parameter after first default must have default value, too!

```
                // OK:
                void foo (int = 0)
                void foo (int n, double x = 2.5)
                void foo (int a, int b = 1, float c = 3.5)

                // NOGO:
                void foo (int a, int b = 1, int c)
```

- Overloading
    - Functions with the same name but different parameter lists
    - Cannot overload on return type alone
        - Example: OK

```
                // OK: same name, different parameter lists
                int abs (int i) {
                    return ((i < 0) ? -i : i);
                }

                double abs (double d) {
                    return ((d < 0.0) ? -d : d);
                }

                ...
                int a = -5;
                double d = -2.23;
                auto x = abs(a);    // int abs(int)
                auto y = abs(b);    // double abs(double)
```

        - Example: NOT OK

```
                // NOT OK: same name, same parameter lists
                int foo (int i) {
                    return (2 * i);
                }

                double foo (int i) {
                    return (2.5 * i)
                }

                // Does not compile!
```

Implementation

- Recursion
    - = function calling itself
    - Needs a break condition
    - Looks more elegant than loops
        - But in many cases slower
    - Recursion depth is limited (by stack size)
    - Example

```
                int factorial (int n) {
                    // break condition:
                    if (n < 2) return 1;
                    // recursive call: n! = n * (n-1)!
                    return (n * factorial(n - 1));
```

```
            }

    - Declaration vs. Definition
        - Can only call functions that are already known (from before/above)
        - Only one definition allowed per source file ("translation unit")
        - OK to have any number of declarations
            - "Announcing the existence of a funciton by specifying its
               signature"
        - Example: BROKEN

            // COMPILER ERROR: - 'odd'/'even' not known before 'main'!
            // COMPILER ERROR: - 'odd' not known before 'even'!

            int main () {
                int i = 0;
                cin >> i;
                if (odd(i))  cout << "is odd\n";
                if (even(i)) cout << "is even\n";
            }

            bool even (int n) {
                return !odd(n);
            }

            bool odd (int n) {
                return (n % 2);
            }

    Design

    - Contracts
        - When designing a funcion, think about:

            (1) Preconditions
                    - What do you expect/demand from input values?
            (2) Postconditions
                    - What guarantees should you give regarding output values?
            (3) Invariants
                    - What do callers/users of the function expect to not change
            (4) Purpose
                    - Has the function a clearly defined purpose?
            (5) Name
                    - Does the function's name reflect its purpose?
            (6) Parameters
                    - Can a caller/user easily confuse their meaning?

        - Precondition Checks

            (1) Wide Contract Functions
                    - Perform precondition checks
                        - i.e., check input parameter values (or program state)
                          for validity
            (2) Narrow Contract Functions
                    - Do not perform precondition checks
                        - i.e., the caller has to make sure that input arguments
                          (and program state) are valid

    - Attribute [[nodiscard]]   C++17
        - Encourages compilers to issue warnings if function return values are
          discarded
        - Example

            [[nodiscard]] bool prime (int i) { ... }

            // return value(s) used:
            bool const yes = prime(47);
            if (prime(47)) { ... }
```

```
            // return value discarded/ignored:
            prime(47);  // COMPILER WARNING

      - Example from the standard library
          - std::vector's empty() function is declared with [[nodiscard]] as
            of C++20
              - Can be confused with clear()

                  std::vector<int> v;
                  // ...
                  if (v.empty()) { ... } // OK

                  v.empty();  // C++20: COMPILER WARNING
                  // oops ... did someone meant to clear it?

      - Declare your function return values [[nodiscard]]
          - If calling it without using the return value makes no sense in any
            situation
          - If users could be confused about its purpose, if the return value
            is ignored

  - No-Throw Guarantee: noexcept  C++11
      - The noexcept keyword
          - Specifies that a function promises to never throw exceptions / let
            exceptions escape:

                  void foo () noexcept { ... }

          - If an exception escapes from a noexcept function anyway:
              - The program will be aborted

  Some Mathematical Functions

          #include <cmath>

          double sqrt (double x)              ..square root
          double pow (double a, double b)     ..power                  a^b
          double abs (double x)               ..absolute value
          double sin (double x)               ..sine
          double cos (double x)               ..cosine
          double exp (double x)               ..exponential            e^x
          double log (double x)               ..logarithm              log(x)
          double floor (double x)             ..next smaller integer   |x|
          double ceil (double x)              ..next larger integer    |x|
          double fmod (double x, double y)    ..remainder of x/y


  ----------------------------------------------------------------------------
  Memory (Basics)
  ----------------------------------------------------------------------------

  Memory Model

  - On Paper: C++'s Abstract Memory Model
      - Memory Organisation
          (1) Memory is divided into bytes (usually 1 byte = 8 bits)
          (2) Each byte has an address

          <- ...| 00100100 | 00000000 | 00000100 | ...
                      0x15       0x14       0x13

      - Object = piece of memory
      - Example

              std::int16_t i = 1234;

                  - An object with:
                      - name i
```

```
                      - size of 2 bytes (= 16 bits)
                      - value 0000010011010010
                          - According to its type int16_t represents 0d1234

          - Note:
              - Abstract model does not say anything about
                  (a) how memory is partitioned
                  (b) cache hierarchies

          - Object Storage Duration Types

              (1) Automatic
                      - Object lifetime tied to start and end of { ... } block
                        scopes
                      - Local variables, function parameters
              (2) Dynamic
                      - Object lifetime controlled with special instructions
                      - Objects that can be created/destroyed on demand and
                        independent of block scopes
              (3) Thread
                      - Object lifetime tied to start and end of a thread
                      - Per-thread storage
              (4) Static
                      - Object lifetime tied to start and end of the program
                      - Singletons, ...

      - In Practice: Actual Memory Handling
          - Practical realizations of C++'s memory model

              (1) Are constrained by features and limitations of the target
                  platform (CPU/memory architecture, operating system, compiler)
              (2) Need to fix choices left open by the C++ standard
                      - e.g., number of bits in a byte (8 on most platforms)
              (3) Need to support object storage duration/lifetime schemes
                  described by the C++ standard
                      - Automatic, dynamic, thread, static

          - Common Solution: Dedicated Memory Paritions For Automatic/Dynamic
            Storage Duration

              (1) HEAP (also called "Free Store")
                      - Used for objects of dynamic storage duration
                          - e.g., contents of a std::vector
                      - Big
                          - Can be used for bulk storage (most of main memory)
                      - Possible to allocate and deallocate any object on demand
                      - (De-)allocations in no particular order
                          - Fragmentation
                      - Slow allocation
                          - Need to find contiguous unoccupied space for new obj
              (2) STACK
                      - Used for objects of automatic storage duration
                          - e.g., local variables, function parameters
                      - Small
                          - Usually only a few MB
                      - Fast allocation
                          - New objects are always put on top
                      - Objects de-allocated in reverse order of their creation
                      - Can't de-allocate objects below the topmost (= newest)

      Automatic Storage

      - Example: Local Variables

                  int main () {                              // 01
                      int i = 4;                             // 02
                      for (int j = 0; j < i; j++) {          // 03
                          if (j > 2) {                       // 04
```

```
                  int k = 2 * j;              // 05
              }
                                              // 06
          }
                                              // 07
      }                                       // 08

        (01) Program starts            (02) Local variable i is pushed
                                            on the stack

                  ....                           ....
      STACK   |            |                 |            |
              +------------+                 +------------+
              |            |                 |            |
              +------------+                 +------------+
              |            |           int i |          4 | <-
              +------------+                 +------------+
              |            | <-              |            |
              +------------+                 +------------+
              |            |                 |            |
                  ....                           ....

        (03) Loop-local variable j     (04.0) Second loop iteration:
                is pushed on the              j is incremented to 1
                stack.
             First loop iteration

                  ....                           ....
              |            |                 |            |
              +------------+                 +------------+
              |            |                 |            |
              +------------+                 +------------+
        int j |          0 | <-        int j |          1 | <-
              +------------+                 +------------+
        int i |          4 |           int i |          4 |
              +------------+                 +------------+
              |            |                 |            |
                  ....                           ....

        (04.1) Third loop iteration:   (04.2) Third loop iteration:
                j is incremented to 2         j is incremented to 3


                  ....                               ....
              |            |                     |            |
              +------------+                     +------------+
              |            |                     |            |
              +------------+                     +------------+
        int j |          2 | <-            int j |          3 | <-
              +------------+                     +------------+
        int i |          4 |               int i |          4 |
              +------------+                     +------------+
              |            |                     |            |
                  ....                               ....

        (05) Condition j > 2 is true       (06) Leaving if-branch scope:
                -> entering if-branch                k is popped from the
             Branch-local var k is                   stack
                pushed on the stack               Note: k's value still
                                                     remains in memory -
                                                     only the stack's top
                                                     position markder is
                                                     decreased by one


                  ....                               ....
              |            |                     |            |
              +------------+                     +------------+
```

```
            int k |          6 | <-              |          6 |
                  +------------+                 +------------+
            int j |          3 |           int j |          3 | <-
                  +------------+                 +------------+
            int i |          4 |           int i |          4 |
                  +------------+                 +------------+
                  |            |                 |            |
                      ....                           ....

        (07) Leaving the for loop's scope:  (08) Leaving the scope of the
                j is popped from the stack       main function:
             Note: k's value still                  i is popped from the
                remains in memory -                 stack
                only the stack's top             Stack top position marker
                position markder is              is decreased by one
                decreased by one

                      ....                           ....
                  |            |                 |            |
                  +------------+                 +------------+
                  |          6 |                 |          6 |
                  +------------+                 +------------+
                  |          3 |                 |          3 |
                  +------------+                 +------------+
            int i |          4 | <-              |          4 |
                  +------------+                 +------------+
                  |            |                 |            | <-
                  +------------+                 +------------+
                  |            |                 |            |
                      ....                           ....
```

Dynamic Storage: std::vector

- Dynamic Memory Allocation
    - Now: by only using std::vector
    - Next: more standard library containers (set, map, ...)
    - Later: manual dynamic memory allocation
    - In modern C++
        - Manual allocation is actually only really necessary if an own
          dynamic data structure/container is implemented

- std::vector Memory Layout
    - Each vecotr object holds a separate buffer that is dynamically
      allocated (on the heap) where the actual content is stored

```
          vector<int> v {0,1,2,3,4}

          contiguous
          buffer on   0 1 2 3 4
             HEAP     ^
          ------------|---------------------
          vector      |
          object on   v
            STACK
```

    - A vector object v itself could also be allocated on the heap

- std::vector Growth Scheme
    - Memory blocks, once allocate, can't be resized!
        - No guarantee that there is space left directly behind previously
          allocated memory block
    - Dynamic array implementations separate the array object from the
      actual memory block for stroing values
    - Growth is then done the following way:

        (1) Dynamically allocate new, (~1.1-2x) larger memory block
        (2) Copy/move old values to new block
        (3) Destroy old, smaller block

```
    - std::vector Size vs. Capacity

            .size()                         ..number of elements in vector
            .resize(new_num_of_elements)
            .capacity()                     ..num of available mem slots
            .reserve(new_capacity)

        - Example

                                                capacity    size
        vector<int> v;                                 0       0
        v.push_back(7);         7                      1       1
        v.reserve(4);           7 _ _ _                4       1
        v.push_back(8);         7 8 _ _                4       2
        v.push_back(7);         7 8 9 _                4       3

        auto s = v.size();      s: 3
        auto c = v.capacity();  c: 4

        v.resize(6,0)           7 8 9 0 0 0            6       6

        - If knowing the (approximate) number of elements in advance:
            - .reserve() before adding elements to the vector!

    - std::vector Memory Lifetime Example

        void foo (                          // 05
            vector<int> w,                  // 06
            int x)                          // 07
        {
            w[0] = x;                       // 08
        }                                   // 09

        int main () {                       // 01
            vector<int> v;                  // 02
            v.push_back(3);                 // 03
            v.push_back(6);                 // 04
            foo(v, 8);                      // 10
        }                                   // 11

        - Procedure:
            01) Program starts
            02) v is put on the stack
            03) vector v allocates buffer on the heap and puts 3 into it
            04) vector v allocates new buffer in order to make room for the
                next element 6
                    04.1) Old elements of v are copied to new buffer and new
                          element 6 is added
                    04.2) v deallocated its old buffer (i.e. from heap)
            05) program execution jumps to function foo
            06) entering foo: local parameter w is put on the stack; w allocates
                its buffer (on heap)
                    06.1) w copies elements from v; w is now a local copy of
                          argument v
            07) local parameter x is put on the stack
            08) the first element of w is changed (in the heap)
            09) foo ends: its local parameters are removed from the stack,
                beginning with x
                    09.1) when w is destroyed, it deallocates its heap buffer
            10) returned from foo; note that the change to local parameter w had
                no effect outside of foo
            11) program ends: when v is destroyed, it deallocates its heap buffr


    ----------------------------------------------------------------------------
    Strings (Basics)
    ----------------------------------------------------------------------------
```

- std::string
  - Dynamic array of char (similar to vector<char>)
  - Concatenation with + or +=
  - Single character access with [index]
  - Modifiable ("mutable") unlike in e.g., Python or Java
  - regular: deeply copyable, deeply comparable
  - Example

```
#include <string>

std::string hw = "Hello";
std::string s = hw;         // copy of hw
hw += " World!";

cout << hw << '\n'          // Hello World!
     << hw[4] << '\n'       // o
     << s << '\n';          // Hello
```

- char = std::string's Element Type
  - One char can hold a single character
  - Smallest integer type (usually 1 byte)
  - char literals must be enclosed in single quotes: 'a', 'b', ...
  - Example

```
char c1 = 'A';
char c2 = 65;                   // ASCII code of 'A'

cout << c1 << '\n'              // A
     << c2 << '\n'              // A
     << (c1 == c2) << '\n';     // 1

std::string s = "xyz";
s[1] = c1;
cout << s << '\n';              // xAz
s += c2;
cout << s << '\n';              // xAzA
```

  - Special Characters

```
\n      new line
\t      tab
\'      single quote
\"      double quote
\\      backslash itself
```

- std::string Manipulation
  - Example

```
string s = "I am sorry, Dave.";

// Changes to string object
s.insert(5, "very")         I am very sorry, Dave.
s.erase(6. 2)               I am sry, Dave.
s.replace(12,5,"Frank")     I am sorry, Frank
s.resize(4)                 I am
s.resize(20, '?')           I am sorry, Dave.???

// No changes to string object
s.find("r")                 7 (first occurrence from begin)
s.rfind("r")                8 (first occurrence from end)
s.find("X")A                string::npos (not found)
s.find('a',5)               13 (first occ. starting at 5)
s.substr(5,6)               "sorry," (returns new str object)
s.contains("sorry")         true (C++23)
s.ends_with("ave.")         true (C++20)
s.starts_with('I')          true (C++20)
```

```
    - Literals

            'a'     // char Literal

        - "C string Literal"
            - Example

                auto a = "seven of";    // type of a is char const[]
                auto b = a;             // b refers to same object as a

                a+= " nine";            // COMPILER ERROR: cannot be modified
                auto c = "al" + "cove"; // COMPILER ERROR

                std::string s = a;      // a is copied into s
                s += " nine";           // OK: s is std::string

        - "std::string Literal"s    C++14
            - Example

                #include <string>
                using namespace std::string_literals;

                auto s1 = "seven of"s;  // type of s1 is std::string
                auto s2 = s1;           // s2 is a copy of s1

                s1 += " nine";          // OK
                cout << s1 << '\n'      // seven of nine
                     << s2 << '\n';     // seven of

                auto s3 = "uni"s + "matrix"s;   // OK
                cout << s3 << '\n';             // unimatrix

        - Joining
            - String literals that are only separated by whitespace are joined:

                "first" "second" -> "first second"

            - Example

                std::string s =
                    "This is one literal"
                    "split into several"
                    "source code lines!";

        - Raw String Literals
            - Advantage: special characters can be used without escaping

                R"(raw "C"-string c:\users\joe)"        char const[]    C++11
                R"(raw "std"-string c:\users\moe)"s     std::string     C++14???

            - Syntax:

                R"DELIMITER( characters.. )DELIMITER"

            - DELIMITER
                - Can be a sequence of 0 to 16 characters except spaces, (, )
                  and \

    - String-Like Function Parameters
        - Use the following for read-only parameters!   C++17

                    std::string_view

            - Lightweight
                - Cheap to copy
                - Can be passed by value
            - Non-owning
                - Not responsible for allocating or deleting memory
```

```
          - Read-only view
              - Does not allow modification of target string
          - Of a string(-like) object
              - std::string/"literal"/...
          - Primary use case: read-only function parameters
          - Avoids expensive temporary strings when string literals are passed
            to functions
          - Can speed up accesses by avoiding a level of indirection:

                   string const&     -->    string      --> s o m e  t e x t
                reference to string      string object  ^   dynmic mem block
                                                        |
                    string_view      -------------------+
                string_view object

      - Example

              #include <string>
              #include <string_view>

              int edit_distance (std::string_view s1, std::string_view s2) {
                  ...
              }

              std::string input "abx";
              int dist = edit_distance("abc", input);

      - Usage of parameter types

                If ...                              Use Parameter Type
              +---------------------------------------------------------------+
              | always needing a copy               std::string               |
              |  of the input string                "pass by value"           |
              +---------------------------------------------------------------+
              | wanting ro access                   #include <string_view>    |
              |   - don't (always) need copy        std::string_view          |
              |   - using C++17/20                                            |
              +---------------------------------------------------------------+
              | wanting ro access                   std::string const&        |
              |   - don't (always) need copy        "pass by const reference" |
              |   - using C++98/11/14                                         |
              +---------------------------------------------------------------+
              | wanting the function to modify      std::string &             |
              | the input string in-place           "pass by (non-const) ref" |
              |   - Try to avoid such output                                  |
              |     parameters                                                |
              +---------------------------------------------------------------+

  - std::getline

      - Read entire lines/chunk of text at once
      - Target string can be re-used (saving memory)
      - Example

              std::string s;
              getline(std::cin, s);           // Read entire line
              getline(std::cin, s, '\t');     // Read until next tab
              getline(std::cin, s, 'a');      // Read until next 'a'


  ----------------------------------------------------------------------------
  References
  ----------------------------------------------------------------------------

  Capabilities (& Limitations)

  - non-const References
      - Example
```

```
            int   i = 2;
            int& ri = i;    // reference to i

            // ri and i refer to the same object/memory location
            cout <<  i << '\n';     // 2
            cout << ri << '\n';     // 2

            i = 5;
            cout <<  i << '\n';     // 5
            cout << ri << '\n';     // 5

            ri = 88;
            cout <<  i << '\n';     // 88
            cout << ri << '\n';     // 88
```

- References cannot be "null"
  - i.e., they must always refer to an object
- A reference must always refer to the same memory location
- Reference type must agree with the type of the referenced object
- Example

```
            int   i = 2;
            int   k = 3;
            int& ri = i;    // Reference to i

            ri = k;         // assigns value of k to i (target of ri)

            int& r2;        // COMPILER ERROR: reference must be initialized
            double& r3= i;  // COMPILER ERROR: types must agree
```

- const References
  - Means: read-only access to an object
  - Example

```
            int i = 2;
            int const& cri = i;     // const reference to i

            // cri and i refer to the same object / memory location
            // const means that value of i cannot be changed through cri

            cout << i   << '\n';    // 2
            cout << cri << '\n';    // 2

            i = 5;
            cout << i   << '\n';    // 5
            cout << cri << '\n';    // 5

            cri = 88;               // COMPILER ERROR: const!
```

- auto References
  - Reference type is deduced from right hand side of assignment
  - Example

```
            int i = 2;
            double d = 2.023;
            double x = i + d;

            auto & ri = i;          // ri: int &
            auto const& crx = x;    // crx: double const&
```

Usage

- References in Range-Based for Loops
  - Example

```
            std::vector<std::string> v;
            v.resize(10);
```

```
            // modify vector elements:
            for (std::string & s : v) { cin >> s; }
            // read-only access to vector elements:
            for (std::string const& s : v) { cout << s; }

            // modify:
            for (auto & s : v) { cin >> s; }
            // read-only access:
            for (auto const& s : v) { cout << s; }
```

- const Reference Parameters
    - Read-Only Access -> const&
        - Avoids expensive copies
        - Clearly communicates read-only intent to users of function
    - Example: Function that computes median

```
            // Only needs to read values from vector!

            // pass by value -> copy    (no good)
            int median (vector<int>);

            auto v = get_samples("huge.dat");
            auto m = median(v);
            //runtime & memory overhead!

            // pass by const& -> no copy    (good)
            int median (vector<int> const&);

            auto v = get_samples("huge.dat");
            auto m = median(v);
            // no copy -> no overhead!
```

    - Example: Mixed passing (by ref + by value)

```
            incl_first_last ({1,2,4},{6,7,8,9}) -> {1,2,4,6,9}
```

        - The implementation works on a local copy 'x' of the first vector
          and only reads from the second vector via const reference 'y':

```
            auto incl_first_last (std::vector<int> x,
                                  std::vector<int> const& y) {
                if (y.empty() return x;
                // append to local copy 'x'
                x.push_back(y.front());
                x.push_back(y.back());
                return x;
            }

            int main () {
                std::vector<int> v1 = {1,2,4};
                std::vector<int> v2 = {6,7,8,9};
                std::vector<int> result = incl_first_last(v1,v2);

                // Print (read only) values of the vector result:
                for (auto const& v : result) {std::cout << v;}
            }
```

- non-const Reference Parameters
    - Example: Function that exchanges values of two variables

```
            void swap (int& i, int& j) {
                int temp = i;              // copy i's value to temp
                i = j;                     // copy j's value to i
                j = temp;                  // copy temp's value to j
            }

            int main () {
```

```
                    int a = 5;
                    int b = 3;
                    swap(a,b);
                    cout << a << '\n'       // 3
                         << b << '\n';      // 5
                }

        - TIP:
            - Use the following to exchange values of objects

                #include <utility>

                std::swap

        - TIP:
            - Avoid non-const references: because "output parameter"

    - Function Parameters: copy / const& / & ?

            void read_from (int);                       // fundamental types
            void read_from (std::vector<int> const&);
            void copy_sink (std::vector<int>);
            void write_to  (std::vector<int> &);

        - Read from cheaply copyable object (all fundamental types)
            -> pass by value

                    double sqrt (double x) { ... }

        - Read from object with larger (> 64bit) memory footprint
            -> pass by const&

                    void print (std::vector<std::string> const& v) {
                        for (auto const& s : v) { cout << s << ' '; }
                    }

        - Copy needed inside function anyway
            -> pass by value
                - Pass by value instead of copying explicitly inside the functn
                    - Reason: check more advanced articles

                    auto without_umlauts (std::string s) {
                        s.replace('Ã¶', "oe");              // modify local copy
                        ...
                        return s;                           // return by value!
                    }

        - Write to function-external object
            -> pass by non-const&
                - Avoid such "output parameters" in general

                    void swap (int& x, int& y) { ... }

    - Avoid Output Parameters!
        - Functions with non-const ref parameters like

                    void foo (int, std::vector<int>&, double);

          can create confusion/ambiguity at the call site:

                    foo(i, v, j);

            - Which of the argumets (i, v, j) is changed and which remains
              unchanged?
            - How and when is the referenced object changed and is it changed
              at all?
            - Does the reference parameter only act as output (function only
              writes to it) or also as input (function also reads from it)?
```

```
            -> In general hard to debug and to reason about!

      - Example: An interface that creates nothing but confusion

            void bad_minimum (int x, int& y) {
                if (x < y) y = x;
            }

            int a = 2;
            int b = 3;
            bad_minimum(a,b);
            // Which variable holds the smaller value again?

    Binding Rules

    - Rvalues vs. Lvalues
        - Lvalues
            - Expressions of which we can get memory address
            - Refer to objects that persist in memory
            - Everything that has a name
                - e.g., variables, function parameters, ...
        - Rvalues
            - Expressions of which we can't get memory address
            - Literals
                - e.g., 123, "string literal", ...
            - Temporary results of operations
            - Temporary objects returned from functions
        - Example

            int a = 1;          // a and b are both lvalues
            int b = 2;          // 1 and 2 are both rvalues
            a = b;
            b = a;

            a = a * b;          // (a * b) is an rvalue
            int c = a * b;      // c is a lvalue

            a * b = 3;          // COMPILER ERROR: cannot assign to rvalue

            std::vector<int> read_samples(int n) { ... }
            auto v = read_samples(1000);        // v is a lvalue
                                                // read_samples(1000) rvalue

    - Reference Binding Rules

            &               only binds to Lvalues
            const&          binds to const Lvalues and Rvalues

        - Example

            // const& here is an Rvalue
            bool is_palindrome (std::string const& s) {...}

            std::string s = "uhu";
            cout << is_palindrome(s) << ", "
                 << is_palindrome("otto") << '\n';      // OK, const&

        - Example

            // & binding here to Lvalues
            void swap (int& i, int& j) { ... }

            int i = 0;
            swap(i, 5);     // COMPILER ERROR: can't bind ref. to literal

    Pitfalls
```

```
    - Never Return A Reference To A Function-Local ...
        - Example

                // & binds here to Rvalue?
                int& increase (int x, int delta) {
                    x += delta;
                    return x;
                }                           // local x destroyed

                int main() {
                    int i = 2;
                    int j = increase(i,4);  // accesses invalid reference!
                }

        - Only valid if referenced object outlives the function!

                // Here the &'s bind to Lvalues
                int& increase (int&, int delta) {
                    x += delta;
                    return x;               // references non-local int
                }                           // OK, reference still valid

                int main() {
                    int i = 2;
                    int j = increase(i,4);  // OK, i and j are 6 now
                }

    - Careful With Referencing vector Elements!
        - References to elements of a std::vector might be invalidated after any
          operation that changes the number of elements in the vector!

                vector<int> v {0,1,2,3};
                int& i = v[2];
                v.resize(20);
                i = 5;       // UNDEFINED BEHAVIOR: original memory might be gone

        - Dangling Reference
            - Reference that refers to a memory location that is no longer valid

    - Avoid Lifetime Extension!
        - References can extend the lifetime of temporaries (rvalues)

                auto cosnt& r = vector<int>{1,2,3,4};

                -> vector exists as long as reference r exists

        - What about an object returned from a function?

                    std::vector<std::string> foo () { ... }

            - Take it by value (recommended):

                    vector<string> v1 = foo();
                    auto v2 = foo();

            - Ignore it (worse):
                -> gets destroyed right away

                    foo();

            - Get const reference to it (worse):
                -> lifetime of temporary is extended for as long as the
                   reference lives

                    vector<string> const& v3 = foo();
                    auto const& v4 = foo();

            - Don't take a reference to its members! (worst idea)
```

```
            - No lifetime extension for members of returned objects
                - Here: the vetor's content

                    string const& s = foo()[0]; // dangling reference!
                    cout << s;                  // UNDEFINED BEHAVIOR

    - TIP
        - DON'T use lifetime extension through references!
            - Easy to create confusion
            - Easy to write bugs
            - No real benefit
        - JUST TAKE RETURNED OBJECTS BY VALUE
            - Does not involve expensive copies for most functions and types
              in modern C++


----------------------------------------------------------------------------
Aggregate Types
----------------------------------------------------------------------------

- Type Categories (simplified)

    - Fundamental Types
        - void, bool, char, int, double, ...

    - Simple Aggregates
        - Main Purpose: grouping data
            - aggregate: may contain one/many fundamental or other
              aggregate-compatible types
            - no control over interplay of constituent types
            - "trivial" if only (compiler generated) default construction /
              destruction / copy / assignment
            - "standard memory layout" (all members laid out contiguous in
              declaration order), if all members have same access control
              (e.g., all public)

    - More Complex Custom Types
        - Main Purpose: enabling correctness/safety guarantees
            - cutsom invariants and control over interplay of members
            - restricted member access
            - member functions
            - user-defined construction / member initialization
            - user-defined destruction / copy / assignment
            - may be polymorphic (contain virtual member functions)

- How T Define / Use
    - Example: Type with 2 integer coordinates

            struct point {
                int x;      // <- "member variable"
                int y;
            };

            // Create new object (on stack)
            point p {44, 55};

            // print members' values
            cout << p.x << ' ' << p.y;  // 44 55

            // Assigning to member values:
            p.x = 10;
            p.y = 20;
            cout << p.x << ' ' << p.y;  // 10 20

        - Member variables are stored in the same order as they are declared

                 ...                              ...
                +-------+                        +-------+
```

```
                 | -7682 | STACK                     | -7682 | STACK
                 +-------+                           +-------+
                 | 23988 |                           | 23988 |
                 +-------+                           +-------+
             p.y |    55 | <- top                p.y |    20 | <- top
                 +-------+                           +-------+
             p.x |    44 |                       p.x |    10 |
                 +-------+                           +-------+
                    ...                                 ...
```

- Why Custom Types / Data Aggregation?

    - Interfaces become easier to use correctly
        - semantic data grouping: point, date, ...
        - avoids many function parameters and this, confusion
        - can return multiple values from function with one dedicated type instead of multiple non-const reference "output parameters"

    - Without: Horrible Interfaces!

```
void closest_point_on_line (double lx2, double ly1, double lx2i,
double ly2, double px, double py, double& cpx, double& cpy) {
    ...
}
```

        - Many parameters of same type
            -> easy to write bugs
        - Non-const reference output parameters
            -> error-prone
        - Internal representation of a line is also baked into the interface

    - With: A LOT Better!

```
struct point { double x; double y; };
struct line  { point a; point b; };

point closest_point_on_line (line const& l, point const& p) {
    ...
}
```

        - Straigh-forward interface
        - Easy to use correctly
        - If internal representation of line changes (e.g., point + direction instead of 2 points)
            -> Implementation of "closest_point_on_line" needs to be
               changed too, but its 'interface' can stay the same
                -> most of calling code doesn't need to change!

- Aggregate Initialization

```
Type { arg1, arg2, ..., argN }
```

    - Brace-enclosed list of member values
    - In order of member declaration
    - Example

```
enum class month {jan = 1; feb = 2; ... ; dec = 12};

struct date {
    int yyyy;
    month mm;
    int dd;
};

int main () {
    date today {2020; month::mar, 15};
    // C++98, but also still OK:
    date tomorrow = {2020, month::mar, 16};
```

```
                }

    - Compounds
       - Example: date as member of person

                enum class mont { jan=1, feb=2, ... , dec=12 };

                struct date {
                    int yyyy;
                    month mm;
                    int dd;
                };

                struct person {
                    std::string name;
                    date bday;
                };

                int main () {
                    person jlp { "Jean-Luc Picard", {2305, month::jul, 13} };
                    cout << jlp.name;       // Jean-Luc Picard
                    cout << jlp.bday.dd;    // 13

                    date yesterday { 2020, month::jun, 16 };
                    person rv = { "Ronald Villiers", yesterday };
                }

    - Copying
       - Copies are always deep copies of all members

                enum class month { jan=1, ... };

                struct date {
                    int yyyy;
                    month mm;
                    int dd;
                };

                int main () {
                    date a {2020, month::jan, 7};
                    date b = a;                         // deep copy of a
                    b.dd = 22;                          // change b
                }

          - State after last line of main:

                           ...
                        |        | STACK
                        +--------+
                   b.dd |     22 | <- top
                        +--------+
                   b.mm |      1 |
                        +--------+
                 b.yyyy |   2020 |
                        +--------+
                   a.dd |      7 |
                        +--------+
                   a.mm |      1 |
                        +--------+
                 a.yyyy |   2020 |
                        +--------+
                        |        |
                           ...

       - Copy Construction
          - Create new object with same values as source
       - Copy Assignment
          - Overwrite existing object's values with that of source
```

```
        - Example

                struct point { int x; int y; };

                point p1 {1, 2};    // construction

                point p2 = p1;      // copy construction
                point p3 ( p1 );    // copy construction
                point p4 { p1 };    // copy construction

                auto  p5 = p1;      // copy construction
                auto  p6 ( p1 );    // copy construction
                auto  p7 { p1 };    // copy construction

                p3 = p2;    // copy assignment
                            // (both p2 & p3 existed before)

    - Value vs. Reference Semantics

        - Value Semantics
            = variables refer to object themselves:

                - deep copying
                    - Produces a new, independent object
                    - Object (member) values are copied
                - deep assignment
                    - Makes value of target equal to that of source object
                - deep ownership
                    - Member variables refer to objects with same lifetime as
                      containing object
                - value-based comparison
                    - Variables compare equal if their values are equal

            - Value semantics is the default behavior for fundamental types in
              almost all programming languages
                - Also the default for aggregates/user-defined types in C++

        - Reference Semantics
            = variables are references to objects:

                - shallow copying
                    - Copies of a variable refer to the same object
                - shallow assignment
                    - Assignment makes a variable refer to a different object
                - shallow ownership
                    - Member variables are also just references
                - identity-based comparison
                    - Variables compare equal if they refer to the same object

            - Most other mainstream languages (Java, Python, C#, Swift, ...) use
              (baked-in) reference semantics for user-defined types

        - Situation in C++
            - Default: value semantics for ALL types (except C-style arrays)
            - Optional reference semantics possible for ALL types
                - By using references or pointers

    - std::vector of Aggregates
        - Value Semantics ->
            - vector<T>'s storage contains objects of type T themselves, not
              just "references" or "pointers" to them (as in Java/C#/...)
            - if vector object gets destroyed
                -> contained T objects get destroyed
        - Example

                vector<int> v { 0,1,2,3,4 };

                    contiguous
```

```
    buffer on   0 1 2 3 4
      HEAP      ^
    ------------|--------
    vector      |
    object on   v
     STACK

struct p2d { int x; int y; };
vector<p2d> v {{1,2},{5,6},{8,9}};

    contiguous
    buffer on   1 2 5 6 8 9
      HEAP      ^
    ------------|--------
    vector      |
    object on   v
     STACK
```

- The "Most Vexing Parse"
    - Can’t use empty parentheses for object construction due to an ambiguity in C++’s grammar:

```
struct A { ... };

A a ();      // declares function ’a’ without parameters and
             // return type ’A’
A a;         // constructs an object of type A
A a {};      // constructs an object of type A
```

Function Call Mechanics

- How Function Calls Work
    - Example

```
int square (int p) {        // 04
    int x;                  // 05
    x = p * p;              // 06
    return x;               // 07
}                           // 08
int main() {                // 01
    int y = 2;              // 02
    int i = square(y);      // 03, 09
    int k = i + 1;          // 10
}                           // 11
```

- Procedure
    - The exact order in which things are put on the stack during a function call (the "calling convention") depends on the platform (CPU architecture + OS + Compiler)

```
(01) The program starts          (02) Local var y is put on
                                      stack


       ...                              ...
    |        | STACK                 |        | STACK
    +--------+                       +--------+
    |        |                       |        |
    +--------+                       +--------+
    |        |                       |        |
    +--------+                       +--------+
->  |        |                   ->  |  y: 2  |
    +--------+                       +--------+
    |        |                       |        |
       ...                              ...
```

```
                    (03)  Local var i is put on      (03.1) Placeholder for the
                          stack                              return value of the func
                                                             is put on the stack


                             ...                                  ...
                          |        | STACK                 |              | STACK
                          +--------+                       +--------------+
                          |        |                    -> | placeholder  |
                          +--------+                       +--------------+
                    ->    |  i: ?  |                       |        i: ?  |
                          +--------+                       +--------------+
                          |  y: 2  |                       |        y: 2  |
                          +--------+                       +--------------+
                          |        |                       |              |
                             ...                                  ...


                    (03.2) Current instruction's     (03.2) Frame Pointer marks the
                           mem addr is put on                beginning of the stack
                           stack (to know where              frame of the current
                           to resume after leaving           function; everything in
                           the function)                     current stack frame is
                                                             treated as function local
                                                             (Pointer needed because
                                                              different func calls can
                                                              have different stack
                                                              frame sizes)


                                ...                                   ...
                          |              | STACK                 |              | STACK
                          |              |                       +--------------+
                          +--------------+                  fp-> |              |
                    ->    | return addr  |                    -> | return addr  |
                          +--------------+                       +--------------+
                          | placeholder  |                       | placeholder  |
                          +--------------+                       +--------------+
                          |        i: ?  |                       |        i: ?  |
                          +--------------+                       +--------------+
                          |        y: 2  |                       |        y: 2  |
                          +--------------+                       +--------------+
                          |              |                       |              |
                                ...                                   ...


                    (04)  Execution jumps to the mem (04.1) Function param p is put
                          addr of the func square            on the stack (value is
                                                             determined by call arg)


                                                                      ...
                                ...                              |              | STACK
                          |              | STACK                 +--------------+
                          +--------------+                  ->   |        p: 2  |
                  fp->    |              |                       +--------------+
                    ->    | return addr  |                  fp-> | return addr  |
                          +--------------+                       +--------------+
                          | placeholder  |                       | placeholder  |
                          +--------------+                       +--------------+
                          |        i: ?  |                       |        i: ?  |
                          +--------------+                       +--------------+
                          |        y: 2  |                       |        y: 2  |
                          +--------------+                       +--------------+
                          |              |                       |              |
                                ...                                   ...
```

```
                (05) Function-local var x        (06) The result of expression
                     is put on the stack              p * p is assigned to x


                          ...                                 ...
                    |             | STACK               |             | STACK
                    +-------------+                     +-------------+
              ->    |        x: ? |               ->    |        x: 4 |
                    +-------------+                     +-------------+
                    |        p: 2 |                     |        p: 2 |
                    +-------------+                     +-------------+
              fp->  | return addr |               fp->  | return addr |
                    +-------------+                     +-------------+
                    | placeholder |                     | placeholder |
                    +-------------+                     +-------------+
                    |        i: ? |                     |        i: ? |
                    +-------------+                     +-------------+
                    |        y: 2 |                     |        y: 2 |
                    +-------------+                     +-------------+
                    |             |                     |             |
                          ...                                 ...

                (07) The statement return x;     (08) Leaving func square: the
                     copies the value of x            stack’s top position is
                     into the return value            decreased below the stack
                     placeholder                      frame (all func locals are
                                                      popped from the stack)


                          ...                                 ...
                    |             | STACK               |             | STACK
                    +-------------+                     +-------------+
              ->    |        x: 4 |                     |        x: 4 |
                    +-------------+                     +-------------+
                    |        p: 2 |                     |        p: 2 |
                    +-------------+                     +-------------+
              fp->  | return addr |                 ->  | return addr |
                    +-------------+                     +-------------+
                    |           4 |                     |           4 |
                    +-------------+                     +-------------+
                    |        i: ? |                     |        i: ? |
                    +-------------+                     +-------------+
                    |        y: 2 |                     |        y: 2 |
                    +-------------+                     +-------------+
                    |             |                     |             |
                          ...                                 ...

                (09) Execution returns to        (09.1) Assignment int = i ..
                     the call site by jumping           causes the return value
                     to the previously stored           to be copied into i
                     return addr


                          ...                                 ...
                    |             | STACK               |             | STACK
                    +-------------+                     +-------------+
                    |        x: 4 |                     |        x: 4 |
                    +-------------+                     +-------------+
                    |        p: 2 |                     |        p: 2 |
                    +-------------+                     +-------------+
                ->  | return addr |                     | return addr |
                    +-------------+                     +-------------+
                    |           4 |                 ->  |           4 |
                    +-------------+                     +-------------+
                    |        i: ? |                     |        i: 4 |
                    +-------------+                     +-------------+
```

```
                |         y: 2 |                      |         y: 2 |
                +--------------+                      +--------------+
                |              |                      |              |
                      ...                                   ...


            (09.2) Return value is               (10) Local variable k is put
                   popped from the                    on the stack
                   stack

                      ...                                   ...
                |              | STACK                |              | STACK
                +--------------+                      +--------------+
                |         x: 4 |                      |         x: 4 |
                +--------------+                      +--------------+
                |         p: 2 |                      |         p: 2 |
                +--------------+                      +--------------+
                | return addr  |                      | return addr  |
                +--------------+                      +--------------+
                |            4 |                 ->   |         k: 5 |
                +--------------+                      +--------------+
            ->  |         i: 4 |                      |         i: 4 |
                +--------------+                      +--------------+
                |         y: 2 |                      |         y: 2 |
                +--------------+                      +--------------+
                |              |                      |              |
                      ...                                   ...


            (11) The program ends; all associated
                 variables are popped from the stack


                      ...
                |              | STACK
                +--------------+
                |         x: 4 |
                +--------------+
                |         p: 2 |
                +--------------+
                | return addr  |
                +--------------+
                |         k: 5 |
                +--------------+
                |         i: 4 |
                +--------------+
                |         y: 2 |
                +--------------+
            ->  |              |
                      ...
```

- No References To Locals
    - What if we changed the return type to int&?

```
int& square (int p) {
    int x;
    x = p * p;
    return x;            // 01
}                        // 02
int main() {
    int y = 2;
    int& i = square(y);  // 03
    int k = i + 1;
}
```

(01) Stack content right before returning from square:
                                  - Function-local var x

```
                                      - Function parameter p
           ...                        - Address of next instruction
     |              | STACK             after function call
     +--------------+                 - Placeholder for return value
->   |         x: 4 |                   of square
     +--------------+                 - Local variables y and i of
     |         p: 2 |                   main
     +--------------+
fp-> | return addr  |
     +--------------+
     | placeholder  |
     +--------------+
     |         i: ? |
     +--------------+
     |         y: 2 |
     +--------------+
     |              |
           ...


  (01.1) The statement return x;      (02) Leaving func square:
         copies the ADDRESS of x           Stack's top position is
         into the return value             decreased to below the
         placeholder                       stack frame (func locals
                                           are popped from stack)
                                           Execution returns to the
                                           call site by jumping to
                                           the previously stored
                                           return value


           ...                                   ...
     |              | STACK              |              | STACK
     +--------------+                    +--------------+
->   |         x: 4 |                    |         x: 4 |
     +--------------+                    +--------------+
     |         p: 2 |                    |         p: 2 |
     +--------------+                    +--------------+
fp-> | return addr  |                 -> | return addr  |
     +--------------+                    +--------------+
     |   addr of x  |                    |   addr of x  |
     +--------------+                    +--------------+
     |         i: ? |                    |         i: ? |
     +--------------+                    +--------------+
     |         y: 2 |                    |         y: 2 |
     +--------------+                    +--------------+
     |              |                    |              |
           ...                                   ...


  (03) The assignment int& i = ... causes the return value (= memory
       address of an integer) to be copied into reference i&
                                      - The memory location of x is above
                                        the stack's current top position
           ...                        - Any subsequent stack allocation
     |              | STACK             causes it to be overwritten with
     +--------------+                   other values
     |         x: 4 |
     +--------------+                 -> UNDEFINED BEHAVIOR
     |         p: 2 |                     - The runtime behavior of such a
     +--------------+                       program is undefined/non-
     | return addr  |                       deterministic:
     +--------------+                          - Might work, might not
  -> |   addr of x  |
     +--------------+
   ! |  i: INVALID  |
     +--------------+
     |         y: 2 |
```

```
                +-------------+
                |             |
                      ...
```

- Common Compiler Optimizations
   - Modern C++ Compiler
      - Perform several optimizations
         - Especially at higher optimization levels -O2 and -O3
         - Make function calls much faster

   - Return Value Optimization (RVO)
      - Applies to statements like

```
return Type{};
return Type{argument, ...};
```

      - Example

```
Point foo (...) {
    ...
    return Point{...};
}

Point res = foo();
```

         - No extra placeholder for the return value is allocated
         - No copy takes place
         - The external object res is directly constructed at the call site

      - Mandatory optimization
         - i.e., guaranteed to be performed as of C++17

   - Named Return Value Optimization (NRVO)
      - Applies to statements like

```
return local_variable;
```

      - Example

```
Point foo (...) {
    Point loc;
    ...
    return loc;
}

Point res = foo();
```

         - No extra placeholder for the return value is allocated
         - No copy takes place
         - The local object loc and the external object res are treated as one and the same
            - Results in only one allocation at the call site

      - Not mandatory
         - But almost all modern compilers will perform it, if possible

   - Inlining
      - Calls to small/short functions are replaced with the code of the function
      - Example

```
int square (int x) {
    return x * x;
}                                           int y = 0;
int y = 0;                   --->           std::cin >> y;
std::cin >> y;                              std::cout << (y*y);
std::cout << square(y);
```

```
      - For this to happen, the compiler must:
              (1) "see" the function declaration
              (2) "see" the function’s full definition
          - This is not necessarily the case if different parts of a
            program are compiled separately
      - One source of C++’s performance advantages
```