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

--------------------------------------------------------------------------------
Function Objects
--------------------------------------------------------------------------------


    ----------------------------------------------------------------------------
    Function Objects
    ----------------------------------------------------------------------------

    - Objects whose type provides at least one member function
        - overload of operator()


                class Multiplier {
                    int m_;
                public:
                    // constructor:
                    explicit constexpr Multiplier (int m) noexcept : m_{m} {}
                    // "call operator":
                    constextpr int operator () (int x) const noexcept {
                        return m_ * x;
                    }
                };

        - can be used like a function

                Multiplier triple(3);
                int i = tripple(2);     // i: 6

        - can be stateful:

                class Accumulator {
                    int sum_ = 0;
                public:
                    void operator () (int x) noexcept { sum_ += x; }
                    int total () const noexcept { return sum_; }
                };

                Accumulator acc;
                acc(2);
                acc(3);
                int sum = acc.total();  // sum: 5

        - can be used to customize behavior

                // of, e.g., standard library algorithms:
                if ( std::any_of(begin(v), end(v), in_internval{-2,8}) ) ...
                //               custom function object ^


    Example: Interval Query

                class in_interval {
                    int a_;
                    int b_;
                public:
                    // constructor:
                    in_interval (int a, int b) noexcept: a_{a}, b_{b} {}

                    // "call operator":
                    [[nodiscard]] constexpr
                    bool operator () (int x) const noexcept {
                        return x >= a_ && x <= b_;
                    }
                };

                // make an object
                in_interval test {-10,5};

                // invoke its opeator()
                cout << test(1);        // true
                cout << test(5);        // true
                cout << test(-12);      // false
                cout << test(8);        // false


    - Finding Intervals

        find_if(@begin, @end, f (o)->bool)
        -> @1st_element for wich f is true
        -> @end if no such element found

              @begin|         ||@end
                |9|0|4|1|8|3|7|2|   o >= 6
                         ^ @1st_element


                auto i = find_if(begin(v)+2, begin(v)+7, in_interval{6,8});


    - Partitioning with Intervals

        partition(@begin, @end, f(o)->bool) -> @ftrue_end

                                                     @ftrue_end
            @begin|               ||@end                 v
                  |1|4|6|0|2|8|3|5| |   o <= 2    |1|2|0|6|4|8|3|5| |
                                                  |     |         |
                                                  +-----+---------+
                                                   true    false

            - NOTE:
                - The relative order of elements within the resulting partitions
                  need not to be the same as in the original sequence


                auto i = partition(begin(v), end(v), in_interval{-1,2});

                for_each(begin(v), i, [](int x){ std::cout << x << ' ';});
                for_each(i, end(v),   [](int x){ std::cout << x << ' ';});


    Guidelines


    - Avoid Stateful operator()

        - Stateful
            - The current result of operator() depends on previous calls of
              operator()
                - e.g., because member variable values are both used for
                  computing the result and changed in the same call to
                  operator()

        - CARE
            - Many (standard) algorithms do not guarantee any order in which
              passed-in function objects are "called"
                - This is especially the case for the parallel versions of the
                  standard algorithms that were introduced with C++17
```

- Passing stateful function objects
  - might yield different results depending on:
    (1) the concrete implementation of a particular algorithm and
    (2) on the state of the function object prior to passing it to the algorithm

- Better
  - Subsequent calls to operator() should be independent from each other
  - Prefer to make operator() const, i.e., not alter the function object’s state at all
  - If using a non-const operator() with a parallel standard algorithm
    - e.g., for tracking status information
    - Make sure it is concurrency-safe
      - Example
        - Access to resources that are shared between multiple threads, like e.g., I/O-streams has to be managed properly

Standrad Library Function Objects

- Comparisons

  #include <functional>

  ```
  - std::equal_to             - std::less
  - std::not_equal_to         - std::greater_equal
  - std::greater              - std::less_equal
  ```

  - C++11
    - Must specify operand type explicitly: std::greater<Type>{}

  - C++14
    - No need for specifying operand type: std::greater<>{}

  - Example

  ```
  // set with descending order (default is ’less’):
  std::set<int,std::greater<>> s;

  // compare with ’greater’ instead of the default ’less’:
  std::vector<int> v1 = {1,4,5};
  std::vector<int> v2 = {1,2,5};

  cout << lexicographical_compare(begin(v1), end(v1),
                                  begin(v2), end(v2),
                                  std::greater<>{});  // true
  ```

- Arithmetic Operations

  #include <functional>

  ```
  - std::plus             - std::divides
  - std::minus            - std::modulus
  - std::multiplies       - std::negate
  ```

  - C++11
    - Must specify operand type explicitly: std::minus<Type>{}

  - C++14
    - No need for specifying operand type: std::minus<>{}

  - Example: Left Fold Using Binary Operation

  ```
  accumulate(@begin, @end, w) (+) = o + o
  ```

```
        accumulate(@begin, @end, w, +(x,o)->O)
        -> w + o_0 + o_1 + .. + o_n

        - Uses operator + as default, if no fold operation is given as
          fourth argument
            -> result is sum of the input elements

            int sum = accumulate(begin(v), end(v), 0);  // sum

            int product = accumulate(begin(v), end(v), 1,
                                     std::multiplies<>{});  // product


----------------------------------------------------------------------------
Lambdas (Basics)
----------------------------------------------------------------------------


- Reminder: Function Classes and Objects

    - class provides at least one operator () (...) {...}
    - can be invoked like a function
    - can be stateful (unlike functions)

        struct in_interval {
            explicit in_interval(int min, int max): min_{min}, max_{max} {}

            bool operator () (int x) const noexcept {
                return x >= min_ && x <= max_;
            }
        private:
            int min_, max_;
        };

        in_interval inside {-10,10};
        if (inside(5)) cout << "inside\n"; else cout << "outside\n";


- Lambdas (C++11)

        - compiler-generated function objects
        - can be used like anonymous functions

    - Examples

            [] {return 200;}

            [] (int x, int y) {
                return (0.5 * (x + y))};    // with parameter list

            [] (int x, int y) -> double {
                return (0.5 * (x + y))};    // explicit return type


- partition

    partition(@first, @last, f(o)->bool)

            auto v = vector<int>{5,3,-3,2,7,1,0,99,3};
            auto i = partition(begin(v), end(v), in_interval{-1,4} );

            if (i != end(v)) cout << *i << '\n';    // 5

        - with a lambda (C++11)

            auto v = vector<int>{5,3,-3,2,7,1,0,99,3};
            auto i = partition(begin(v), end(v),
                               [](int x){ return x >= -1 && x <= 4; });
```

- Variable Capturing (C++11)

```
[=]     (...) {...}     ..captures all by value
[&]     (...) {...}     ..captures all by reference
[x, &y] (...) {...}     ..captures x by value, y by reference
[=, &y] (...) {...}     ..captures all except y by value
```

```
vector<int> v {1,2,3,4,5};

int i = 2;

transform(begin(v), end(v), begin (v),
    [&] (int x) {               // i captured by reference
        ++i; return (x * i);    // v = {3,8,15,24,35}
    });
cout << i << '\n';              // i = 7
```

- Storing Closures (C++11)

    - type names of closures only known to compiler
      -> use auto if you need to store closures

```
vector<int> v {1,2,3,4,5};

auto squ = [] (int x) { return (x * x); };

transform(begin(v), end(v), begin(v), squ);
```

- generate

```
generate(@first, @last, f()->o)

    |0|0|0|0|   ->    |2|4|6|8|
```

    - Example

```
struct even_ints {
    int operator() { i +=2; return i; }
private:
    int i = 0;
};

vector<int> v;
v.resize(9, 0);

generate(begin(v)+2, begin(v)+6, even_ints{} );

// DOES NOT WORK?
```

    - Example

```
vector<int> v;
v.resize(9,0);

int i = 0;

generate(begin(v)+2, begin(v)+6, [&]{ i += 2; return i; });
```

    - Example

```
                vector<int> v;
                v.resize(9,0);

                int i = 0;
                auto even_ints = [&]{ i += 2; return i; };

                generate(begin(v)+2, begin(v)+6, even_ints);
```

- Generic Lambdas (C++14)

```
                // value parameters
                [] (auto x, auto y) {return (x + y)/2; }

                // const reference parameters
                [] (auto const & x, const & auto y) {return (x + y)/2; }

                // non-const reference parameters
                [] (auto & x) { ++x; }

                // MIX
                [] (auto & x, auto y, auto const & z) {...}
```

- transform

```
            transform(@first, @last, @out, [](...){...})

                ...|a2|a3|a4|...    ->    ...|lmbd(a2)|lmdb(a3)|lmbd(a4)|...
```

  - Example

```
                vector<some_arithm_type> v {...};

                transform(begin(v), end(v), begin(v),
                    [](auto const& x) { return x*x; });
```

- Generalized Capture (C++14)

  - useful for:
    - adding new member variables to closures
    - moving objects into closures

  - Example

```
                auto myfn1 = [ i = 5 ] (int x) { return x + i; }

                class ExpensiveToCopyType { ... };

                ExpensiveToCopyType f {1,7,8};

                auto myfn2 = [ cf = std::move(f) ] (int x) { return cf(x); }
```

- YOUTUBE

  - Back To Basics: Lambdas

    https://www.youtube.com/watch?v=IgNUBw3vcO4

  - Lambdas In Action

    https://www.youtube.com/watch?v=UOu_1Foq4mk

  - Lambdas In C++

    https://www.youtube.com/watch?v=ZHw2XHij1is