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

--------------------------------------------------------------------------------
Custom Types - Part 1
--------------------------------------------------------------------------------

    ----------------------------------------------------------------------------
    Basic Custom Types / Classes
    ----------------------------------------------------------------------------

    - 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)


    Why Custom Types?

        - Correctness Guarantees
            - invariants
                - Behavior and/or data properties that never change
            - avoid data corruption by controlling/restricting access to data
              member
            - restrict input/output value of functions by using dedicated types

        - Reusable Abstractions
            - easy-to-use interfaces that hide low-level implementation details
            - stable interfaces that are not affected by changing internal
              implementations
            - reusable abstractions for commonly needed facilities (e.g.,
              dynamic array)

        - Resource Management
            - Also called "RAII" (Resource Acquisition Is Initialization)
                - acquire some resource (memory, file handle, connection, ...)
                  when object is constructed
                - release/clean up resource when object is destroyed (de-
                  allocate memory, close connection, ...)

    - Motivation: Monotonous Counter
        - Stores an integer
        - Is initialized with 0
        - Invariant:
            - Count can only increase
```

```
        - No decrease, no reset

                monotonous_counter c;
                cout << c.reading();    // prints 0
                c.increment();
                cout << c.reading();    // prints 1
                c.increment();
                c.increment();
                cout << c.reading();    // prints 3

      - A simple aggregate cannot guarantee this:

                struct frail_counter {
                    int count;
                }

                frail_counter c;
                cout << c.count; // any value
                c.count++;
                c.count = 11;

        - Integer member not automatically initialized to 0
        - One can freely modify any integer member of an aggregate
        - No advantage over just using an integer

  - Motivation: Ascending Sequence
      - Should store integers
      - Invariant
        - Number of elements can only increase
            - i.e., one can only insert new elements, but not remove them
      - Invariant
        - Elements must always be sorted in ascending order

                ascending_sequence s;
                s.insert(5)                 // 5
                s.insert(-8)                // -8 5
                s.insert(42)                // -8 5 42
                cout << s.size();           // prints 3
                cout << s[0];               // prints -8

      - A simple aggregate cannot guarantee this:

                struct chaotic_sequence {
                    std::vector<int> nums;
                };

                chaotic_sequence s;
                s.nums.push_back(8);        // 8
                s.nums.push_back(1);        // 8 1
                s.nums.push_back(4);        // 8 1 4
                s.nums.pop_back(4);         // 8 1

        - Numbers not necessarily sorted in ascending order
        - We can manipulate "num" without restriction
        - No advantage over just using a plain std::vector

  Restricted Member Access

  - Member Functions

                class monotonous_counter {
                    int count_;             // <- data member
                    ...
                    void increment () {     // <- member function
                        ++count_;
                    }
                };
```

```
                class ascending_sequence {
                    std::vector<int> seq_,  // <- data member
                    ...
                    void insert (int x) {   // <- member function
                        // insert "x" into nums at the right position
                    }
                };

        - Member functions can be used to
            - manipulate or query data member
            - control/restrict access to data member
            - hide low-level implementation details
            - ensure correctness:
                - keep/guarantee invariants
            - ensure clarity:
                - well-structured interfaces for users of type
            - ensure stability:
                - internal data representation (mostly) independent from
                  interface
            - avoid repetition/boilerplate:
                - only one call necessary for potentially complex operations

    - public vs. private Visibility
        - Private members are only accessible through member functions:

            class ascending_sequence {
            private:
                std::vector<int> seq_;
                // ... more private members
            public:
                void insert (int x) { ... }
                auto size () const { return seq_.size(); }
                // ... more public members
            };

            int main () {
                ascending_sequence s;
                s.insert(8);                // OK: 'insert' is public
                auto n = s.size();          // OK: 'size' is public
                auto i = s.seq_[0];         // COMP ERROR: 'seq_' private
                auto m = s.seq_.size();     // COMP ERROR
                s.seq_.push_back(1);        // COMP ERROR
            }

        - struct vs. class - main difference is default visibility

            struct point {              class point {
                ...             <=>     public:
            };                              ...
                                        };

            class point {               struct point {
                ...             <=>     private:
            };                              ...
                                        };

            - Usual use:

            struct      ..simple aggregates of public data
            class       ..private data, member functions, invariants, ...

    - const Member Functions
        - Only const-qualified member functions can be called on const objects

            class ascending_sequence {
                std::vector<int> seq_;
            public:
                void insert { ... }
```

```
            auto size () const { return seq_.size() }
        };

        int main () {
            ascending_sequence s;
            s.insert(88);           // OK: s is not const
            auto const& cs = s;
            cs.insert(5);           // COMP ERR: 'insert' is NOT const
        }
```

- A function taking a const(reference) parameter promises not to modify it
  - This promise is checked & enforced by the compiler

```
        void foo (ascending_sequence const& s) {
            // 's' is const reference ^^^^
            auto n = s.size();                  // OK: 'size' is const
            s.insert(5);            // COMP ERR: 'insert is NOT const
```

- Members are const inside of const-qualified member functions

```
        class monotonous_counter {
            int count_;
        public:
            int reading () const {
                // COMP ERR: "count_" is const:
                count_ += 2;        // <- !!!
                return count_;
            }
        };

        class ascending_sequence {
            std::vector<int> seq_;
        public: ...
            auto size () const {    // 'seq_' is const
                // COMP ERR: calling non-const 'push_back'
                seq_.push_back(0);

                // OK: vector's member 'size()' is const
                return seq_.size();
            }
        };
```

- Member functions can be overloaded by const
  - Two member functions can have the same name (and parameter lists) if one is const-qualified and the other one isn't
    - This makes it possible to clearly distinguish read-only access from read/write actions

```
        class interpolation { ...
            int t_;
            ...
        public:
            ...
            // setter/getter pair:
            void threshold (int t)  { if (t > 0) t_ = t; }
            int  threshold () const { return t_; }

            // write access to a 'node'
            node&       at (int x) { ... }
            // read-only access to a 'node'
            node const& at (int x) const { ... }
        };
```

- Member Declaration vs. Definition

```
        class MyType {
            int n_;
```

```
            // more data members ...
        public:
            // declaration + inline definition
            int count () const { return n_; }

            // declaration only
            double foo (int, int);
        };

        // separate definition
        double MyType::foo (int x, int y) {
            // lots of stuff ...
        }
```

- Definitions of complex member functions
    - Are usually put outside of the class
        - And into a separate source file!
- Small member functions
    - e.g., interface adapter functions (like count)
    - Should be implemented "INLINE"
        - i.e., directly in the class body for maximum performance

- Operator Member Functions
    - Special member function

```
        class X { ...
            Y operator [] (int i) { ... }
        };
```

enables the "subscript operator":

```
        X x;
        Y y = x[0];
```

- Example

```
        class ascending_sequence {
        private:
            std::vector<int> seq_;
        public: ...
            void insert (int x) { ... }
            int operator [] (size_t index) const { return seq_[index]; }
        };

        int main () {
            ascending_sequence s;
            s.insert(9);            // s.seq_: 9
            s.insert(2);            // s.seq_: 2 9
            s.insert(4);            // s.seq_: 2 4 9
            cout << s[0] << '\n';   // prints '2'
            cout << s[1] << '\n';   // prints '4'
        }
```

Initialization

- Member Initialization
    (1) Member Initializers C++11

```
        class counter {                         class Foo {
            // counter should start at 0            int i_ = 10;
            int count_ = 0;                         double x_ = 3.14;
        public:                                 public:
            ...                                     ...
        };                                      };
```

(2) Constructor Initialization Lists
    - constructor ("ctor") = special member function
        - Executed when an object is created

```
            class counter {                 class Foo {
                int count_;                     int i_;     // 1st
            public:                             double x_;  // 2nd
                counter(): count_{0} {}     public:
                ...                             Foo(): i_{10}, x_{3.14} {}
            };                                  // same order: i_, x_
                                                ...
                                            };

            - IMPORTANT
                - Make sure that the member order in initialization lists is
                  always the same as the member declaration order!
                    - A differenc order in the initialization list might
                      lead to indefined behavior such as access to
                      uninitialized memory
                    - Some compilers warn about this
                        - g++/clang++ with -Wall or -Wreorder

    - Constructors
        - constructor ("ctor")
            - Special member function that is executed when an object is created
            - Constructor's "function name" = type name
            - Has no return type
            - Can initialize data members via 'initialization list'
            - Can execute code before first usage of an object
            - Can be used to establish invariants
        - default constructor
            - Constructor that takes no parameters

            class Samples {
                int min_;
                int max_;
                std::vector v_;
            public:
                // default constructor:
                Samples (): min_{0}, max_{1}, v_{min_,max_} {v_.reserve(8);}

                explicit // special constructor:
                Samples (int x): min_{x}, max_{x}, v_{x} {v_.reserve(8);}

                int add (int i) {
                    if (i < min_) min_ = i;
                    else if (i > max_) max_ = i;
                    v_.push_back(i);
                }

                int min () const { return min_; }
                int max () const { return max_; }
                ...
            };

            Samples s1;     // default ctor -> s1.v_: 0 1 _ _ _ _ _ _
            Samples s2 {3}; // special ctor -> s2.v_: 3 _ _ _ _ _ _ _

        - Separate Definition of Constructors
            - Works the same as for other member functions

            class MyType { ...
            public:
                MyType ();  // declaration
                ...
            };
            // separate definition
            MyType::MyType ():: ... { ... }

        - AGAIN
            - Make sure that the member order in initialization lists is always
```

```
              the same as the member declaration order!
                - Above:
                    - In the default constructor we need to make sure to access
                      min_ and max_ in v_{min_,max_} only after they have been
                      initialized

    - Default vs. Custom Constructors
        - No user-defined constructor
            -> compiler generates one:

                class BoringType { public: int i = 0; };

                BoringType obj1;        // OK
                BoringType obj2 {};     // OK

        - At least one special constructor
            -> compiler does NOT generate default constructor:

                class SomeType { ...
                public:
                    // special constructor:
                    explicit SomeType (int x) ... { ... }
                };

                SomeType s1 {1};    // OK: special (int) constructor
                SomeType s2;        // COMP ERR: no default constructor!
                SomeType s3 {};     // COMP ERR: no default constructor!

        - TypeName() = default;
            -> compiler generates implementation of default constructor:

                class MyType { ...
                public:
                    MyType () = default;
                    // special constructor:
                    explicit MyType (int x) ... { ... }
                };

                MyType m1 {1};      // OK: special (int) constructor
                MyType m2;          // OK
                MyType m3 {};       // OK

            - TIP
                - If using '= default':
                    - Make sure to initialize data members with member
                      initializers

    - Explicit Constructors <-> Implicit Conversions

                // functions with a 'Counter' parameter
                void foo (Counter c) { ... }
                void bar (Counter const& c) { ... }

        - Bad: Implicit Conversion          - Good: Explicit Constructors

            class Counter {                     class Counter {
                int count_ = 0;                     int count_ = 0;
            public:                             public:
                                                    explicit
                Counter (int initial):              Counter (int initial):
                    count_{initial} {}                  count_{initial} {}
                ...                                 ...
            };                                  };

            // makes 'Counter' from '2':        // no implicit conversion:
            foo(2);             // OK           foo(2);             // COMP ERR
            bar(2);             // OK           bar(2);             // COMP ERRG
```

```
        foo(Counter{2});    // OK           foo(Counter{2});    // OK
        bar(Counter{2});    // OK           bar(Counter{2});    // OK
```

- IMPORTANT:
  - Make user-defined constructors explicit by default!
    - Implicit conversions are a major source of hard-to-find-bugs!
    - Only use non-explicit constructors, if direct conversions from the parameter type(s) is ABSOLUTELY needed and has an unambiguous meaning
    - Care:
      - As of C++11 one can implicitly construct objects from braced lists of values!

- Constructor Delegation C++11
  - "Call" other constructor in an intialization list

```
class Range {
    int a_;
    int b_;
public:
    // 1) special constructor
    explicit Range (int a, int b): a_{a}, b_{b} {
        if (b_ > a_) std::swap(a_,b_);
    }

    // 2) special [a,a] constructor - delegates to [a,b] ctor
    explicit Range (int a): Range{a,a} {}

    // 3) default constructor - delegates to [a,a] ctor
    Range (): Range{0} {}
    ...
};

Range r1;       // 3) -> r1.a_: 0  r1.b_: 0
Range r2 {3}    // 2) -> r2.a_: 3  r2.b_: 3
Range r3 {4,9}  // 1) -> r3.a_: 4  r3.b_: 9
Range r4 {8,2}  // 1) -> r3.a_: 2  r3.b_: 8
```

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

```
class 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
```

Design, Conventions & Style

- General Guidelines

  - Each type should have exactly one purpose
    because it reduces the likelihood of future modifications to it
    - reduced risk of new bugs
    - keeps code depending on your type more stable

  - Keep data members private and
    use member functions to access/modify data
    - avoids data corruption / allows guarantees of invariants
    - users of your type don’t need to change their code if you change the type’s internal implementation

  - const-qualify all non-modifying member functions
    in order to clearly advertise how and when the internal state of an object changes

```
            - makes it harder to use your type incorrectly
            - enables compiler mutability checks
            - better reasoning about correctness, especially in scenarios with
              concurrent access to objects, e.g., from multiple threads

    - Types in Interfaces

            #include <cstdint>
            #include <numeric_limits>

            class monotonous_counter {
            public:
                // public type alias
                using value_type = std::uint64_t;       // note!
            private:
                value_type count_ = 0;
            public:
                value_type reading () const { return count_; }
                ...
            };

            const auto max = std::numeric_limits<monotonous_counter::value_type>
                ::max();

        - WARNING
            - Don't leak implementation details:
                - Only make type aliases public, if the aliased types are used
                  in the public interface of your class
                    - i.e., used as return types or parameters of public member
                      functions
                - Do not make type aliases public if the aliased types are
                  only used in private member functions or for private data
                  members

    - Member vs. Non-Member
        - How to implement a feature / add new functionality?

            (1) only need to access public data (e.g., via member functions)
                    -> implement as free standing function
            (2) need to access private data
                    -> implement as member function

        - Example: interval-like type gap
            - How to implement a function that makes a new "gap" object with
              both bounds shifted by the same amount?

                        class gap [
                            int a_;
                            int b_;
                        public:
                            explicit gap (int a, int b): a_{a}, b_{b} {}
                            int a () const { return a_; }
                            int b () const { return b_; }
                        };

            - Good: Free-Standing Function

                        gap shifted (gap const& g, int x) {
                            return gap{g.a()+x, g.b()+x};
                        }

                    - Implementation only depends on the public interface of gap
                    - We didn't change type gap itself
                        -> other code depending on it doesn't need to be
                           recompiled

            - Bad: Member Function
```

```
                        class gap {
                            ...
                            gap shifted (int x) const {
                                return gap{a_+x, b_+x};
                            }
                        };

                    - Other users of gap might want a "shifted" function with
                      different semantics
                        - But they are now stuck with ours
                    - All other code depending on gap needs to recompile

    - Avoid Setter/Getter Pairs!
        - use "action" / "verb" functions instead of just "setters"
        - usually models problems better
        - more fine-grained control
        - better code readability / expression of intent
        - Examples
            - Good: descriptive actions

                    class Account { ...
                        void deposit (Money const&);
                        Money try_withdraw (Money const&);
                        Money const& balance () const;

            - Bad: setter/getter pair

                    class Account { ...
                        void set_balance (Money const&);

                        Money const& balance () const;
                    };

    - Naming
        - Names should reflect the purpose of a type/function
        - Examples
            - Good: Understandable

                class IPv6_Address {...};
                class ThreadPool {...};
                class cuboid {...};

                double volume (cuboid const&) {...}

            - Not good: Too generic

                class Manager {...};
                class Starter {...};
                class Pool {...};

                int get_number (Pool const&) {...}

        - Be constistent in naming types and (member) functions
            - Example Style1

                class type_name {...};
                int free_function () {...};
                int member_function () {...};
                int localVariable;
                int memberVariable_;

            - Example Style2

                class TypeName {...};
                int free_function () {...};
                int memberFunction () {...};
                int localVariable;
                int memberVariable_;
```

```
        - WARNING
            - Do not use leading underscores or double underscores in names of
              types, variables, functions, private data members, ...
                - Can invoke undefined behavior!
            - Names beginning with underscores and/or containing double
              underscores:
                - are reserved for the standard library and/or
                - are compiler-generated entities
            - Common convention
                - use trailing underscores for private data members

    - Use Dedicated Types!
        - Why?
            - to restrict input parameter values
            - to ensure validity of intermediate results
            - to guarantee return value validity
        -> Compiler as correctness checker:
            - "if it compiles, it should be correct"

                // unambigous interface
                double volume (Cuboid const&)

                // input guarantee: angle is in radians
                Square make_rotated (Square const&, Radians angle);

                // input: only valid quantity (e.g., > 0)
                Gadget duplicate (Gadget const& original, Quantity times);

                // result guarantee: vector is normalized
                UnitVector3d dominant_direction (WindField const&);

                // avoid confusion with a good units library
                si::kg mass (EllipsoidShell const&, si::g_cm3 density);

                bool has_cycles (DirectedGraph const&);

                // understandable control flow & logic:
                Taxon species1 = classify(image1);
                Taxon species2 = classify(image2);
                Taxon lca = taxonomy.lowest_common_ancestor(species1, species2);


    Example Implementations

    - Example 1: Monotonous Counter

            // New counters start at 0
            // Can only count up, not down
            // Read-only access to current count value

            #include <iostream>         // std::cout
            #include <cstdint>          // std::uint64_t

            class monotonous_counter {
            public:
                using value_type = std::uint64_t;
            private:
                value_type count_ = 0;  // initial
            public:
                monotonous_counter () = default;
                explicit monotonous_counter (value_type init)
                    noexcept: count_{init} {}
                void increment () noexcept { ++count_; }
                [[nodiscard]] value_type reading () const
                    noexcept { return count_; }
            };
```

```
        int main () {
            monotonous_counter c;
            c.increment();
            std::cout << c.reading();   // prints 1
            c.increment();
            c.increment();
            std::cout << c.reading();   // prints 3
        }
```

- Example 2: Ascending Sequence

```
        // Stores integers
        // Read-only access to stored elements by index
        // Can only insert new elements, but not remove them
        // Elements are always sorted in ascending order
        // Content only modifiable through public interface

        #include <iostream>     // std::cout
        #inlcude <vector>       // std::vector
        #include <algorithm>    // std::lower_bound

        class ascending_sequence {
        public:
            using value_type = int;
        private:
            using storage_t = std::vector<value_type>;
            storage_t seq_;
        public:
            using size_type = storage_t::size_type;
            void insert (value_type x) {
                // use binary search to find insert position
                seq_.insert(std::lower_bound(seq_.begin(),
                    seq_.end(), x), x);
            }
            [[nodiscard]] value_type operator [] (size_type idx)
                const noexcept { return seq_[idx]; }
            [[nodiscard]] size_type size ()
                const noexcept { return seq_.size(); }
            // enable range based iteration
            [[nodiscard]] auto begin () const noexcept {
                return seq_.begin();
            }
            [[nodiscard]] auto end () const noexcept {
                return seq_.end();
            }
        };

        int main () {
            ascending_sequence s;   // s.seq_: _
            s.insert(7);            // s.seq_: 7
            s.insert(2);            // s.seq_: 2 7
            s.insert(4);            // s.seq_: 2 4 7
            s.insert(9);            // s.seq_: 2 4 7 9
            s.insert(5);            // s.seq_: 2 4 5 7 9
            std::cout << s[3]       // prints 7
            for (auto x : s) {
                std:cout << x << ' ';   // 2 4 5 7 9
            }
            // use type aliases
            ascending_sequence::value_type x = 1;
            ascending_sequence::size_type  n = 2;
        }
```

Pointers

```
    - Why do we need them?

        - Observince Objects
            - indirection without copying: referencing/keeping track of objects
            - if we want to change the target of an indirection at runtime
                -> can't use references

        - Accessing Dynamic Memory
            - access objects of dynamic storage
                - i.e., objects whose lifetime is not tied to a variable/scope

        - Building Dynamic, Node-Based Data Structures

            - Dynamic Arrays

                +---+---+---+---+---+---+---+---+---+---+---+
                |   | 4 | * |   |   | 0 | 1 | 2 | 3 |   |   |
                +---+---+---+---+---+---+---+---+---+---+---+
                      |               ^
                      |_______________|

            - Linked Lists

                +---+---+---+---+---+---+---+---+---+---+---+
                |   | 0 | * |   |   | 1 | * |   | 2 | * |   |
                +---+---+---+---+---+---+---+---+---+---+---+
                      |               ^   |       ^   |
                      |_______________|   |_______|   |_____

            - Trees / Graphs

                +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                |   | 2 | * | * |   | 5 | * | * |   |   |   | 7 | * | * |   |
                +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                      ^                   |   |               ^
                      |___________________|   |_______________|


    - Pointer to Object of Type T
        - stores a memory address of an object of type T
        - can be used to inspect/observe/modify the target object
        - can be redirected to a different target (unlike references)
        - may also point to no object at all (be a "Null Pointer")


                            MEMORY
          pointer to       +-------+                +---+     +---+
           object c        |       | 0x07           | p | --> | 5 |
                |          +-------+                +---+     +---+
                +-------> p | 0x03 | 0x06 --+                    c
                           +-------+        |
                           |       | 0x05   |
                           +-------+        |
                           |       | 0x04   |
                           +-------+        |
                         c |   5   | 0x03 <-+
                           +-------+
                           |       | 0x02
                           +-------+
                           |       | 0x01
                           +-------+
                           |       | 0x00
                           +-------+


        - Raw Pointers: T*
            - essentially an (unsigned) integer variable stroing a mem address
            - size: 64 bits on 64 bit platforms
```

```
            - many raw pointers can point to the same address/object
            - lifetimes of pointer and target object are independent

        - Smart Pointers C++11

            std::unique_pointer<T>      - used to access dynamic storage
                                            - i.e., objects on the "heap"
                                        - only one unique_ptr per object

            std::shared_pointer<T>      - used to access dynamic storage
            std::weak_pointer<T>            - i.e., objects on the "heap"
                                        - many shared_ptrs and/or weak_ptrs
                                          per object
                                        - target object lives as long as at
                                          least one shared_ptr points to it

    - Operators

        - Address Operator &
                                                              STACK
                char  c = 65;                               +--------+
                char* p = &c;                               |        | 0x05
                                                            +--------+
                    - raw pointer variable of type T*     p | 0x03   | 0x04 --+
                      can store an address of an object     +--------+        |
                      of type T                           c |   65   | 0x03 <--+
                                                            +--------+
                                                            |        | 0x02
                                                            +--------+

        - Dereference Operator *
                                                              STACK
                char  c = 65;                               +--------+
                char* p = &c;                               |        | 0x05
                p* = 88;                                    +--------+
                char  x = *p;                             x |   88   | 0x04
                                                            +--------+
                    - &c returns memory address of c      p | 0x03   | 0x04 --+
                    - *p accesses value at the addr         +--------+        |
                      in p                                c |   88   | 0x02 <--+
                                                            +--------+
                                                            |        | 0x01
                                                            +--------+

        - Member Access Operator ->
                                                              STACK
                struct Coord {                              +--------+
                    char x = 0;                             |        | 0x06
                    char y = 0;                             +--------+
                };                                        p | 0x03   | 0x05 --+
                                                            +--------+        |
                Coord a {12,34};                        b.x |   34   | 0x04   |
                Coord* p = &a;                              +--------+        |
                                                        a.x |   12   | 0x03 <--+
                char v = p->x;  // v = 12                   +--------+
                char w = p->y;  // w = 34                   |        | 0x02
                                                            +--------+
                // alternative:
                char s (*p).x;  // s = 12
                char t (*p).y;  // t = 34

    - Syntax

        +-----------------+--------------------------+--------------------------+
        |                 |           *              |            &             |
        +-----------------+--------------------------+--------------------------+
        | Type Modifier   | Pointer Declaration      | Reference Declaration    |
        |                 | Type* ptr = nullptr;     | Type& ref = variable;    |
```

```
+-----------------+--------------------------+-------------------------+
| Unary Operator  | Dereferencing            | Taking Address          |
|                 | value = *pointer;        | pointer = &variable;    |
+-----------------+--------------------------+-------------------------+
| Binary Operator | Multiplication           | Bitwise AND             |
|                 | product = expr1 * expr2; | bitand = expr1 & expr2; |
+-----------------+--------------------------+-------------------------+
```

- Declaration Pitfall

```
int*   p1, p2;  // int*, int
int   *p1, *p2  // int*, int*
```

  - Better and Unambiguous

```
int* p1 = ...;
int* p2 = ...;
```

- Redirection
  - Unlike references, pointers can be redirected

```
int a = 0;        |
int b = 0;        |       a: 0    b: 0
                  |
int* p = &a;      | p->a  a: 0    b: 0
                  |
*p = 2;           | p->a  a: 2    b: 0
                  |
p = &b;           | p->b  a: 2    b: 0
                  |
*p = 9;           | p->b  a: 2    b: 9
                  |
cout << a;        | 2
cout << b;        | 9
```

- nullptr C++11

  - special pointer value
  - is implicitly convertible to false
  - not necessarily represented by 0 in memory! (depends on platfrom)

  - Coding Convention: nullptr signifies "value not available"
    - set pointer to nullptr or valid address on initialization
    - check if not nullptr before dereferencing

```
int* p = nullptr;   // init to nullptr
if (...) {
    int i = 5;
    p = &i;         // assign valid address
    ...
    // check before dereferencing!
    if (p) *p = 7;
    ...
    // set to nullptr, signalling 'not available'
    p = nullptr;
}
// i's memory is freed, any pointer to i would be invalidated!
```

- const and Pointers

  - Purposes
    (1) read-only access to objects
    (2) preventing pointer redirection

  - Syntax

```
+-------------------+------------------+----------------+
|                   | pointed-to value | pointer itself |
| pointer to type T |    modifiable    |   modifiable   |
+-------------------+------------------+----------------+
| T *               |        Y         |       Y        |
| T const *         |        N         |       Y        |
| T * const         |        Y         |       N        |
| T const * const   |        N         |       N        |
+-------------------+------------------+----------------+
```

- Examples

```
int i = 5;
int j = 8;

int const* cp = &i;
*cp = 8;    // COMPILER ERROR: pointed-to value is const
cp = &j;    // OK

int *const pc = &i;
*pc = 8;    // OK
pc = &j;    // COMPILER ERROR: pointer itself is const

int const*const cpc = &i;
*cpc = 8;   // COMPILER ERROR: pointed-to value is const
cpc = &j;   // COMPILER ERROR: pointer itself is const
```

- An ongoing debate about style

```
        East const             |          const West
                               |
  one consistent rule: what's  |  (still) more widespread, but
   left of const is constant   |        less consistent
---------------------------------------------------------------
                               |
   int const c = ...;          | const int c = 1;
   int const&  cr = ...;       | const int& cr = ...;
   int const * pc = ...;       | const int* pc = ...;
   int *const  cp = ...;       | int *const cp = ...;
   int const*const cpc = ...;  | const int *const cpc = ...;
```

- The "this" Pointer

  - available inside member functions

```
this        ..returns the address of an object itself
this->      ..can be used to access members
*this       ..accesses the object itself
```

  - Example

```
class IntRange {
    int l_ = 0;
    int r_ = 0;
public:
    explicit
    IntRange (int l, int r): l_{l}, r_{r} {
        if (l_ > r_) std::swap(l_, r_);
    }
    int left ()  const { return l_; }
    // can also use 'this' to access members:
    int right () const { return this->r_;}
    ...
    // returns reference to object itself
    IntRange& shif (int by) {
```

```
                l_ += by;
                r_ += by;
                return *this;
            }
            IntRange& widen (int by) {
                l_ -= by;
                r_ += by;
                return *this;
            }
        };

        IntRange r {1,3};                                      │ 1 3
        r1.shift(1);                                           │ 2 4
        r1.shift(2).widen(1);    // chaining possible!         │ 3 7
```

- Forward Declarations of Types
    - Sometimes necessary if one needs two types to refer to each other:

```
        // forward declaration
        class Hub;

        class Device {
            Hub* hub_;
            ...
        };

        class Hub {
            std::vector<Device const*> devs_;
            ...
        };
```

```
                              +------------+
                              │ Device 1   │
                              +------------+
                                    ^
                                    |
                                    v
              +------------+      +------------+      +------------+
              │ Device 2   │ <->  │   Hub 1    │ <->  │ Device 3   │
              +------------+      +------------+      +------------+
                                    ^
                                    |
                                    v
                              +------------+      +------------+
                              │   Hub 2    │ <->  │ Device 4   │
                              +------------+      +------------+
```

- In order to define a type, the memory sizes of all its members must be known
    - This is only possible if the full definition of all members is known
        - We can 'declare' the existence of Hub, because Device only needs a pointer to it
        - NOTE: all pointer types have the same size

- Avoid Pointers If Possible

    - Pointers are prone to dangling
        - dangling = pointer points to an invalid/inaccessible memory addr
        - value stored in pointer can be any address
        - programmer has to make sure pointer target is valid/still exists
        - Examples

```
        int* p;      // p not initialized!
        *p = 7;      // UNDEFINED BEHAVIOR
```

```
            p = nullptr;
            *p = 7;     // UNDEFINED BEHAVIOR access to nullptr

            {
                int x = 8;
                p = &x;
            }
            *p = 7;     // UNDEFINED BEHAVIOR access to freed memory
```

- Error-prone argument passing

```
            void swap_values (int* a, int* b) {
                int t = *a;
                *a = *b;
                *b = t;
            }

            int x = 3; y = 4;
            swap_values(&x, &y)          // OK
            swap_values(&x, 0)           // UNDEFINED BEHAVIOR
            swap_values(&x, nullptr)     // UNDEFINED BEHAVIOR
```

- TIP:
    - Prefer references if possible
        - Especially for function parameters

Destructors

- Special Member Functions

T::T()	default constructor	runs when new T obj is created
T::T(param...)	special constructor	runs when new T obj is created with argument(s)
T::~T()	destructor	runs when existing T obj is destroyed

- The compiler generates a default constructor and a destructor if don't defined by us
- Four more special members (also automatically generated):

T::T(T const&)	copy constuctor
T& T::operator = (T const&)	copy assignment operator
T::T(T &&)	move constructor
T& T::operator = (T &&)	move assignment operator

- User-Defined Constuctor and Destructor

```
        class Point { ... };

        class Test {
            std::vector<Point> w_;
            std::vector<int> v_;
            int i_ = 0;
        public:
            Test() {
                std::cout << "constuctor\n";
            }
            ~Test() {
                std::cout << "destructor\n";
            }
            // more member functions ...
        };
```

```
            ...
            if (...) {
                ...
                Test x; // prints 'constructor'
                ...
            } // prints 'destructor'


        - Execution Order on Destruction
            - After the destructor body has run the destructors of all data
              members are executed in reverse declaration order
                - This happens automatically and cannot be (easily) changed

                    x goes out of scope -> executes ~Test():
                        (1) std::cout << "destructor\n";
                        (2) x's data members are destroyed:
                            (2.1) i_ is "destroyed" (fundamental types don't
                                  have a destructor)
                            (2.2) v_ is destroyed
                                    -> executes destructor ~vector<int>():
                                        - vector "destroys" int elements in its
                                          buffer; fundamental type -> no dstrctr
                                        - deallocates buffer on the heap
                                        - v_'s remaining data members are
                                          destroyed
                            (2.3) w_ is destroyed
                                    -> executes destructor ~vector<Point>():
                                        - vector destroys 'Point' elements in
                                          its buffer
                                        - each ~Point() destructor is executed
                                        - deallocates buffer on the heap
                                        - w_'s remaining data members are
                                          destroyed


    - RAII
        - Resource Acquisition Is Initialization

            - object construction: acquire resource
            - object destruction:  release resource

        - Example: std::vector
            - Each vector object is owner of        vector<int> v {0,1,2,3,4};
              a separate buffer on the heap
              where the actual content is           contiguous
              stored                                buffer on   0 1 2 3 4
            - This buffer is allocated on              HEAP     ^
              demand and de-allocated if the        ------------|-------------
              vector object is destroyed            vector      |
                                                    object on   v
                                                      STACK
        - Ownership
            - An object is said to be an owner of a resource (memory, file
              handle, connection, thread, lock, ...) if it is responsible for
              its lifetime (initialization/creation, finalization/destruction)

        - Reminder: C++ uses Value Semantics
            - means, variables refer to objects themselves
                - i.e., they are not just references/pointers
            - Properties:
                - deep copying
                - deep assignment
                - deep ownership                    // !!
                - value-based comparison
            - No need for a garbage collector:
                - Because the lifetime of members is tied to its containing
                  objects
```

- Example: Resource Handler
  - Common Situation
    - Need to use an external (C) library that does it's own resource management
      - Resources are usually handled with pairs of
        - (1) initialization functions
          - e.g., lib_init()
        - (2) finalization functions
          - e.g., lib_finalize()
  - Problem: Resource Leaks
    - Common to forget to call the finalization functions
    - Leads to:
      - Hung-up devices
      - Memory not being freed
      - ...

  - Solution: RAII Wrapper

    (1) Call initialization function in constructor
    (2) Call finalization function in destructor

    - Additional advantage:
      - Wrapper class can also be used to store context information like connection details, device ids, etc. that are only valid between initialization and finalization
    - Such wrapper should in most cases be made non-copyable
      - Since it handles unique resources

```
#include <gpulib.h>

class GPUContext {
    int gpuid_;
public:
    explicit
    GPUContext (int gpuid = 0): gpuid_{gpuid} {
        gpulib_init(gpuid_);
    }
    ~GPUContext () {
        gpulib_finalize(gpuid_);
    }
    [[nodiscard]] int gpu_id () const noexcept {
        return gpuid_;
    }
// make non-copyable:
    GPUContext (GPUContext const&) = delete;
    GPUContext& operator = (GPUContext const&) = delete;
};

int main () {
    ...
    if (...) {
        // create/initialize context
        GPUContext gpu;
            // do something with it
        ...
    }   // automatically finalized!
    ...
}
```

- Example: RAII Logging
  - constructor of 'Device' gets pointer to a 'UsageLog' object
  - 'UsageLog' can be used to record actions during a 'Device' object's lifetime
  - destructor informs 'UsageLog' if 'Device' is no longer present
  - the 'UsageLog' could also count the number of active devices, ...

```
            class File { ... };
            class DeviceID { ... };

            class UsageLog {
            public:
                explicit UsageLog (File const&);
                ...
                void armed (DeviceID);
                void disarmed (DeviceID);
                void fired (DeviceID);
            };

            class Device {
                DeviceID id_;
                UsageLog* log_;
                ...
            public:
                explicit
                Device (DeviceID id, UsageLog* log = nullptr):
                    id_{id}, log_{log}, ...
                {
                    if (log_) log_->armed(id_);
                }

                ~Device () { if (log_) log_->disarmed(id_); }
                void fire () {
                    ...
                    if (log_) log_->fired(id_);
                }
                ...
            };

            int main () {
                File file {"log.txt"}
                UsageLog log {file};
                ...
                Device d1 {DeviceID{1}, &log};
                d1.fire();
                {
                    Device d2 {DeviceID{2}, &log};
                    d2.fire();
                }
                d1.fire();
            }

        - log.txt

            device 1  armed
            device 1  fired
            device 2  armed
            device 2  fired
            device 2  disarmed
            device 1  fired
            device 1  disarmed


    - The Rule of Zero
        = (try to) "write zero special member functions

        - Avoid writing special member functions unless you need to do
          RAII-style resource management or lifetime-based tracking
            - The compiler generated default constructor and destructor
              are sufficient in most cases

        - Initialization doesn't always require writing constructors
            - Most data members can be initialized with Member Initializers

        - Do not add "empty destructors" to types!
```

- The presence of a user-defined destructor prevents many optimizations and can seriously impact performance!

- You almost never need to write destructors
    - In modern C++ memory management strategies are mostly (and should be) encapsulated in dedicated classes (containers, smart pointers, allocators, ...)

Exceptions

Intro

- What Are Exceptions?

    - Objects that can be "thrown upwards" the call hierarchy
        - throwing transfers control back to the caller of the current func
        - they can be "caught"/handled via 'try .. catch' blocks
        - if not handled, exceptions propagate up until they reach 'main'
        - if an exception is not handled in 'main'
            -> 'std::terminate' will be called
        - default behavior of 'std::terminate' is to abort the program

```
                         void bar() {
                             ...
                     +--     throw error{};
                     |       ...
           (1)       |   }
        go back to   |
          caller     |   void foo() {
                     |       ...
                     +->     bar();      --+
                             ...           |
                         }                 |      (2)
                                           |  exception not
                         int main() {      |  handled -> go
    (3)                      ...           |  back to caller
exception not        +--     foo();    <---+
 handled ->          |       ...
 terminate           +-> }
 program
```

- First Example

    - Original motivation for exceptions
        - report the failure of a constructor to properly initialize an object (i.e., failure to establish the required class invariants)
            - NOTE:
                - A constructor does not have a return type that could be used for error reporting

```
#include <stdexcept> // standard exception types

class Fraction {
    int numer_;
    int denom_;
public:
    explicit constexpr
    Fraction (int numerator, int denominator):
        numer_{numerator}, denom_{denominator}
    {
        if (denom_ == 0)
            throw std::invalid_argument{"denominator must not be
                zero"};
    }
    ...
```

```
    };

    int main () {
        try {
            int d = 1;
            std::cin >> d;
            Fraction f {1,d};
            ...
        }
        catch (std::invalid_argument const& e) {
            // deal with / report error here
            std::cerr << "error: " << e.what() << '\n';
        }
        ...
    }
```

- Usages: Report Contract Violations

    (1) Precondition Violations
        - Precondition = expectation regarding inputs (valid func args)
        - Violation Examples
            - Out-of-bounds container index
            - Negative number for square
        - "Wide Contract" Functions perform precondition checks before using their input values
            - Usually not used in performance-critical code:
                - One does not want to pay the cost of input validity checks if passed-in args are already known to be valid

    (2) Failure To Establish / Preserve Invariants
        - Public member function fails to set valid member values
        - Example
            - Out of memory during vector growth

    (3) Postcondition Violations
        - Postcondition = expectation regarding outputs (return values)
        - Violations:
            (a) function fails to produce valid return value
            (b) function corrupts global state
        - Examples
            - Constructor fails
            - Can't return result of division by zero

    - Advantages / Disadvantages of Exceptions

        (+) separation of error handling code from business logic
        (+) centralization of error handling (higher up the call chain)
        (+) nowadays negligible performance impact when no exception is thrown
        (-) usually performance impact when exception is thrown
        (-) performance impact due to extra validity checks
        (-) easy to produce resource/memory leaks (see below)

- Alternatives to Exceptions

    - Input Value Is Invalid (Precondition Violation)
        - "narrow contract" functions:
            - make sure arguments are valid before passing them
        - use parameter types that preclude invalid values
        - this is preferred nowadays for better performance

    - Failed To Establish / Preserve Invariants
        - error states/flags
        - set object to special, invalid value/state

    - Can't Return Valid Value (Postcondition Violation)
        - return error code via separate output parameter (reference or pointer)

```
            - return special, invalid value
            - use special vocabulary type that can either:
                (a) contain a valid result or
                (b) "nothing", like:
                        - C++17: 'std::optional'
                        - Haskell: 'Maybe'

    - Standard Library Exceptions
        - Exceptions are one of the few places wher the C++ standard library
          uses inheritance:
            - All standard exception types are subtypes of std::exception

                std::exception
                    logic_error
                        invalid_argument
                        domain_error
                        length_error
                        out_of_range
                        ...
                    runtime_error
                        range_error
                        overflow_error
                        underflow_error
                        ...

            - Example

                try {
                    throw std::domain_error {
                        "Error Text"
                    };
                }
                catch (std::invalid_argument const& e) {
                    // handle only 'invalid_argument'
                    ...
                }
                // catches all other std exceptions
                catch (std::exception const& e) {
                    std::cout << e.what()
                    // prints "Error Text"
                }

        - "Wide Contract" Functions
            - Offered by some standard library containers
            - Report invalid input values by throwing exceptions

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

                // narrow contract: no checks, max performance
                int a = v[6];   // UNDEFINED BEHAVIOR

                // wide contract: checks if out of bounds
                int b = v.at(6);    // throws std::out_of_range

    - Handling

        Re-Throwing                         Catching All Exceptions

        try {                               try {
            // potentially throwing             // potentially throwing
            // code                             // code
        }                                   }
        catch (std::exception const&) {     catch (std::exception const&) {
            throw; // re-throw                  // handle failure
                   // exception(s)
        }                                   }

        - Centralize Exception Handling!
```

```
        - Avoids code duplication
            - If same exception types are thrown in many different places
        - Useful for converting exceptions into error codes

                void handle_init_errors () {
                    try { throw;  // re-throw! }
                    catch (err::device_unreachable const& e) { ... }
                    catch (err::bad_connection const& e) { ... }
                    catch (err::bad_protocol const& e) { ... }
                }
                void initialize_server (...) {
                    try {
                        ...
                    } catch (...) { handle_init_errors(); }
                }
                void intitialize_clients (...) {
                    try {
                        ...
                    } catch (...) { handle_init_errors(); }
                }


Problems and Guarantees

- Resource Leaks

    - Almost any piece of code might throw exceptions
        -> heavy impact on design of C++ types and libraries
    - Potential source of subtle resource/memory leaks, if used with
        - external C libraries that do their own memory management
        - (poorly designed) C++ libraries that don't use RAII for
          automatic resource management
        - (poorly designed) types that don't clean up their resources
          on destruction

    - Example: Leak due to C-style resource handling
        - i.e., two separate functions for resource initialization (connect)
          and finalization (disconnect)

    void add_to_database (database const& db, std::string_view filename) {
        DBHandle h = open_dabase_connection(db);

        auto f = open_file(filename);
        // if 'open_file' throws -> connection NOT CLOSED!

        // do work...
        close_database_connection(h);   // <- not reached if 'open_file'
                                        // threw
    }

- Use RAII To Prevent Leaks!

    - What's RAII again?
        - constructor: resource acquisition
        - destructor:  resource release/finalization

    - If exception is thrown
        -> objects in local scope destroyed: destructors called
        -> with RAII: resources properly released/finalized

            class DBConnector {
                DBHandle handle_;
            public:
                explicit
                DBConnector (Database& db):
                    handle_{make_database_connection(db)} {}
                ~DBConnector () { close_database_connection(handle_); }
```

```
        // make connector non-copyable:
        DBConnector (DBConnector const&) = delete;
        DBConnector& operator = (DBConnector const&) = delete;
    };

    void add_to_database (database const& db, std::string_view
    filename) {
        DBConnector(db);

        auto f = open_file(filename);
        // if 'open_file' throws -> connection closed!

        // do work normally...
    }   // connection closed!
```

- Write an RAII wrapper if using a library (e.g., from C) that employs separate functions for initialization and finalization of resources
- Often makes sense to...
    - Make the wrapper non-copyable (especially if not having control over the referenced external resources):

        (1) Delete the copy constructor
        (2) Delete copy assignment operator

- Destructors: Don't Let Exceptions Escape!
    - ...or resources may be leaked!

```
        class E {
        public:
            ~E () {
                // throwing code -> BAD!
            }
            ...
        };
        class A {
            // some members:
            G g; F f; E e; D d; C c; B b;
            ...
        };


  ~A(){ }
       |           ~E runnint
       | +-+ +-+ +-+
       | | | | | | |
       v | v | v | v
     +---+---+---+---+---+---+
     | b | c | d | e | f | g |
     +---+---+---+---+---+---+
     |  already  | | | still |
       destroyed   |   alive
                   v
                 throws -> not all members destroyed properly!
                            (destructors of f and g are not called)
```

- In Destructors:
    - Catch Exceptions From Potentially Throwing Code!

```
        class MyType {
        public:
            ~MyType () { ...
                try {
                    // y throwing code...
                } catch ( /* ... */ ) {
                    // handle exceptions...
                } ...
            }
        };
```

```
  - Exception Guarantees
      - In case an exception is thrown:

          - No Guarantee
              - must be assumed of any C++ code unless its documentation says
                otherwise:
                  - operations may fail
                  - resources may be leaked
                  - invariants may be violated (= members may contain invalid
                    values)
                  - partial execution of failed operations may cause side
                    effects (e.g., output)
                  - exceptions may propagate outwards

          - Basic Guarantee
              - invariants are preserved, no resources are leaked
              - all members will contain valid values
              - partial execution of failed operations may cause side effects
                  - e.g., values might have been written to file

              - This is the least one should aim for!

          - Strong Guarantee ("commit or rollback semantics")
              - operations can fail, but will have no observable side effects
              - all members retain their original values

              - Memory-allocating containers should provide this guarantee
                  - i.e., containers should remain valid and unchanged if
                    memory allocation during growth fails

          - No-Throw Guarantee (strongest)
              - operations are guaranteed to succeed
              - exceptions not observable from outside:
                  - either:
                      (a) none thrown
                      (b) caught internally
              - documented and enforced with 'noexcept' keyword

              - Prefer this:
                  (1) in high performance code
                  (2) on resource constrained devices

  - Not-Throw Guarantee: noexcept C++11

              void foo () noexcept { ... }

          - 'foo' promises to never throw exceptions / let any escape
          - if an exception escapes from a noexcept function anyway:
              -> program will be terminated

      - Think carefully if you can keep the no-throw promise!

          - noexcept is part of a function's interface
              - even part of a function's type as of C++17
          - changing noexcept functions back into throwing ones later might
            break calling code that relies on not having to handle exceptions

      - Conditional noexcept

              A  noexcept( expression )       declares 'A' as noexcept if
                                              expression yields true
              A  noexcept( noexcept( B ))     declares 'A' as noexcept if 'B'
                                              is noexcept

          - Example
```

```
            constexpr int N = 5;

            // 'foo' is noexcept if N < 9
            void foo () noexcept( N < 9 ) { ... }

            // 'bar' is noexcept if foo is
            void bar () noexcept( noexcept(foo()) ) {
                ...
                foo();
                ...
            }

        - noexcept(true) by default
            - are all implicitly-declared special members
                - default constructors
                - destructors
                - copy constructors, move constructors
                - copy-assignment operators, move-assignment operators
                - inherited constructors
                - user-defined destructors
            - unless
                - they are required to call a functino that is noexcept(false)
                - an explicit declaration says otherwise


    More

    - Termination Handler
        - Uncaught exception in 'main'
                (1) std::terminate is called
                (2) which calls the termination handler
                (3) which by default calls std::abort
                        - Terminates the program normally

        - Set custom handler

                std::set_terminate(handler);

            - sets the function(object) that is called by std::terminate

                #include <stdexcept>
                #include <iostream>

                void my_handler () {
                    std::cerr << "Unhandled Exception!\n";
                    std::abort();   // terminate program
                }
                int main () {
                    std::set_terminate(my_handler);
                    ...
                    throw std::exception{};
                    ...
                }

    - Exception Pointers

        std::current_exception

            - captures the current exception object
            - returns a std::excpetion_ptr referring to that exception
            - if there's no exception
                -> an empty std::exception_ptr is returned

        std::exception_ptr

            - either holds a copy or a reference to an exception

        std::rethrow_exception(exception_ptr)
```

- throws an exception object referred to by an exception pointer

- Example

```
#include <exception>
#include <stdexcept>

void handle_init_errors (std::exception_ptr eptr) {
    try {
        if (eptr) std::rethrow_exception(eptr);
    }
    catch (err::bad_connection const& e) { ... }
    catch (err::bad_protocol const& e) { ... }
}

void initialize_client () {
    if (...) throw err::bad_connection;
    ...
}

int main () {
    std::exception_ptr eptr;
    try {
        initialize_client();
    ...
    } catch (...) {
        eptr = std::current_exception();
    }
    handle(eptr);
} // eptr destroyed -> captured exceptions destroyed
```

- Counting Uncaught Exceptions C++17

  std::uncaught_exceptions

  - returns the number of currently unhandled exceptions in the current thread

```
#include <exception>

void foo () {
    bar(); // might have thrown
    int count = std::uncaught_exceptions();
    ...
}
```