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

--------------------------------------------------------------------------------
Diagnostics
--------------------------------------------------------------------------------

    ----------------------------------------------------------------------------
    Basics: Warnings & Tests
    ----------------------------------------------------------------------------

    Terms And Techniques

        - Warnings          compiler messages hinting at        (see below)
                            potentially problematic runtime
                            behavior / subtle pitfalls
        - Assertions        statements for comparing and        (see below)
                            reporting expected and actual
                            values of expressions
        - Testing           compare actual and expected         (see below)
                            behavior of parts or entire prog
        - Code Coverage     how much code is actually           gcov, ...
                            executed and/or tested
        - Static Analysis   finds potential runtime problems    ASAN, UBSAN. ...
                            like undefined behavior by
                            analyzing the source code
        - Dynamic Analysis  finds potential problems like       valgrind, ...
                            memory leaks by running the
                            actual program
        - Degbugging        step through code at runtime and    (next up)
                            inspect in-memory values
        - Profiling         find out how much each function/
                            loop/code block contributes to
                            the total running time, memory
                            consumption, ...
        - Micro Bench-      small tests that measure the
          marking           runtime of single a function or
                            a block of statements/calls rather
                            than the whole program

    - Remember: Use Dedicated Types!

        - to restrict input parameter values
        - to ensure validity of intermediate results
        - to guarantee validity of return values


    Compiler Warnings

        - Compiler Error   = program not compilable
        - Compiler Warning = program compilatble, but there is a problematic
                             piece of code that might lead to runtime bugs

    - gcc/clang Options

        - Most important

            -Wall, -Wextra, -Wpedantic, -Wshadow, -Werror
            -fsitize=undefined,address

        - Recommended Set (Production Level)

            -Wall -Wextra -Wpedantic -Wshadow -Wconversion -Werror
            -fsanitize=undefined,address -Wfloat-equal -Wformat-nonliteral
            -Wformat-security -Wformat-y2k -Wformat=2 -Wimport -Winvalid-pch
```

```
            -Wlogical-op -Wmissing-declarations -Wmissing-field-initializers
            -Wmissing-format-attribute -Wmissing-include-dirs -Wmissing-noreturn
            -Wnested-externs -Wpacked -Wpointer-arith -Wredundant-decls
            -Wstack-protector -Wstrict-null-sentinel -Wswitch-enum -Wundef
            -Wwrite-strings

        - High Performance / Low Memory / Security

            Might be VERY noisy!

            -Wdisabled-optimization -Wpadded -Wsign-conversion -Wsign-promo
            -Wstrict-aliasing=2 -Wstrict-overflow=5 -Wunused -Wunused-parameter


    Assertions

    - Runtime Assertions

        #include <cassert>
        assert(bool_expression);

            - Aborts the program if expresion yields false

            - Use cases:
                - check expected values/conditions at runtime
                - verify preconditions (input values)
                - verify invariants (e.g., intermediate states/results)
                - verify postconditions (output/return values)
            - Important:
                - Runtime assertions should be deactivated in release builds to
                  avoid any performance impact
            - Example

                    #include <cassert>

                    double sqrt (double x) {
                        assert ( x >= );
                        ...
                    }
                    ...
                    double r = sqrt(-2.3);

        - Commas must be protected by parentheses

            - assert
                - is a preprocessor macro
                    - commas are interpreted as macro argument separator:

                        assert( min(1,2) == 1  );       // ERROR
                        assert((min(1,2) == 1 ));       // OK

        - Messages

            - Can be added with a custom macro (there is no standard way)

                        #define assertmsg(expr, msg) assert (((void)msg, expr))

                        assertmsg(1+2==2, "1 plus 1 must be 2");

        - (De-)Activation - g++/clang

            - Assertions are deactivated by defining preprocessor macro:

                        NDEBUG

                - Example
                    - With compiler switch:
```

```
                    $ g++ -DNBEBUG ...
```

- Static Assertions C++11

```
    static_assert(bool_constexpr, "message");
    static_assert(bool_constexpr);  C++17
```

    - Aborts compilation if a compile-time constant expression yields false

```
        ...
        using index_t = int;
        index_t constexpr DIMS = 1; // oops

        void foo () {
            static_assert(DIMS > 1, "DIMS must be at least 2");
            ...
        }

        index_t bar (...) {
            static_assert(
                std::numeric_limits<index_t>::is_integer &&
                std::numeric_limits<index_t>::is_is_signed,
                "index type must be a signed integer");
            ...
        }
```

Testing

- Guidelines

    - Use Assertions
        - to check expectations/assumptions that are not already expressible/guaranteed by types:
            - expected values that are only available at runtime
            - preconditions (input values)
            - invariants (e.g., intermediate states/results)
            - postconditions (output/return values)
        - Note:
            - Runtime assertions should be deactivated in release builds
                - to avoid any performance impact

    - Write Tests
        - as soon as the basic purpose and interface of a function or type is decided
            - faster development: less need for time-consuming logging and debugging sessions
            - easier performance tuning: one can continuously check if still correct
            - documentation: expectations/assumptions are written down in code

    - Use A Testing Framework
        - More convenient and less error-prone:
            - predefined checks, setup facilities, test runner, ...

        - Beginners/Smaller Projects:

```
                    doctest
```

            - very compact and self-documenting style
            - easy setup: only include a single header
            - very fast compilation

        - Larger Projects:

```
                    Catch2
```

```
            - same basic philosophy as doctest
            - value generators for performing same test with different
              values
            - micro benchmarking with timer, averaging, etc...
            - slower to compile and slightly more complicated to set up
              than doctest

- doctest - Simple Test Case (from doctest tutorial)

            #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
            #include "doctest.h"

            int factorial (int n) {
                if (n <= 1) return n;
                return factorial(n-1) * n;
            }

            TEST_CASE("testing factorial") {
                CHECK(factorial(0)  == 1);
                CHECK(factorial(1)  == 1);
                CHECK(factorial(2)  == 2);
                CHECK(factorial(3)  == 6);
                CHECK(factorial(10) == 3628800);
            }

        - The test fails, because the implementation of facatorial doesn't
          handle the case of n = 0 properly

- doctest - Subcases (from doctest tutorial)

            TEST_CASE("vectors can be sized and resized") {
                std::vector<int> v(5);

                REQUIRE(v.size() == 5);
                REQUIRE(v.capacity() >= 5);

                SUBCASE("push_back increases the size") {
                    v.push_back(1);
                    CHECK(v.size() == 6);
                    CHECK(v.capacity() >= 6);
                }
                SUBCASE("reserve increases the capacity") {
                    v.reserve(6);
                    CHECK(v.size() == 5);
                    CHECK(v.capacity() >= 6);
                }
            }

        - If CHECK fails:
            - test is marked as fails, but execution continues
        - If REQUIRE fails:
            - execution stops

- Don't Use con/cout/cerr Directly!

    - Bad: direct use of global I/O streams makes functions or types hard
      to test:

            void bad_log (State const& s) { std::cout << ... }

    - Good: In Functions: Pass Streams By Reference

            struct State { std::string msg; ... };

            void log (std::ostream& os, State const& s) { os << s.msg; }

            TEST_CASE("State Log") {
```

```
                State s {"expected"};
                std::ostringstream oss;
                log(oss, s);
                CHECK(oss.str() == "expected");
            }
```

- Good: Class Scope: Store Stream Pointers
  - But try to write types that are independent of streams or any other particular I/O method!

```
            class Logger {
                std::ostream* os_;
                int count_;
            public:
                explicit
                Logger (std::ostream* os): os_{os}, cout_{0} {}
                ...
                bool add (std::string_view msg) {
                    if (!os_) return false;
                    *os_ << cout_ <<": "<< msg << '\n';
                    ++cout_;
                    return true;
                }
            };

            TEST_CASE("Logging") {
                std::ostringstream oss;
                Logger log {&oss};
                log.add("message");
                CHECK(oss.str() == "o: message\n");
```

Debugging With gdb

- gdb / Frontends

- gdb	- Qt Creator can connect to gdb
- cgdb	- Visual Studio Code can connect to gdb
- gdbgui	- Vim -> :help Termdebug

- Example

```
        $ vi sum.cpp

            #include <iostream>
            #include <cstdlib>

            int sum_up_to (int n) {
                if (n < 0) return 0;
                int sum = 0;
                for (int = 0; i < n; ++i) {
                    sum +=i;
                }
                return sum;
            }

            int main (int argc, char* argv[]) {
                if (argc < 2) {
                    std::cout << "Usage: " << argv[0] << " <n>\n";
                    return 0;
                }
                int n = std::atoi(argv[1]);
                int sum = sum_up_to(n);
                std:cout << "result: " << sum << '\n';
            }
```

- Compile For Debugging

```
$ g++ -g -o sum sum.cpp
```

- Start The Debugger

```
$ gdb sum
(gdb)
```

- Running Your Program

```
run <args>
```

```
(gdb) run 5
Starting program: /home/.../sum
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/\
libthread_db.so.1".
result: 10
[Inferior 1 (process 6122) exited normally]
```

- Setting Breakpoints

Command	Stop at
break 12	line numer 12 in currently active source code file
break sum_up_to	first executable line of function sum_up_to within ALL source code files
break sum.cpp:7	line number 7 in file sum.cpp (if not in the same directory, use relative/full pathname)
break sum.cpp:main	first executable line of function main in file sum.cpp

- Stepping Through Your Program

```
$ cgdb sum
(gdb) break main
Breakpoint 1 at 0x123b: file debug-01-sum.cpp, line 14.

(gdb) run 5
Starting program: /home/.../sum 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/\
libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffdfb8) at sum.cpp:14
14              if (argc < 2) {

(gdb) next
18              int n = std::atoi(argv[1]);

(gdb) next
19              int sum = sum_up_to(n);

(gdb) step
sum_up_to (n=5) at debug-01-sum.cpp:5
5               if (n < 0) return 0;

... several times next

(gdb) print sum
```

```
        $2 = 1

        ... several times next

        (gdb) next
        result: 10
        21      }
```

- Conditional Breakpoints

```
        break 20 if i == 2000

        break 180 if i < 0

        break test.cpp:34 if (x & y) == 1

        break myfunc if i % (j+3) != 0
```

- Managing Breakpoints

```
        info breakpoints        ..show all breakpoints
        delete                  ..delete all breakpoints
        delete 1                ..delete breakpoint number 1
        clear                   ..delete breakpoint at the next instruction
        disable 2               ..disable breakpoint number 2
        enable 2                ..(re-)enable breakpoint number 2
        save breakpoints <file> ..save breakpoints to file
        source <file>           ..load breakpoints from file
```

- Inspecting and Setting Variables

  - monitor local variables

```
        info locals
```

  - print value of a variable / an expression

```
        print x
        print x + 2
```

  - set value of variable

```
        set x = 12
```

- Useful gdb Commands

```
        <Tab>               ..complete command or function name
        run [<arg>...]      ..run program (with cmd line arg(s)
        break <loc>         ..set breakpoint at beginning of function
                              or at a specific line
        step                ..next instruction, step into function
        next                ..next instruction, step over function
        jump <loc>          ..jump to location (useful for exiting
                              long/endless loops
        continue            ..continue until next breakpoint or end of
                              program
        until <loc>         ..continue until location (function/line)
        finish              ..finish (step out of) current function
        print <expression>  ..print value of expression, e.g. variable
        info breakpoints    ..list all breakpoints
        info locals         ..list local variables and their values
        backtrace           ..show call stack
```

Address Sanitizer

- Address Sanitizer (ASan)

    - g++, clang++
    - detects memory corruption bugs
        - memory leaks
        - access to already freed memory
        - access to incorrect stack areas
        ...
    - instruments your code with additional instructions
        - roughly 70% runtime increase
        - roughly 3-fold increase in memory usage

- Example: Dereferencing Null Pointer

```
$ vi asan.cpp

    #inlcude <iostream>

    int main () {
        int* p = nullptr;
        cout << "p = " << *p << '\n';
    }
```

    - Address Sanitizer in Action

```
$ g++ asan.cpp -o asan -fsanitize=address
$ ./asan
=================================================================
==7498==ERROR: AddressSanitizer: SEGV on unknown address \
0x000000000000 (pc 0x55f255a73315 bp 0x7ffc50ab1d70 sp \
0x7ffc50ab1d60 T0)
==7498==The signal is caused by a READ memory access.
==7498==Hint: address points to the zero page.
    #0 0x55f255a73315 in main (/home/.../asan+0x1315)
    #1 0x7fd6af429d8f in __libc_start_call_main ../sysdeps/nptl/\
    libc_start_call_main.h:58
    #2 0x7fd6af429e3f in __libc_start_main_impl ../csu/\
    libc-start.c:392
    #3 0x55f255a731e4 in _start (/home/.../asan+0x11e4)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (/home/.../asan+0x1315) in main
==7498==ABORTING
```

Undefined Behavior Sanitizer

- Undefined Behavior Sanitizer (UBSAN)

    - clang++,g++
    - detects many types of undefined behavior at runtime
        - dereferencing null pointers
        - reading from misaligned pointers
        - integer overflow
        - division by zero
        - ...
    - instruments your code with additional instructions:
      runtime increase in debug build ~25%

- Example: Signed Integer Overflow

```
$ vi ubsan.cpp

    int main()
    {
        int i = std::numeric_limits<int>::max();
```

```
            i += 1;     // overflow!
            cout << "i = " << i << '\n';
        }

    $ clang++ ubsan.cpp -o ubsan
    $ ./ubsan
    i = -2147483648
```

- UBSAN in Action

```
    $ clang ++ ubsan.cpp -o ubsan -fsanitize=undefined
    $ ./ubsan
    ubsan.cpp:6:4: runtime error: signed integer overflow: 2147483647 + 1 \
    cannot be represented in type 'int'
    SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan.cpp:6:4 \
    in
    i = -2147483648
```

---

valgrind

- Valgrind

  - detects common runtime errors
    - readint/writing feed memory or incorrect stack areas
    - using values before they have been initialized
    - incorrect freeing of memory, such as double freeing
    - mususe of funcitons for memory allocations
    - memory leaks - unintentional memory consumption often related to program logic flaws
      - leading to loss of memory pointers prior to deallocation

```
    valgrind [options] ./program [program options]
```

  - Options

```
        --help
        --tool=memcheck     ..check the memory of your program
        --leak-check=full   ..see the details of leaked memory
        --verbose
```

- Example

```
    $ vi compare.cpp

        #include <iostream>

        bool f (int i) {
            return (i == 5)
        }

        int main () {
            int i;
            if ( f(i) ) {
                cout << "MATCH\n";
            }
        }

    $ g++ -g -o compare compare.cpp
    $ valgrind ./compare
    ==8216== Memcheck, a memory error detector
    ==8216== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward \
             et al.
    ==8216== Using Valgrind-3.18.1 and LibVEX; rerun with -h for \
             copyright info
    ==8216== Command: ./compare
```

```
==8216==
==8216== Conditional jump or move depends on uninitialised value(s)
==8216==    at 0x1091B5: main (compare.cpp:10)
==8216==
==8216==
==8216== HEAP SUMMARY:
==8216==     in use at exit: 0 bytes in 0 blocks
==8216==   total heap usage: 1 allocs, 1 frees, 72,704 bytes
           allocated
==8216==
==8216== All heap blocks were freed -- no leaks are possible
==8216==
==8216== Use --track-origins=yes to see where uninitialised values
         come from
==8216== For lists of detected and suppressed errors, rerun with: -s
==8216== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
         0)
```

- Example

```
$ vi readwrite.cpp

    #include <iostream>
    #include <vector>

    int main () {
        constexpr int dim = 100;
        int i;
        std::vector<float> v(dim);

        for(i = 0; i < dim; ++i)
            v[i] = 0.0;

        float k = 2.0f;
        float sup = k;

        k = v[i];
        v[i] = sup;

        std::cout << "GOOD END\n";
    }

$ g++ readwrite.cpp -g -o readwrite
$ valgrind --tool=memcheck ./readwrite
==8359== Memcheck, a memory error detector
==8359== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward
         et al.
==8359== Using Valgrind-3.18.1 and LibVEX; rerun with -h for
         copyright info
==8359== Command: ./readwrite
==8359==
==8359== Invalid read of size 4
==8359==    at 0x109315: main (readwrite.cpp:15)
==8359==  Address 0x4ddfe10 is 0 bytes after a block of size 400
          alloc'd
==8359==    at 0x4849013: operator new(unsigned long) (in
            /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==8359==    by 0x109BEB: __gnu_cxx::new_allocator<float>::allocate\
            (unsigned long, void const*) (new_allocator.h:127)
==8359==    by 0x109AC7: std::allocator_traits<std::allocator\
            <float> >::allocate(std::allocator<float>&, unsigned
            long) (alloc_traits.h:464)
==8359==    by 0x109A01: std::_Vector_base<float, std::allocator\
            <float> >::_M_allocate(unsigned long) (stl_vector.h:346)
==8359==    by 0x10987C: std::_Vector_base<float, std::allocator\
            <float> >::_M_create_storage(unsigned long) \
            (stl_vector.h:361)
==8359==    by 0x10968E: std::_Vector_base<float, std::allocator\
```

```
                  <float> >::_Vector_base(unsigned long, std::allocator\
                  <float> const&) (stl_vector.h:305)
            ==8359==    by 0x1094E4: std::vector<float, std::allocator<float> \
                  >::vector(unsigned long, std::allocator<float> const&) \
                  (stl_vector.h:511)
            ==8359==    by 0x1092AC: main (readwrite.cpp:7)
            ==8359==
            ==8359== Invalid write of size 4
            ==8359==    at 0x109342: main (readwrite.cpp:16)
            ==8359==  Address 0x4ddfe10 is 0 bytes after a block of size 400 \
                  alloc'd
            ==8359==    at 0x4849013: operator new(unsigned long) (in /usr/\
                  libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
            ==8359==    by 0x109BEB: __gnu_cxx::new_allocator<float>::allocate\
                  (unsigned long, void const*) (new_allocator.h:127)
            ==8359==    by 0x109AC7: std::allocator_traits<std::allocator\
                  <float> >::allocate(std::allocator<float>&, unsigned \
                  long) (alloc_traits.h:464)
            ==8359==    by 0x109A01: std::_Vector_base<float, std::allocator\
                  <float> >::_M_allocate(unsigned long) (stl_vector.h:346)
            ==8359==    by 0x10987C: std::_Vector_base<float, std::allocator\
                  <float> >::_M_create_storage(unsigned long) \
                  (stl_vector.h:361)
            ==8359==    by 0x10968E: std::_Vector_base<float, std::allocator\
                  <float> >::_Vector_base(unsigned long, std::allocator\
                  <float> const&) (stl_vector.h:305)
            ==8359==    by 0x1094E4: std::vector<float, std::allocator<float> \
                  >::vector(unsigned long, std::allocator<float> const&) \
                  (stl_vector.h:511)
            ==8359==    by 0x1092AC: main (readwrite.cpp:7)
            ==8359==
            GOOD END
            ==8359==
            ==8359== HEAP SUMMARY:
            ==8359==     in use at exit: 0 bytes in 0 blocks
            ==8359==   total heap usage: 3 allocs, 3 frees, 74,128 bytes \
                  allocated
            ==8359==
            ==8359== All heap blocks were freed -- no leaks are possible
            ==8359==
            ==8359== For lists of detected and suppressed errors, rerun with: -s
            ==8359== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 \
                  from 0)
```

- Valgrind Tools

```
            --tool=memcheck         ..leaks, invalid reads/writes detection
            --tool=callgrind        ..runtime profiling
            --tool=cachgrind        ..cache profiling
            --tool=massif           ..heap memory profiling
            --tool=helgrind         ..locking order violation detection
            --tool=drd              ..multithreading error detection
```