Programming Expertise Exceptions, Smart Pointers and Working with Libraries

Detlef Groth

2023-07-13

Last week summary

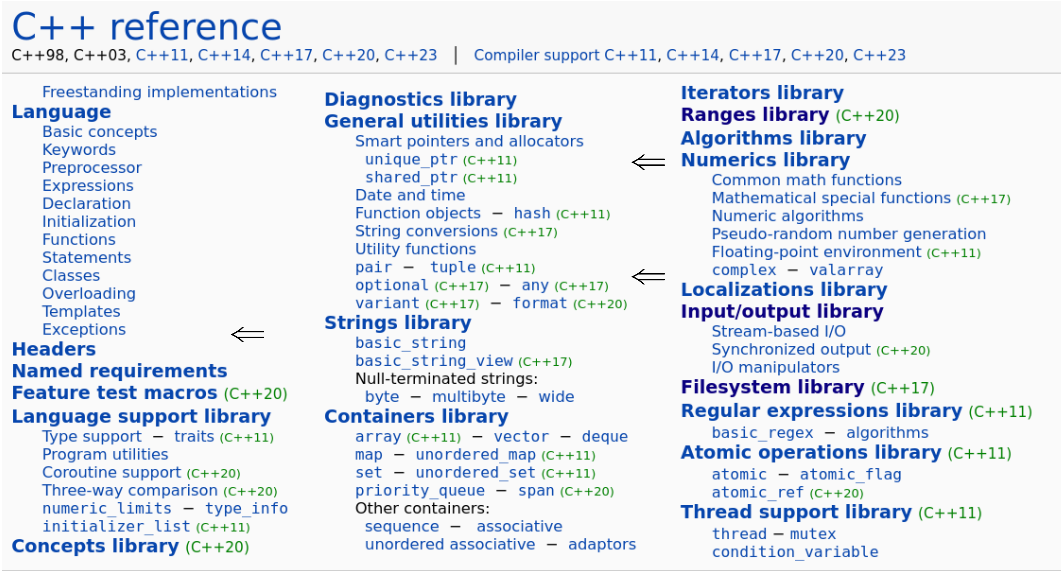
- friend functions
- copy constructors
- this pointer
- namespaces
- templates
- data containers - std::vector and std::map
- slowness of regular expressions
- g++ -O3 compiler flag

g++ -O3 flag

Timings test-regex.cpp:

```
$ g++ test-regex.cpp
$ ./a.out gene_ontology_edit.obo.2020-12-01
lines:      567782  ms:27
find ids:   47345   ms:57
regex ids:  47345   ms:8600
$ g++ -O3 test-regex.cpp
./a.out gene_ontology_edit.obo.2020-12-01
lines:      567782   ms:24
find ids:   47345    ms:50
regex ids:  47345    ms:661
```

Cpp Reference



C++ reference

C++98, C++03, C++11, C++14, C++17, C++20, C++23 | Compiler support C++11, C++14, C++17, C++20, C++23

Freestanding implementations

Language

- Basic concepts
- Keywords
- Preprocessor
- Expressions
- Declaration
- Initialization
- Functions
- Statements
- Classes
- Overloading
- Templates
- Exceptions ⇐

Headers

Named requirements

Feature test macros (C++20)

Language support library

- Type support — traits (C++11)
- Program utilities
- Coroutine support (C++20)
- Three-way comparison (C++20)
- numeric_limits — type_info
- initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

General utilities library

- Smart pointers and allocators
  - unique_ptr (C++11) ⇐
  - shared_ptr (C++11)
- Date and time
- Function objects — hash (C++11)
- String conversions (C++17)
- Utility functions
- pair — tuple (C++11)
- optional (C++17) — any (C++17) ⇐
- variant (C++17) — format (C++20)

Strings library

- basic_string
- basic_string_view (C++17)
- Null-terminated strings:
  - byte — multibyte — wide

Containers library

- array (C++11) — vector — deque
- map — unordered_map (C++11)
- set — unordered_set (C++11)
- priority_queue — span (C++20)
- Other containers:
  - sequence — associative
  - unordered associative — adaptors

Iterators library

Ranges library (C++20)

Algorithms library

Numerics library

- Common math functions
- Mathematical special functions (C++17)
- Numeric algorithms
- Pseudo-random number generation
- Floating-point environment (C++11)
- complex — valarray

Localizations library

Input/output library

- Stream-based I/O
- Synchronized output (C++20)
- I/O manipulators

Filesystem library (C++17)

Regular expressions library (C++11)

- basic_regex — algorithms

Atomic operations library (C++11)

- atomic — atomic_flag
- atomic_ref (C++20)

Thread support library (C++11)

- thread — mutex
- condition_variable

6 Exceptions, Pointers, Testing and Libraries

Outline

Exception:

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

[Tutorialspoint — C++ Exception Handling, 2019]

Exceptions, try, catch, throw

Instead of crashing your app it is better to put critical parts into try - catch blocks. Syntax 1 using manual throw:

```
try {
    if (condition) {
        // something bad happens
       throw "message";
    }
} catch (const char* message) {
    cerr << message;
    exit(1);
    // if you would like to exit the app
}
```

Tutorialspoint:

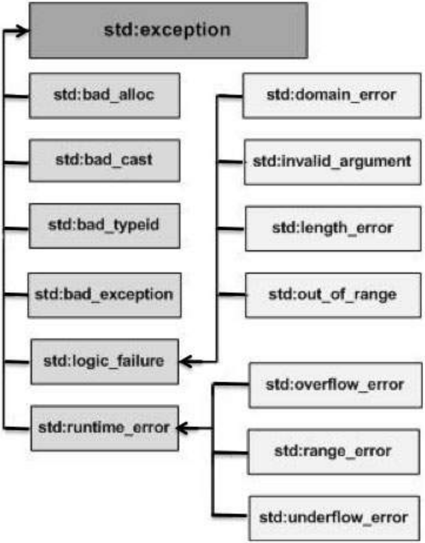
- try – A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- throw – A program throws an exception when a problem shows up. This is done using a throw keyword.
- catch – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

Syntax 2 using generic exception:

```
try {
    // dangerous code
    // more code
    // more dangerous code
} catch (exception const& ex) {
     cerr << "Exception: " << ex.what() <<endl;
    exit(1); // only if you would like to exit the app
}
```

⇒ don't use exceptions if simple checks are available
⇒ like check if a file exists before you try to open it

Cpp-Exceptions



[Tutorialspoint — C++ Exception Handling, 2019]

defining your own exceptions

```
#include <iostream>
#include <exception>
using namespace std;

struct MyException : public exception {
   const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

```
$ g++   -o exception2.cpp.bin -std=c++17 -fconcepts exception2.cpp &&
./exception2.cpp.bin
MyException caught
C++ Exception
```

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way.

[Tutorialspoint — C++ Exception Handling, 2019]

Last years IMatrix project

The overloaded () operator [1]:

```
class IMatrix {
   ...
   // Operator overloading currently only by index
   // getter using int: cout << mt(1,2)
   // weak check against wrong index
   int  operator() (int rid, int cid) const {
     if (rid-1 > nrow() || cid-1 > nrow()) {
       // not good write a message which might be not seen
        std::cout << "error out of range" << std::endl;
          return(0);
      }
      return mt[rid][cid];
   }
```

[1] Adding const allows to compiler to have two methods with the same parameter list.

```
  // setter using address operator:
  // allows us to change the values in the matrix: mt(1,2)=3
  // better check using exception
  int& operator() (int rid, int cid) {
    try {
      if (rid-1 > nrow() || cid-1 > ncol()) {
         throw "invalid matrix indices" ;
      }
      return mt[rid][cid];
    } catch (const char * msg) {
      std::cerr << "Out of Range error: " <<msg<<'\n';
     }
      // problem must return reference
      return mt[0][0];
   }
}
```

std::optional as alternative to exceptions

Assume you are in a GUI you don't like to crash your program with an exception of the function. It might be feasible sometimes in this case just return nothing from a function, if the operation was not sucessful.

Do do this in C++17 we have the `std::optional`.

```
#include <iostream>
#include <string>
#include <optional>
// optional can be used as the return type that may fail
std::optional<std::string> create(bool b) {
    if (b)
        return "Godzilla";
    return {};
}
```

```
int main () {
    std::cout << "create(false) returned "
              << create(false).value_or("empty") << '\n';
    std::cout << "create(true) returned "
              << create(true).value_or("empty") << '\n';
    if (create(false)) {
        std::cout << "it returns something\n";
    } else {
        std::cout << "it returns nothing\n";
    }
}
```

```
$ g++   -o optional.cpp.bin -std=c++17 -fconcepts optional.cpp &&
./optional.cpp.bin
```

```
create(false) returned empty
create(true) returned Godzilla
it returns nothing
```

Backports for C++17 features

For older compilers you might use some backports by programmers like Martin Moene. He backported many new features for older compilers, also the optional type. See https://github.com/martinmoene/optional-lite

```
#include <string>
#include <iostream>
#include "include/nonstd/optional.hpp"
using nonstd::optional;
// optional can be used as the return type that may fail
optional<std::string> create(bool b) {
    if (b)
        return "Godzilla";
    return {};
}
```

```
int main () {
    std::cout << "create(false) returned "
              << create(false).value_or("empty") << '\n';
    std::cout << "create(true) returned "
              << create(true).value_or("empty") << '\n';
    if (create(false)) {
        std::cout << "Was ok, it returns something!\n";
    } else {
        std::cout << "Was a fail, it returns nothing!!\n";
    }
}
```

```
$ g++  -o optional.cpp.bin -std=c++17 -fconcepts optional.cpp &&
./optional.cpp.bin
```

```
create(false) returned empty
create(true) returned Godzilla
Was a fail, it returns nothing!!
```

Dynamic Memory

Stack and Heap:

Memory in your C++ program is divided into two parts:
The stack: variables declared inside the function will take up memory from the stack.
The heap: This is unused memory of the program and can be used to allocate the memory dynamically when program runs.
Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

[Tutorialspoint — C++ Dynamic Memory, 2019]

Stack vs Heap

You can use the stack if you know exactly how much data you need to allocate before compile time and it is not too big. You can use heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

`http://net-informations.com/faq/net/stack-heap.htm`

New operator:

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.

[Tutorialspoint — C++ Dynamic Memory, 2019]

Delete operator:

If you are not in need of dynamically allocated memory anymore, you can (should!!) use delete operator, which deallocates memory that was previously allocated by new operator.

[Tutorialspoint — C++ Dynamic Memory, 2019]

⇒ Attention since C++11 you should use smart pointers instead of `new` and `delete`!

Avoid Memory leaks!

You must release Heap memory manually if using the new operator with the delete operator!!

```
int foo() {
  char *pBuffer; // nothing allocated yet, pointer is on stack
  bool b = true; // allocated on the stack
  if(b) {
    //Create 500 bytes on the stack
    char buffer[500];

    //Create 500 bytes on the heap
    pBuffer = new char[500];

   } //<-- buffer is deallocated here, pBuffer is not
} //<--- oops there's a memory leak,
  // should have called delete[] pBuffer before;
```

⇒ Smart pointers with C++11 avoid memory leaks
`std::unique_ptr<type>` which does cleanup automatically

⇒ `https://www.fluentcpp.com/2017/08/22/smart-developers-use-smart-pointers-smart-pointers-basics/`

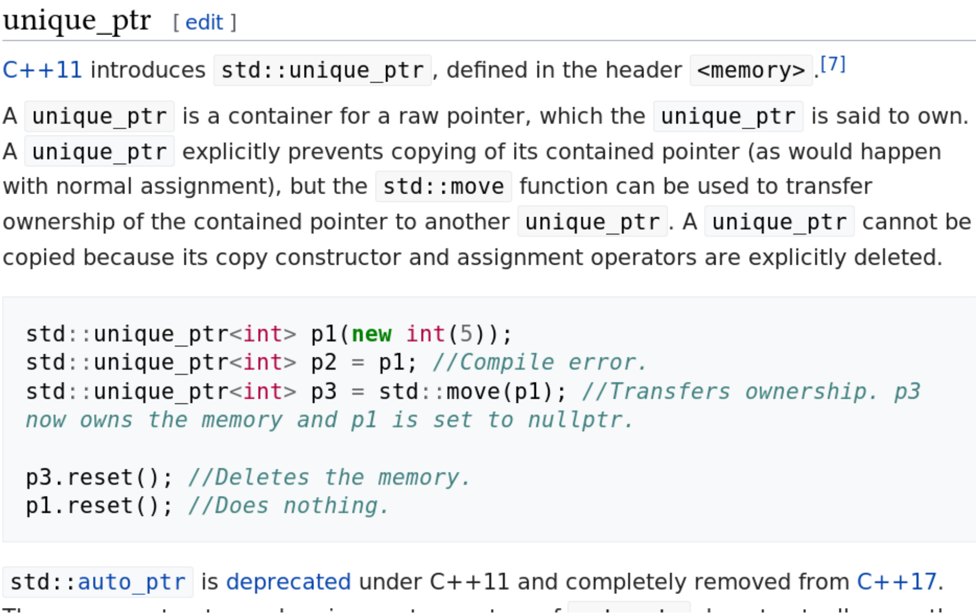
⇒ But even better it is to use is `std::string` or `std::vector` which as well does cleanup automatically. So use the containers of the STL.

Smart pointers

Std::unique ptr:

unique_ptr is one of the Smart pointer implementation provided by c++11 to prevent memory leaks. A unique_ptr object wraps around a raw pointer and its responsible for its lifetime. When this object is destructed then in its destructor it deletes the associated raw pointer. unique_ptr has its -> and * operator overloaded, so it can be used similar to normal pointer.

[thispointer.com: C++11 Smart Pointer – Part 6: unique_ptr, 2019]



unique_ptr [edit]

C++11 introduces `std::unique_ptr`, defined in the header `<memory>`.[7]

A `unique_ptr` is a container for a raw pointer, which the `unique_ptr` is said to own. A `unique_ptr` explicitly prevents copying of its contained pointer (as would happen with normal assignment), but the `std::move` function can be used to transfer ownership of the contained pointer to another `unique_ptr`. A `unique_ptr` cannot be copied because its copy constructor and assignment operators are explicitly deleted.

```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; //Compile error.
std::unique_ptr<int> p3 = std::move(p1); //Transfers ownership. p3
now owns the memory and p1 is set to nullptr.

p3.reset(); //Deletes the memory.
p1.reset(); //Does nothing.
```

`std::auto_ptr` is deprecated under C++11 and completely removed from C++17.

Std::shared ptr:

shared_ptr is a kind of Smart Pointer class provided by c++11, that is smart enough to automatically delete the associated pointer when its not used anywhere. Thus helps us to completely remove the problem of memory leaks and dangling Pointers.
It follows the concept of Shared Ownership i.e. different shared_ptr objects can be associated with same pointer and internally uses the reference counting mechanism to achieve this.

[thispointer.com: C++11 Smart Pointer – Part 1: shared_ptr, 2019]

Pointers on the heap

- `make_shared` - C11++ (prefered way to create)
- `make_unique` - C14++ (prefered way to create) [2]

With a modern cpp compiler there is no reason anymore to use:

```
Dog * fido = new Dog();
delete fido ;
```

Instead you can use:

```
#include <memory>
// complete - but see auto
std::unique_ptr<Dog> fido = std::make_unique<Dog>("Fido");
// shorter - snippet: mku=std::make_unique<%cursor%>();
auto elsa = std::make_unique<Dog>("Elsa");
```

[2] Because `shared_ptr` already has an analogous `std::make_shared`, for consistency we'll call this one `make_unique`. (That C++11 doesn't include `make_unique` is partly an oversight, and it will almost certainly be added in the future. In the meantime, use the one provided below. – Herb Sutter

Dogs on the Heap

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;

class Dog {
    public:
     Dog(std::string name)  {itsName=name;}
     ~Dog() { cout << itsName << " is destroyed!\n"; }
     void bark () { cout << itsName << " says wuff!\n" ; }
    private:
     std::string itsName = "";
};
```

```cpp
int main() {
    // C++98 way
    Dog * fido = new Dog("Fido I.");
    fido->bark();
    delete fido ;
    Dog * fido2 = new Dog("Fido II.");
    fido2->bark();
    // missing fido2 delete here
    // C++11/C++14 way
    auto elsa = make_unique<Dog>("Elsa"); // snippet: mku
    // pointer syntax
    elsa->bark();
    // delete for Elsa not required as
    // Elsa will be automatically destroyed (painless!)!
   return 0;
}
```

```
$ g++  -o dogheap.cpp.bin -std=c++17 -fconcepts dogheap.cpp &&
```

./dogheap.cpp.bin

```
Fido I. says wuff!
Fido I. is destroyed!
Fido II. says wuff!
Elsa says wuff!
Elsa is destroyed!
```

⇒ Fido I was destroyed manually using delete.

⇒ Fido II is not destroyed (no delete), so we have a memory leak :(

⇒ Elsa was destroyed automatically, no memory leak :)

⇒ `make_unique` teaches users “never say new/delete and new[]/delete[]” without disclaimers.

Stack Overflow

To create pointer to arrays of integers on the heap the following would be better:

```
#include <memory>
auto uPtr = std::make_unique<int>(100);
// set a value as usually
uPtr[0]=1
```

The uPtr will have automatic storage duration and will call the destructor when it goes out of scope. The int will have dynamic storage duration (heap) and will be deleted by the smart pointer.
One could generally avoid using new and delete and avoid using raw pointers. With make_unique and make_shared, new isn't required.
`https://stackoverflow.com/questions/42910711/unique-ptr-heap-and-stack-allocation`
⇒ When not to use `std::make_unique`: Don't use it if you need a custom deleter or are adopting a raw pointer from elsewhere (two times deletion - core dump!).

Summary Smart Pointers

- !prefere containers like vector or map!
- !prefere (const) references in function arguments!
- use smart pointers instead of new and delete
- use new and delete only if you deal with foreign raw pointers or if you create your own container
- `https://www.modernescpp.com/index.php/c-core-guidelines-rules-to-smart-pointers`:
- R.20: Use unique_ptr or shared_ptr to represent ownership
- R.21: Prefer unique_ptr over shared_ptr unless you need to share ownership
- R.22: Use make_shared() to make shared_ptrs
- R.23: Use make_unique() to make unique_ptrs
- make snippets

```
#include <memory>
auto pUP = std::make_unique<std::string>("Hello, world!");
std::unique_ptr<int[]> ptr1{ new int[5]{1,2,3,4,5} };
auto ptr2 = std::make_unique<std::array<int, 5>>(
    std::array<int, 5>{1, 2, 3, 4, 5});
auto pSP1 = std::make_shared<std::string>("Hello, world!");
std::shared_ptr<std::string> pSP2 = pSP1;
std::cout << *pUP << std::endl;
```

Exam: don't make you life tricky, avoid pointers. In all the tutorials about smart pointers I saw until no teaching example where the use of a smart pointer was really useful. May be with the fltk-widgets tutorial yes.
Deal with pointers if you run into performace issues:
`https://www.bfilipek.com/2014/05/vector-of-objects-vs-vector-of-pointers.html`
Example: Sorting of a vector of objects is slower than sorting of a vector of smart pointers to those objects.

Scott Meyers ...

... has this to say in Effective Modern C++:

The existence of std::unique_ptr for arrays should be of only intellectual interest to you, because std::array, std::vector, std::string are virtually always better data structure choices than raw arrays. About the only situation I can conceive of when a std::unique_ptr<T[]> would make sense would be when you're using a C-like API that returns a raw pointer to a heap array that you assume ownership of.

`https://www.oreilly.com/library/view/effective-modern-c/9781491908419/ch04.html`

Unit testing

```
int main () {
    IMatrix mt(true);
    IMatrix mt2(3,3,false);
    mt2.set(0,0,5);
    mt2.set(1,0,4); // expected result ??
    mt2.asText();
    std::vector<int> newrow = {7,8,9};
    mt2.addRow(newrow);
    std::vector<int> newcol = {7,8,9,10};
    ...
    // another 300 lines
    ...
    cout << "end" << endl;
}
```

main mess conclusions ...

- During the development of the `IMatrix` class or the dutils namespace a lot of test code accumulates in `main`.
- It would be better to have a separated test approach instead.
- This approach ist called Unit-Testing.
- In its extreme form you write tests first, only thereafter start coding. ⇒ *Extreme Programming*.
- In comparison to other PL, C++ has no default Unit-Test library.
- There is `cppunit` designed after the Java library `JUnit`.
- There are many alternatives in C++.
- Unit testing ensures code integrity during development.
- In larger companies whole departments are responsible for writing the tests.

Unit-Testing frameworks in C++

⇒ 2010: `http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle`

- CppUnit
- Boost.Test
- CppUnitLite
- NanoCppUnit
- Unit++
- CxxTest
- Catch2: `https://github.com/catchorg/Catch2`
- single file test systems
  - utest: `https://github.com/sheredom/utest.h`
  - doctest: `https://github.com/onqtam/doctest`
  - lest: `https://github.com/martinmoene/lest`

Doctest

`https://github.com/onqtam/doctest`

Claim:
The fastest feature-rich C++11/14/17/20 single-header testing framework for unit tests ...

```
// This tells doctest to provide a main()
// only do this in one cpp file
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "include/doctest.h"

// function should live in own file
int factorial(int number) {
   return number <= 1 ? number :
    factorial(number - 1) * number;
}
TEST_CASE("testing the factorial function") {
    CHECK(factorial(1) == 1);
    CHECK(factorial(2) == 2);
    CHECK(factorial(3) == 6);
    CHECK(factorial(10) == 3628800);
    CHECK( factorial(3) == 7 ); // should fail
}
$ g++   -o doctest.cpp.bin -std=c++17 -fconcepts doctest.cpp &&
./doctest.cpp.bin
[doctest] doctest version is "2.4.9"
[doctest] run with "--help" for options
==================================================================
doctest.cpp:11:
TEST CASE:  testing the factorial function

doctest.cpp:16: ERROR: CHECK( factorial(3) == 7 ) is NOT correct!
  values: CHECK( 6 == 7 )

==================================================================
[doctest] test cases: 1 | 0 passed | 1 failed | 0 skipped
[doctest] assertions: 5 | 4 passed | 1 failed |
[doctest] Status: FAILURE!
```

⇒ we could/should move the IMatrix code in main to such a test ...
⇒ there might be better Unit-Test tools than others
⇒ using either of them is better than using neither of them ...
⇒ **Test your code!**

> *Test-driven development (TDD) is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases. This is opposed to software being developed first and test cases created later.*
>
> `https://en.wikipedia.org/wiki/Test-driven_development`

Libraries

- Inspirations: `https://github.com/fffaraz/awesome-cpp`
- Statistics - Armadillo: `http://arma.sourceforge.net/`
- Command line: `https://github.com/p-ranav/argparse`
- Unit Test: `https://github.com/martinmoene/lest`
  `https://github.com/onqtam/doctest`
- CSV files `https://github.com/vincentlaucsb/csv-parser`
  `https://github.com/p-ranav/csv2`
- Ini files `https://github.com/Qix-/tortellini`
- SQLite databases
  `https://github.com/SqliteModernCpp/sqlite_modern_cpp`
- XLSX files `https://github.com/tfussell/xlnt`
- GUI
  - GTK, QT – heavy but complete but GPL
  - wxWidgets – average LGPL
  - Fltk – lightweight and LGPL

Some single file libraries

CLI11 - access to commandline options and help tools:
https://github.com/CLIUtils/CLI11

```
#include <iostream>
#include <string>
#include "include/CLI11.hpp"
int main(int argc, char** argv) {
    CLI::App app{"App description"};
    std::string filename = "default";
    app.add_option("-f,--file", filename, "A help string");
    // app.add_option("-f,--file", filename,
    //   "A help string")->required();
    CLI11_PARSE(app, argc, argv);
    std::cout << "Done!\n";
    return 0;
}
```

```
$ g++  -o cli11.cpp.bin -std=c++17 -fconcepts cli11.cpp && ./cli11.cpp.bin
Done!
```

```
[groth@bariuke build]$ ./cli11.cpp.bin --help
App description
Usage: ./cli11.cpp.bin [OPTIONS]

Options:
  -h,--help                  Print this help message and exit
  -f,--file TEXT             A help string
```

⇒ executables are quite large (750k)

argparse - Argument Parser

https://github.com/p-ranav/argparse

```
#include <utility>
#include "include/argparse.hpp"
int main(int argc, char *argv[]) {
  argparse::ArgumentParser program("program name");
  program.add_argument("square")
    .help("display the square of a given integer")
    .action([](const std::string& value) {
        return std::stoi(value); });
  try {
    program.parse_args(argc, argv);
  }
  catch (const std::runtime_error& err) {
    std::cout << err.what() << std::endl;
    std::cout << program;
    exit(0);
  }
  auto input = program.get<int>("square");
  std::cout << (input * input) << std::endl;
  return 0;
}
```

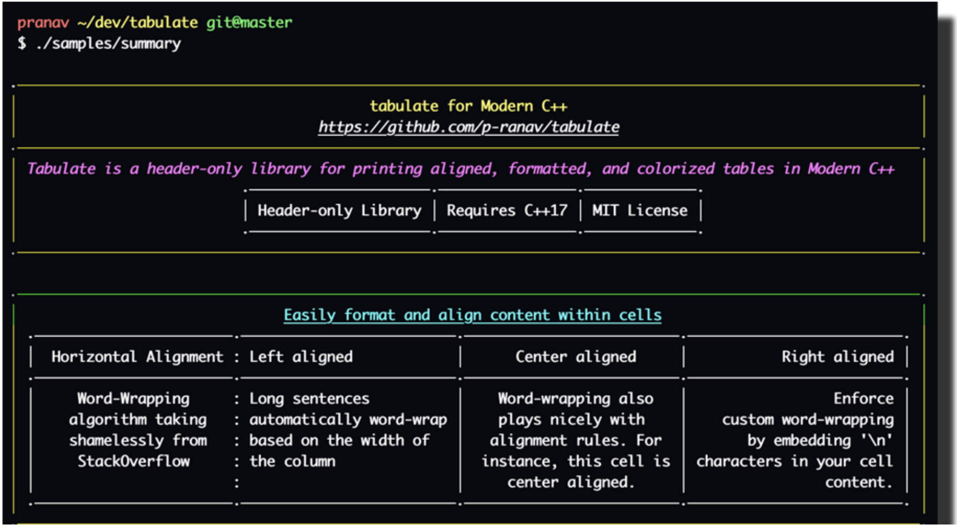
```
$ g++   -o argparse.cpp.bin -std=c++17 -fconcepts argparse.cpp &&
./argparse.cpp.bin
: expected 1 argument(s). 0 provided.
Usage: program name [options] square

Positional arguments:
square                     display the square of a given integer

Optional arguments:
-h --help                  shows help message and exits
-v --version               prints version information and exits
```

⇒ executables are smaller around 250kb

tabulate

github: p-ranav

build passing | C++ 11 | License MIT | version 1.4



```
pranav ~/dev/tabulate git@master
$ ./samples/summary

.------------------------------------------------------------------------------------------.
|                                 tabulate for Modern C++                                    |
|                            https://github.com/p-ranav/tabulate                             |
.------------------------------------------------------------------------------------------.
| Tabulate is a header-only library for printing aligned, formatted, and colorized tables in Modern C++ |
|                  .----------------------.-----------------.-------------.                  |
|                  | Header-only Library  | Requires C++17  | MIT License |                  |
|                  .----------------------.-----------------.-------------.                  |
.------------------------------------------------------------------------------------------.


.------------------------------------------------------------------------------------------.
|                       Easily format and align content within cells                         |
| .--------------------.----------------------.--------------------.-----------------------. |
| | Horizontal Alignment : Left aligned       |   Center aligned   |         Right aligned | |
| .--------------------.----------------------.--------------------.-----------------------. |
| |   Word-Wrapping    : Long sentences       | Word-wrapping also |               Enforce | |
| |  algorithm taking  : automatically word-wrap|  plays nicely with |  custom word-wrapping | |
| |  shamelessly from  : based on the width of | alignment rules. For |   by embedding '\n' | |
| |   StackOverflow    : the column           | instance, this cell is | characters in your cell | |
| |                    :                      |  center aligned.   |              content. | |
| .--------------------.----------------------.--------------------.-----------------------. |
```

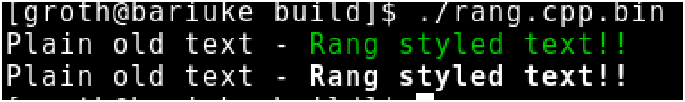
Rang - Colored Terminal

Rang: A Minimal, Header only Modern C++ library for terminal goodies

https://github.com/agauniyal/rang

```
#include "include/rang.hpp"
using namespace std;
using namespace rang;

int main() {
    cout << "Plain old text - "
         << fg::green << "Rang styled text!!"
         << fg::reset << endl;
    cout << "Plain old text - "
         << style::bold << "Rang styled text!!"
         << style::reset << endl;
}
```

```
$ g++   -o rang.cpp.bin -std=c++17 -fconcepts rang.cpp && ./rang.cpp.bin
Plain old text - Rang styled text!!
Plain old text - Rang styled text!!
```



```
[groth@bariuke build]$ ./rang.cpp.bin
Plain old text - Rang styled text!!
Plain old text - Rang styled text!!
```

⇒ executable around 27kb

C++20 std::format for C++17 compilers

Installation on Fedora: `sudo dnf install fmt fmt-devel`

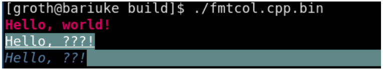
```cpp
#include <iostream>
#include <fmt/format.h>
#include <fmt/printf.h>
using namespace std;
int main () {
    fmt::print("{} {}!\n","Hello","World");
    fmt::printf("Hello %s!\n","World");
    cout << fmt::format("The answer is {}!\n", 42);
    return(0);
}
```

```
$ g++ -lfmt -o ftm.cpp.bin -std=c++17 -fconcepts ftm.cpp && ./ftm.cpp.bin
Hello World!
Hello World!
The answer is 42!
```

⇒ about 200-400kb size of the executable.

fmt has as well terminal colors (but 300/180kb size)

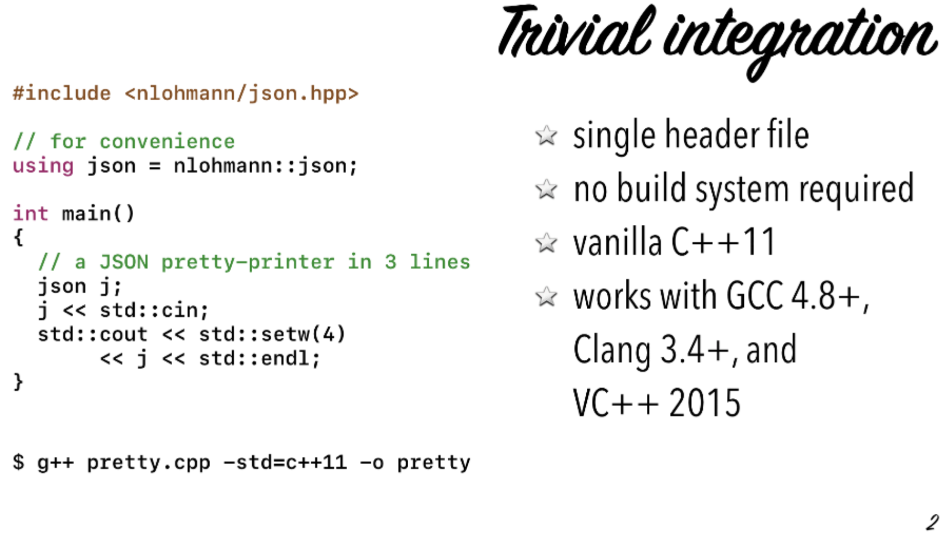
```
#include <fmt/color.h>

int main() {
  fmt::print(fg(fmt::color::crimson)
                | fmt::emphasis::bold,
             "Hello, {}!\n", "world");
  fmt::print(fg(fmt::color::floral_white)
    | bg(fmt::color::slate_gray) |
             fmt::emphasis::underline, "Hello, {}!\n", "???");
  fmt::print(fg(fmt::color::steel_blue) | fmt::emphasis::italic,
             "Hello, {}!\n", "??");
}
```



```
[groth@bariuke build]$ ./fmtcol.cpp.bin
Hello, world!
Hello, ???!
Hello, ??!
```

JSON for Modern C++

https://github.com/nlohmann/json



Trivial integration

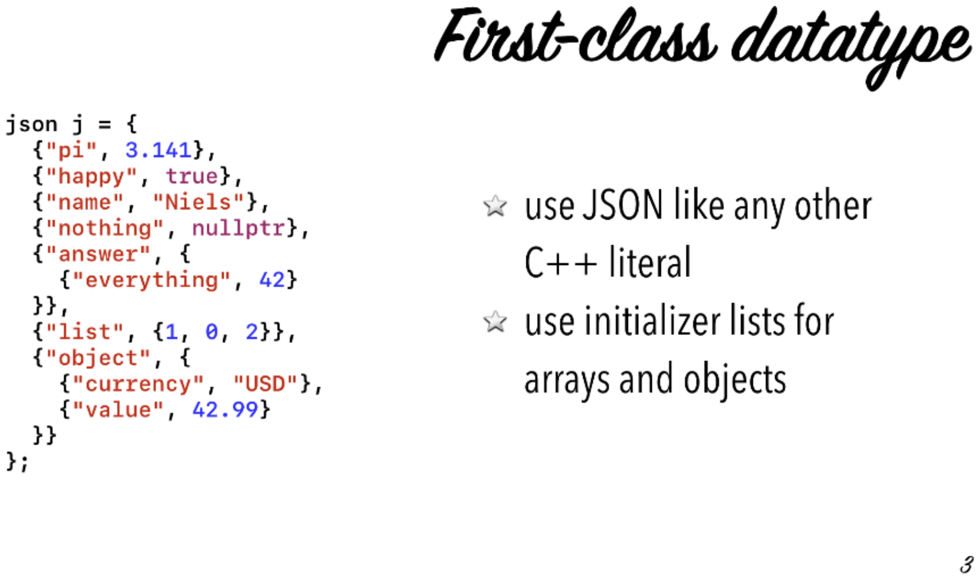
```cpp
#include <nlohmann/json.hpp>

// for convenience
using json = nlohmann::json;

int main()
{
  // a JSON pretty-printer in 3 lines
  json j;
  j << std::cin;
  std::cout << std::setw(4)
       << j << std::endl;
}
```

- single header file
- no build system required
- vanilla C++11
- works with GCC 4.8+, Clang 3.4+, and VC++ 2015

```
$ g++ pretty.cpp -std=c++11 -o pretty
```

2

https://github.com/nlohmann/json



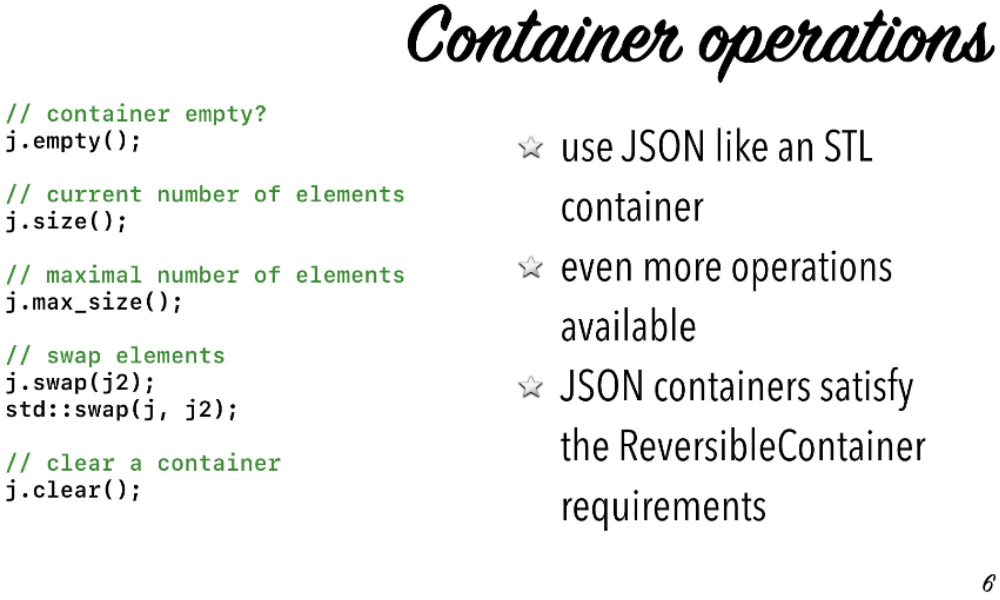
First-class datatype

```
json j = {
  {"pi", 3.141},
  {"happy", true},
  {"name", "Niels"},
  {"nothing", nullptr},
  {"answer", {
    {"everything", 42}
  }},
  {"list", {1, 0, 2}},
  {"object", {
    {"currency", "USD"},
    {"value", 42.99}
  }}
};
```

- use JSON like any other C++ literal
- use initializer lists for arrays and objects

3

https://github.com/nlohmann/json



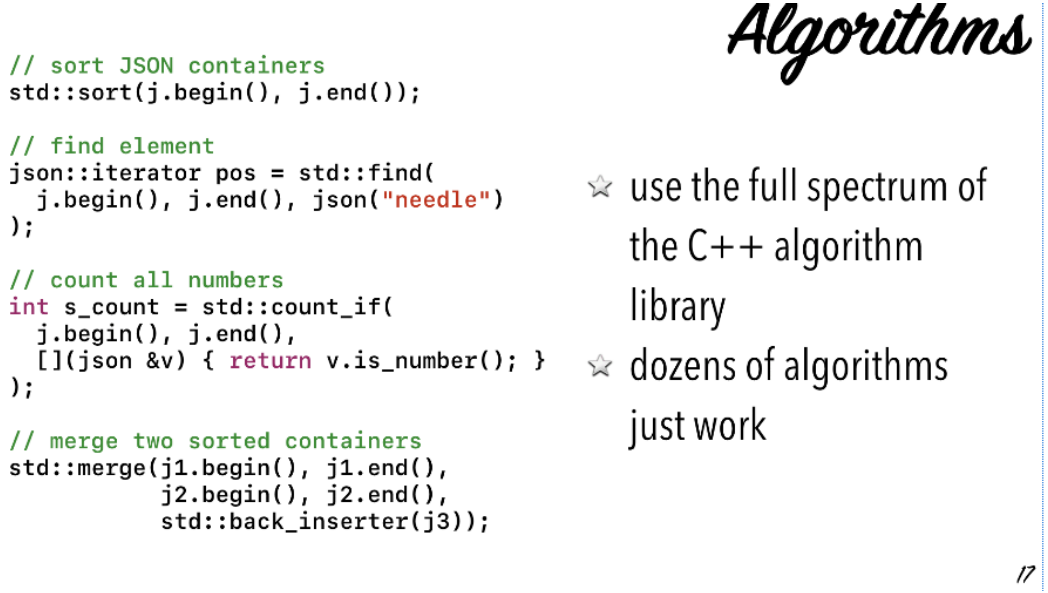
Container operations

```
// container empty?
j.empty();

// current number of elements
j.size();

// maximal number of elements
j.max_size();

// swap elements
j.swap(j2);
std::swap(j, j2);

// clear a container
j.clear();
```

- ☆ use JSON like an STL container
- ☆ even more operations available
- ☆ JSON containers satisfy the ReversibleContainer requirements

6

https://github.com/nlohmann/json



Algorithms

```cpp
// sort JSON containers
std::sort(j.begin(), j.end());

// find element
json::iterator pos = std::find(
  j.begin(), j.end(), json("needle")
);

// count all numbers
int s_count = std::count_if(
  j.begin(), j.end(),
  [](json &v) { return v.is_number(); }
);

// merge two sorted containers
std::merge(j1.begin(), j1.end(),
           j2.begin(), j2.end(),
           std::back_inserter(j3));
```

- ☆ use the full spectrum of the C++ algorithm library
- ☆ dozens of algorithms just work

17

Json Example

Install:

```
wget https://raw.githubusercontent.com/nlohmann/json/develop/
    single_include/nlohmann/json.hpp
mv json.hpp include/
```

```
#include <iostream>
#include "include/json.hpp"
using json = nlohmann::json;
int main () {
    // create JSON object
    json object = {
        {"the good", "il buono"},
        {"the bad", "il cattivo"},
        {"the ugly", "il brutto"}
    };
    // output element with key "the ugly"
    std::cout << object.at("the ugly") << '\n';

    // change element with key "the bad"
    object.at("the bad") = "il cattivo";

    // output changed array
    std::cout << object << '\n';
    for (auto& [key,value]: object.items()) {
        std::cout << "key: " << key << " - value: " <<
            value << std::endl;
    }
    return(0);

}
```

```
$ g++  -o json.cpp.bin -std=c++17 -fconcepts json.cpp && ./json.cpp.bin
```

```
"il brutto"
{"the bad":"il cattivo","the good":"il buono","the ugly":"il brutt
key: the bad - value: "il cattivo"
key: the good - value: "il buono"
key: the ugly - value: "il brutto"
```

⇒ about 200-400kb size of the executable.

Package installations

- header only libraries (just download and place the file)
- few file libraries (just download and place the files)
- more complex libraries:
  - use OS package manager to install (dnf, apt, brew, pacman, winget(?))
  - use C++ package manager (cget, conan, vcpkg)
  - download and build yourself (sometimes tricky, last choice)

Linux package installs

Fedora: dnf install pkgname

Ubuntu/Debian: apt-get install pkgname

Installations:

```
sudo dnf install fmt fmt-devel
sudo dnf install fltk fltk-devel
```

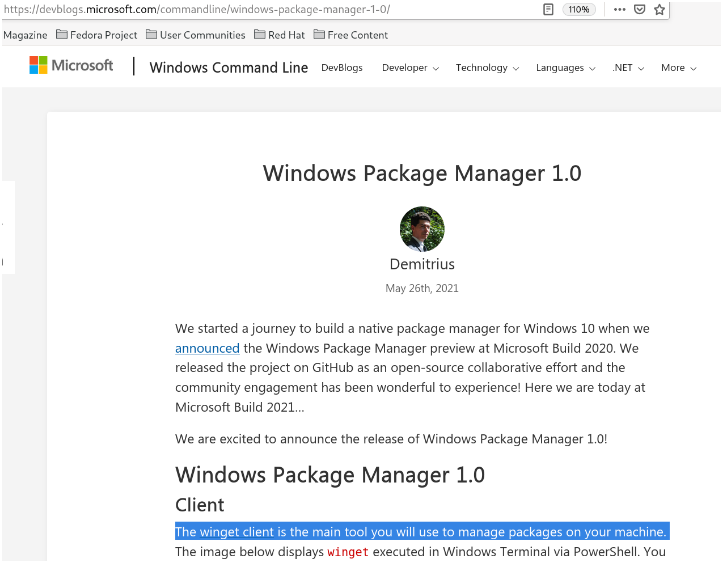
Windows MSYS64 packages

MSYS2 is a software distro and building platform for Windows.
`https://www.msys2.org/`
Packages: `https://packages.msys2.org`
Installation examples:

```
pacman -S mingw-w64-x86_64-fmt
pacman -S mingw-w64-x86_64-nlohmann-json
pacman -S mingw-w64-x86_64-fltk
```

⇒ should work for clang and gcc compilers.

May 2021 - Windows 10: winget



Windows Package Manager 1.0

Demitrius

May 26th, 2021

We started a journey to build a native package manager for Windows 10 when we announced the Windows Package Manager preview at Microsoft Build 2020. We released the project on GitHub as an open-source collaborative effort and the community engagement has been wonderful to experience! Here we are today at Microsoft Build 2021...

We are excited to announce the release of Windows Package Manager 1.0!

Windows Package Manager 1.0

Client

The winget client is the main tool you will use to manage packages on your machine. The image below displays `winget` executed in Windows Terminal via PowerShell. You

Mac OSX: homebrew packages

The Missing Package Manager for macOS ...

`https://brew.sh/`

Installation examples (Untested as I don't have an OSX system):

```
# install homebrew
bash -c "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/
   master/install.sh)"
# install packages
brew install fmt
brew install nlohmann-json
brew install fltk
```

C++ package manager

They are all crossplatform:

- conan: `https://conan.io/` probably the most used one
- vcpkg: `https://docs.microsoft.com/en-us/cpp/build/vcpkg`
- cget: `https://github.com/pfultz2/cget`
- ...

They thereafter work similar to the OS package manager you can try first to install them by the OS package manager.

Example: Installing conan on Msys64 / Windows:

```
# as pip3 was not there
python3 -m ensurepip --default-pip
pip3 install conan --user
```

Install packages from source

Single header files:

- just download and place the header file into your souce directory
- use `#include "path/to/header.hh"` to use it

Larger packages:

- fetch and unpack the source
- switch into source directory
- run configure or cmake or make
- run make
- run make install

Example with fltk sources

```
wget https://www.fltk.org/pub/fltk/1.3.5/fltk-1.3.5-source.tar.gz
tar xfvz fltk-1.3.5-source.tar.gz
cd fltk-1.3.5/
 ./configure --prefix=/home/groth/.local
make
make install
```

In your code you then can include the header files of the library. Probably the compile command must be adapted to use the library. Have a look at the library documentation.
BTW: The fltk-library comes with `fltk-config` which can be used to simplify compilation. You then use:

```
fltk-config --compile filename.cpp
```

This creates an application filename using the right includes and library arguments for the compiler.

GUI Libraries

- there are some mainstream libs like GTK and Qt
- quite heavyweight if you just like to wrap a small tool
- you then deliver several files often weighting some dozen Mb
- alternatively you can use a more lightweight alternative to wrap an existing applications
- example for this: `https://www.fltk.org`
- applications will be statically linked and should be always less than 1 MB
- fltk did not encourages dynamic linking (so/dll) as this creates at some time dependency problems
- library is that small, that it can be statically linked
- install is than just a copy operation of the executable

Fltk - Overview

- crossplatform: Windows, OSX, Unix (X-Windows)
- small libraries, statically linked in
- LGPL license in principle can be used commercially without given own sources
- only library additions must be published
- website: `https://www.fltk.org/`
- cheats: `http://seriss.com/people/erco/fltk/`
- even spreadsheet: `http://seriss.com/people/erco/fltk/#Fl_Table_Spreadsheet_Cell_Row_Col_Edit`

Example Hello World

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
int main(int argc, char **argv) {
  Fl_Window *window = new Fl_Window(340,180);
  Fl_Box *box = new Fl_Box(20,40,300,100,"Hello, World!");
  box->box(FL_UP_BOX);
  box->labelfont(FL_BOLD+FL_ITALIC);
  box->labelsize(36);
  box->labeltype(FL_SHADOW_LABEL);
  window->end();
  window->show(argc, argv);
  return Fl::run();
}
```

⇒ fatal error: FL/Fl.H: No such file or directory

Installing fltk

If you have sudo rights on Fedora:

```
sudo dnf install fltk-devel fltk-static fltk
```

On Ubuntu/Debian try:

```
sudo apt install fltk1.3-dev fltk
```

On Homebrew / Mac-OSX:

```
brew install fltk
```

On Windows / Msys2-64:

```
pacman -S mingw-w64-x86_64-fltk
```

On Windows / vcpkg:

`https://devblogs.microsoft.com/cppblog/`

`vcpkg-a-tool-to-acquire-and-build-c-open-source-libraries-on-windows/`

On Windows with Visual Studio:

`https://bumpyroadtocode.com/2017/08/29/`

`how-to-install-and-use-fltk-1-3-4-in-visual-studio-2017-complete-guid`

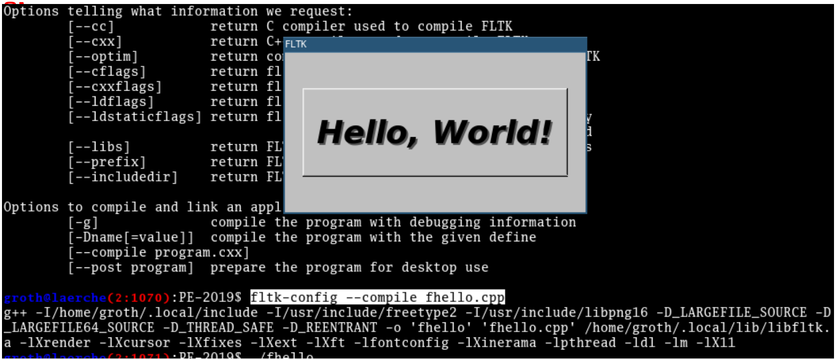
Source code user install:

```
wget https://www.fltk.org/pub/fltk/1.3.5/fltk-1.3.5-source.tar.gz
tar xfvz fltk-1.3.5-source.tar.gz
cd fltk-1.3.5/
./configure --prefix=/home/groth/.local
make
make install
...
=== installing FL ===
Installing include files in /home/groth/.local/include...
=== installing src ===
Installing libraries in /home/groth/.local/lib...
=== installing fluid ===
Installing FLUID in /home/groth/.local/bin...
=== installing test ===
Installing example programs to /home/groth/.local/share/doc/fltk/exampl
=== installing documentation ===
Installing documentation files in /home/groth/.local/share/doc/fltk ..
Installing man pages in /home/groth/.local/share/man ...
```

using fltk-config and smart pointers

```
#include <iostream>
#include <memory>
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
int main(int argc, char **argv) {
  auto window = std::make_unique<Fl_Window>(340,180);
  auto box = std::make_unique<Fl_Box>(20,40,300,100,
    "Hello, World!");
  box->box(FL_UP_BOX);
  box->labelfont(FL_BOLD+FL_ITALIC);
  box->labelsize(36);
  box->labeltype(FL_SHADOW_LABEL);
  window->end();
  window->show(argc, argv);
  std::cout << "fltk working!" << std::endl;
  return Fl::run();
}
```

$ fltk-config –compile fhello3.cpp



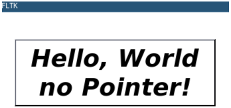
```
Options telling what information we request:
        [--cc]              return C compiler used to compile FLTK
        [--cxx]             return C+
        [--optim]           return co                                    TK
        [--cflags]          return fl
        [--cxxflags]        return fl
        [--ldflags]         return fl
        [--ldstaticflags]   return fl                                    y
                                                                         d
        [--libs]            return FL                                    s
        [--prefix]          return FL
        [--includedir]      return FL

Options to compile and link an appl
        [-g]                compile the program with debugging information
        [-Dname[=value]]    compile the program with the given define
        [--compile program.cxx]
        [--post program]    prepare the program for desktop use

groth@laerche(2:1070):PE-2019$ fltk-config --compile fhello.cpp
g++ -I/home/groth/.local/include -I/usr/include/freetype2 -I/usr/include/libpng16 -D_LARGEFILE_SOURCE -D
_LARGEFILE64_SOURCE -D_THREAD_SAFE -D_REENTRANT -o 'fhello' 'fhello.cpp' /home/groth/.local/lib/libfltk.
a -lXrender -lXcursor -lXfixes -lXext -lXft -lfontconfig -lXinerama -lpthread -ldl -lm -lX11
groth@laerche(2:1071):PE-2019$ ./fhello
```

⇒ smart pointers are possible, but sometimes tricky usage as the standard documentation is free of examples on how to use them :(

using fltk-config and no pointers

```
#include <iostream>
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
int main(int argc, char **argv) {
  auto window = Fl_Window(340,180); window.color(FL_WHITE);
  auto box = Fl_Box(20,40,300,100,"Hello, World\nno Pointer!");
  box.box(FL_UP_BOX); box.color(FL_WHITE);
  box.labelfont(FL_BOLD+FL_ITALIC);
  box.labelsize(36);box.labelcolor(FL_BLACK);
  //box.labeltype(FL_SHADOW_LABEL);
  window.end();
  window.show(argc, argv);
  std::cout << "fltk working!" << std::endl;
  return Fl::run();
}
```

```
$ fltk-config  –compile fhello4.cpp
Ok
```



Hello, World
no Pointer!

⇒ Looks easier, but different syntax then to pass widgets around in callback methods (onclick events etc).

Tools, Hints for C++ Developers

- g++ -O3 flag
- installs: package manager (dnf, apt, pacman, winget, brew)
- project: make (cmake - more bloated)
- formatters: clang-format / astyle
- linters: clang-tidy / cpplint
- documenters: mkdoc / doxygen
- terminal tools: tmux, fzf, strip (reduce executable size)

Summary

- exceptions
- smart pointers
- unit testing
- gui libraries (fltk)
  - install
  - hello world

  Next week ???:
  - (More fltk widgets, Trees, Tables, Charts, ...)???
  - Test exam ...!

Exercise / Test Exam

Will be uploaded next week.

References

Tutorialspoint — C++ Exception Handling. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm. [Online; accessed 24-June-2019].

Tutorialspoint — C++ Dynamic Memory. Tutorialspoint, SIMPLYASLEARNINMG, 2019. URL https://www.tutorialspoint.com/cplusplus/cpp_dynamic_memory.htm. [Online; accessed 16-June-2019].

thispointer.com: C++11 Smart Pointer – Part 6: unique_ptr. thispointer.com, 2019. URL https://thispointer.com/c11-unique_ptr-tutorial-and-examples/. [Online; accessed 24-June-2019].

thispointer.com: C++11 Smart Pointer – Part 1: shared_ptr. thispointer.com, 2019. URL `https://thispointer.com//learning-shared_ptr-part-1-usage-details`. [Online; accessed 24-June-2019].