The C++ Standard Library (1)

Lecture 3

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Abstract

- We will look at the C++ Standard Library
  - A vast collection of extremely useful containers, algorithms, and supporting data structures
- This will be a whirlwind overview over certain aspects and facilities
  - Date/time computation
  - Filesystem operations
  - Revisiting I/O (input/output)
  - Containers
    - Array containers
      - Vector, array
    - Associative container
      - Unordered map/set, map/set
    - Specialized containers
      - Lists, deque,
    - Container adaptors

Git & Github

Managing Source Code Histories

Git and GitHub

- Git and GitHub are common tools used in programming
  - Help managing different versions of your code and collaborate with other developers
- Git was developed in 2005 by Linus Torvalds
  - Open source software for tracking changes in a distributed version control system
- Git is made freely available for anyone to modify and use
  - Available on all platforms, widely used
- Git tracks changes via a distributed version control system
  - Git can track the state of different versions of all files in your project
  - It is distributed because you can access your code files from another computer – and so can other developers.

Git and Github

- GitHub is a web-based platform where Git users build software together
- GitHub is also an hosting provider and version control platform you can use to collaborate on open source projects and share files
- When you're using GitHub, you're working with Git under the hood

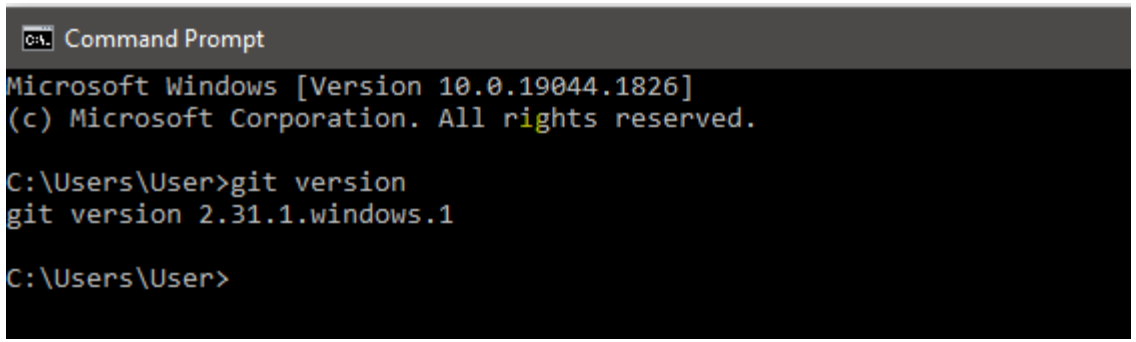
- Git is the (command-line) tool that manages the files
  - VSCode (and many other IDEs) have a graphical user interface that sits on top of Git
- Github is (one of the existing and free) web-platforms you can use to host your Git repositories

Git and Github

- Millions of people all over the world use these tools, and the numbers just keep going up
  - It is being used for any programming language
- More companies are requiring new hires to know how to use Git and GitHub
  - So if you're looking for a developer job, these are essential skills to have

Setting Things Up

- Install Git
  - Comes preinstalled in some Macs and Linux-based systems
  - Simple install for all platforms: https://git-scm.com/download



- Create account on Github: https://github.com

Connect Git to Github

- Set Git user name and email address (do this once)

```
git config --global user.name "Hartmut Kaiser"
git config --global user.email "hartmut.kaiser@gmail.com"
```

- Use same email address as you used for registering on Github

Github Classroom

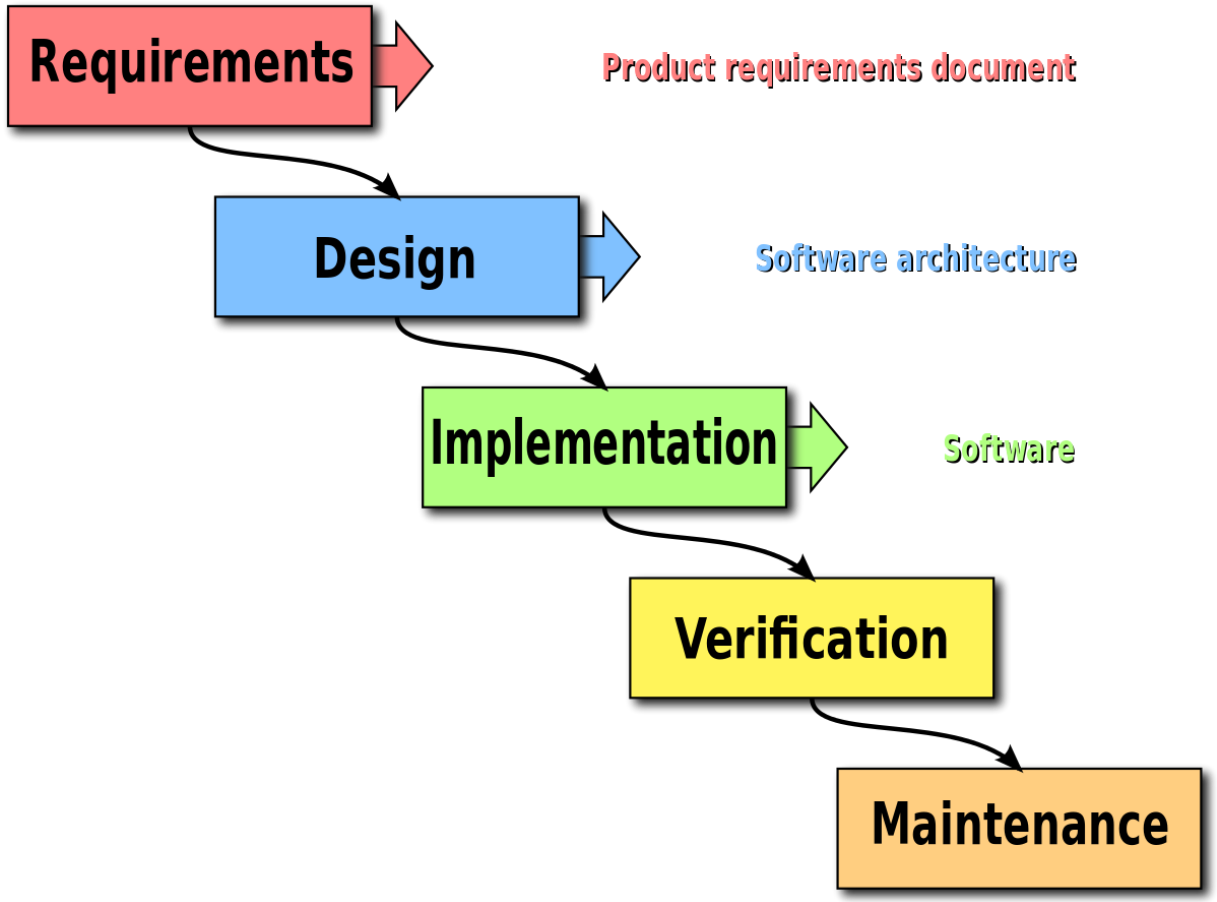
- Website helping to manage assignments
- Based on starter codes in a repository
  - Manages clones (copies) of this repository for each student
  - All repositories are hosted on Github
- Enables automatic grading
- Enables individual feedback to each student
- Well integrated in VSCode

Software Development Notes

The “natural” development process

- Get project requirements
- Disappear and start coding immediately
- Abruptly stop coding and start testing
- Emerge from cave to demo project
- Haphazardly fix bugs as they emerge

Waterfall: A flawed Engineering Process



- Supposedly:
  - Simple to understand and manage
  - Engineers can specify things completely
  - Fixing problems in earlier phases is cheaper

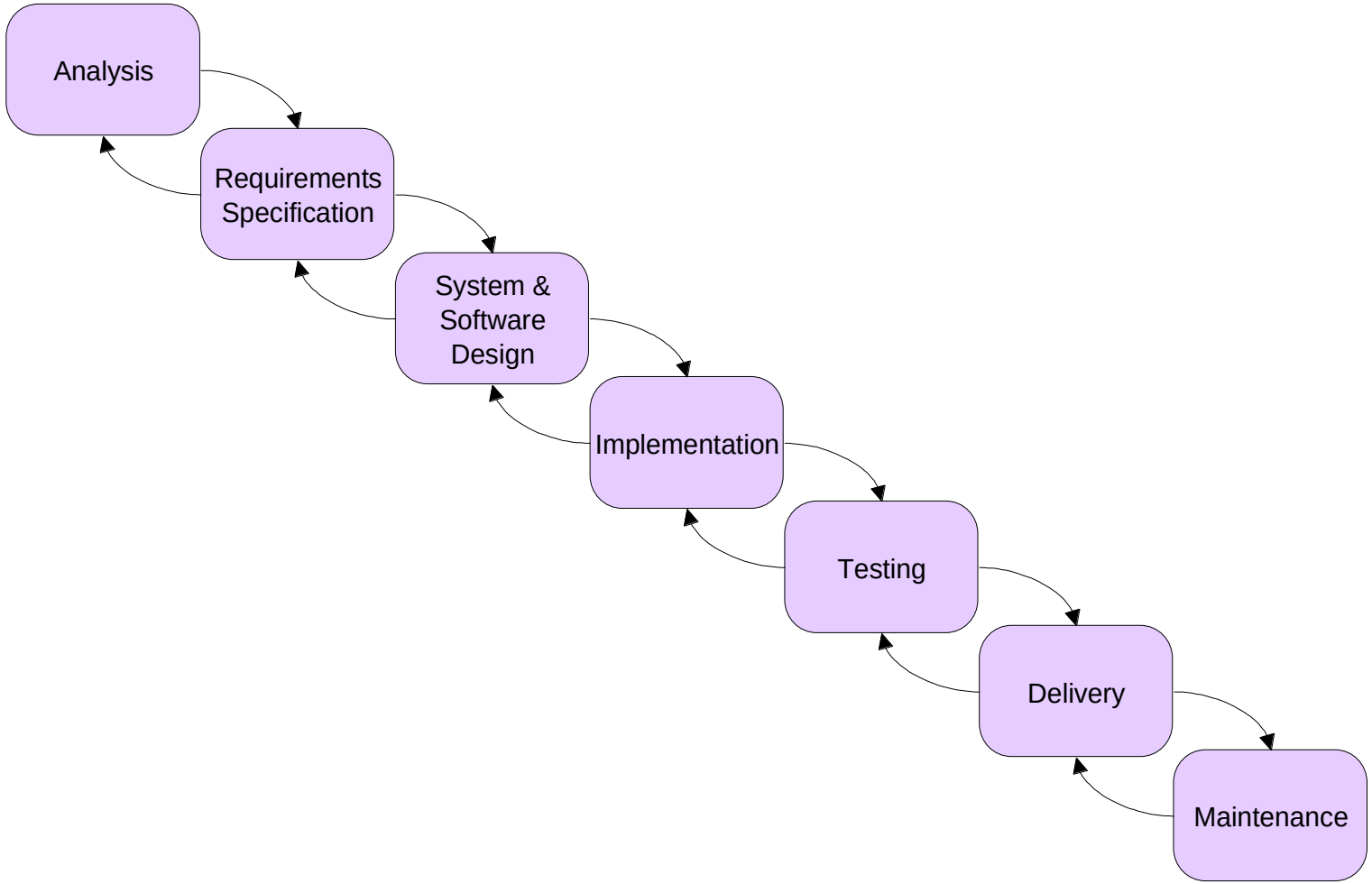
Waterfall is a generic term for the one-way methods used by entirely too many of companies

What's Wrong?

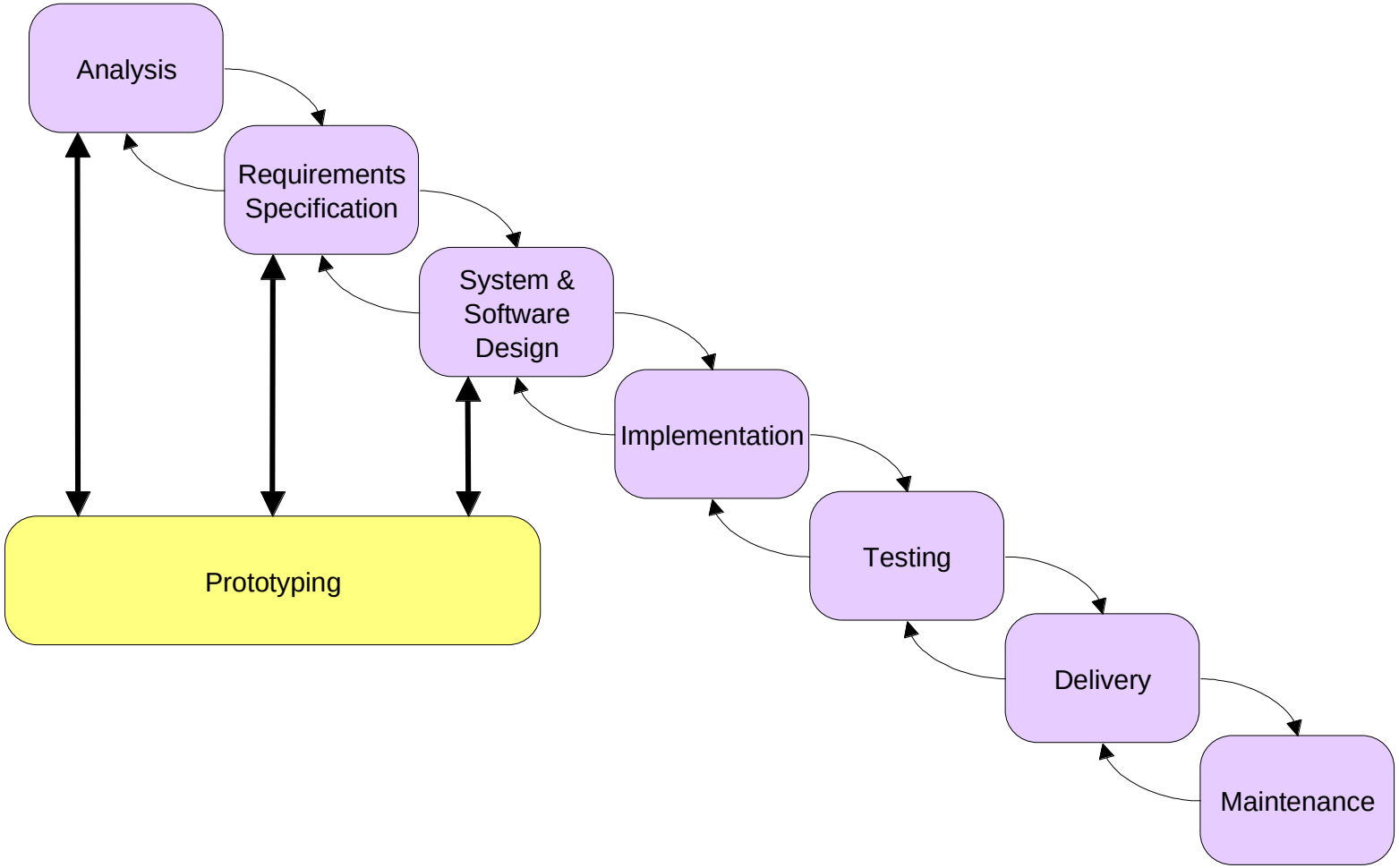
- Software engineering is not like other disciplines
- Requirements specification is never complete (or unambiguous)
- Stakeholders change their minds often
- Nearly infinite complexity in software

- Causing software that is:
  - Heavily delayed
  - Significantly over budget
  - Does not meet the need

Alternative Lifecycle: Modified Waterfall Lifecycle

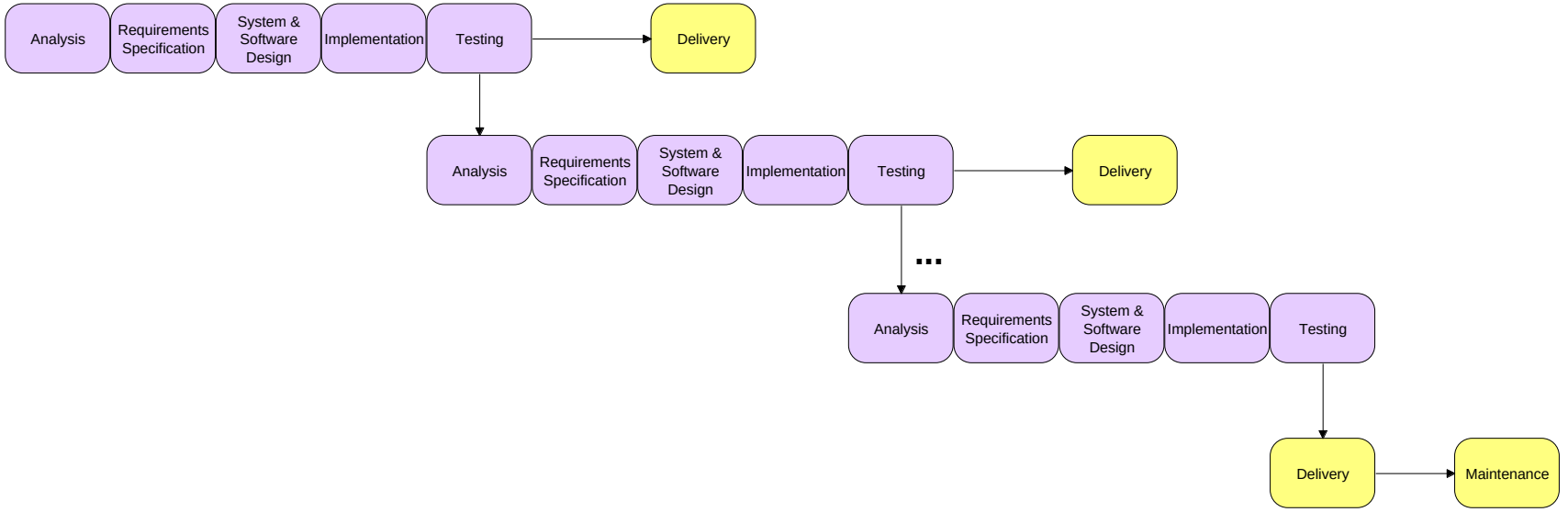


Alternative Lifecycle: Rapid Prototyping



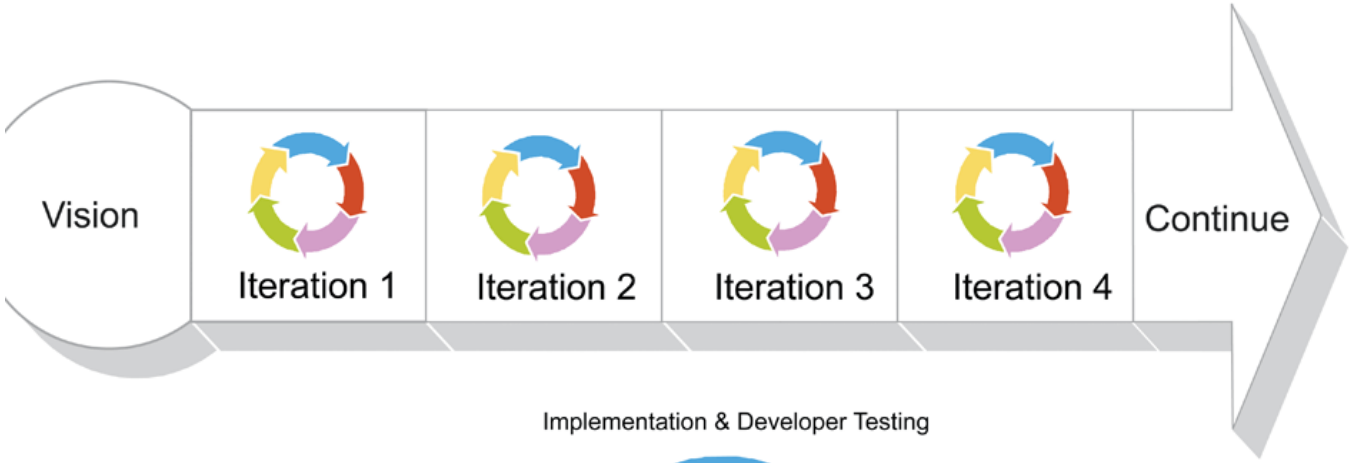
Phased Development

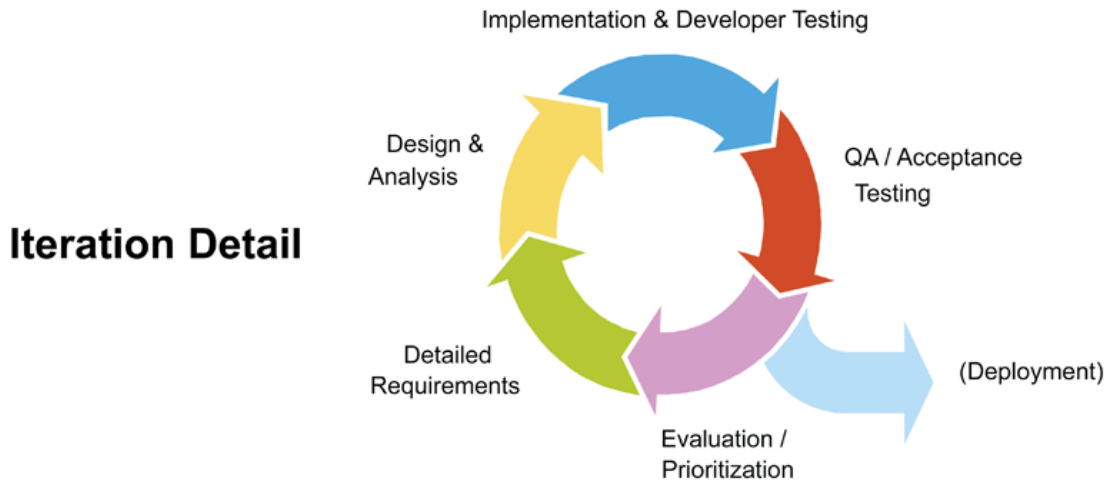
- Incremental
  - Initial release has limited functionality
  - Each release adds new subsystem
- Iterative
  - Each release delivers full system
  - Subsystem changes with new release



An Iterative Development Lifecycle

- Vision is established
- Development cycles through iterations
- Frequent deployment
- User-focus





The Standard C++ Library

Date and Time

Date and Time

- C++ offers a full featured time and date manipulation library
  - Fairly complex, so we will give a light overview
- All date/time types are defined in namespace `std::chrono`
- There are two primary clocks to consider for obtaining time
  - The system clock (`std::chrono::system_clock`)
  - The steady clock (`std::chrono::steady_clock`)
- The system clock matches the system time and should be used when working with the actual time (UTC).
  - Not guaranteed to be contiguous
  - Use `steady_clock` for benchmarking

Date and Time

```
// Get current time from system_clock, returns a time point
auto tp1 = std::chrono::system_clock::now();
std::cout << "The current UTC time is: "
          << tp1 << "\n";

// The current UTC time is: 2023-06-22 20:51:41.278848 +0000


// Difference of two time points is a duration
auto tp2 = std::chrono::system_clock::now();
auto duration = tp2 - tp1;
std::cout << "Time elapsed between calls: "
          << duration << "\n";

// Time elapsed between calls: 21587 [1/10000000]s
```

The system and the steady clock

- The system clock's problem is that it can be externally adjusted (when synchronizing the system's clock with time servers).
  - This poses a problem when we try to make accurate measurements by capturing specific time points.
- The steady clock is a monotonic clock that is not externally adjusted
  - It is meant for measuring time periods, for example, performance metrics.
  - It is unrelated to system time (it can be the time since the last system reboot).

The steady clock

```
// Bring in literals from std::chrono
using namespace std::literals;

// Same interface as system_clock:
auto tp1 = std::chrono::steady_clock::now();
std::this_thread::sleep_for(1ms);              // millisecond literal
auto tp2 = std::chrono::steady_clock::now();

auto duration = tp2 - tp1;
std::cout << "Slept for " << duration << "\n";
std::cout << "Which is " << (duration - 1ms)
          << " more than the requested duration.\n";

// Slept for 8867900ns
// Which is 7867900ns more than the requested duration.
```

Timepoints and Durations

- The supported arithmetic operations follow the expected semantics
- Time literals represent durations
- Durations can be added together or multiplied with scalars
- Adding a duration to a time point produces a new time point with the desired offset
- A difference of two time points is a duration
- Negative durations are supported as well

Date Manipulations

```
// Day in a year can be specified using literals and operator/
std::cout << "Christmas 2024 is on a "
          << weekday(2024y / December / 24d) << "\n";       // Christmas 2024 is on a Tue

// Last day for February 2020
std::cout << "Leap day in 2020: "
          << year_month_day(2020y / February / last) << "\n"; // Leap day in 2020: 2020-02-29

// Last Sunday of 2024
year_month_weekday_last last_sunday = 2024y / December / Sunday[last];
std::cout << "Last Sunday in 2024: "
          << year_month_day(last_sunday) << "\n";           // Last Sunday in 2024: 2024-12-29

// US Thanksgiving in 2024
auto thanksgiving = November / Thursday[4];
std::cout << "Thanksgiving in 2024: "
          << year_month_day(thanksgiving / 2024y) << "\n";  // Thanksgiving in 2024: 2024-11-28
```

Timezones

```
// Monthly meeting each first Wednesday 15:00, as un-zoned time
auto meeting = local_days(2023y / June / Wednesday[1]) + 15h;

// local_time -> zoned_time: the time is local to this zone
auto prague = locate_zone("Europe/Prague");
auto new_york = locate_zone("America/New_York");

zoned_time<seconds> local(prague, meeting);
zoned_time<seconds> remote(new_york, local);                // time zone conversion
std::cout << "Prague time: " << local << "\n";              // Prague time: 2023-06-07 15:00:00 GMT+2
std::cout << "New York time: " << remote << "\n";          // New York time: 2023-06-07 09:00:00 EDT

// Next week's meeting
zoned_time<seconds> next_local(prague, meeting + weeks(1));
zoned_time<seconds> next_remote(new_york, next_local);
std::cout << "Prague next time: " << next_local << "\n";    // Prague time: 2023-06-14 15:00:00 GMT+2
std::cout << "New York next time: " << next_remote << "\n"; // New York time: 2023-06-14 09:00:00 EDT
```

File System

Filesystem Operations

- The `std::filesystem` library offers file-system exploration, manipulation and querying tools
- Files and directories are identified by their paths, which are, by default, relative.
  - The `std::filesystem::absolute()` function turn any relative path into an absolute one based on the current working directory
  - The `std::filesystem::canonical()` function turn any relative path into an absolute one that has no dot, dot-dot elements, or symbolic links
  - The `std::filesystem::equivalent` comparator can be used to check whether two paths refer to the same file-system entity
    - Even taking into account symbolic links, etc.

Filesystem Operations

```
// Construct a path for the current directory
std::filesystem::path local(".");
std::cout << "local == " << local << "\n";          // local == "."

// Get the absolute path, i.e. a path from root
std::filesystem::path from_root = absolute(local);
std::cout << "from_root == " << from_root << "\n"; // from_root == "/some/path/."

// Get the canonical (normalized) full path
std::filesystem::path unique = canonical(local);
std::cout << "unique == " << from_root << "\n";    // unique == "/some/path"

bool eq1 = std::filesystem::equivalent(local, from_root);
bool eq2 = std::filesystem::equivalent(local, unique);
std::cout << std::boolalpha << "eq1 == " << eq1 << ", eq2 == " << eq2 << "\n";
                                                   // eq1 == true, eq2 == true
```

Filesystem Operations

- Directory content can be enumerated using `directory_iterator` or `recursive_directory_iterator`

```
std::filesystem::path local(".");

// iterate over entries in directory specified by path
for (auto const& entry : std::filesystem::directory_iterator(local))
{
    print_file_size(entry);
}

// recursively iterate over entries in directory specified by path
for (auto const& entry : std::filesystem::recursive_directory_iterator(local))
{
    print_file_size(entry);
}
```

Filesystem Operations

- Print the size of a file

```
// process each std::filesystem::directory_entry
void print_file_size(std::filesystem::directory_entry const& entry)
{
    if (entry.is_regular_file())    // Type of object
    {
        // Filename can be extracted from the path
        auto filename = entry.path().filename();
        std::cout << filename << ": " << file_size(entry) << "\n";
                    // size, permissions, etc...
    }
}
```

Filesystem Operations

```
// Create a file with the content: "Current content\n"
std::filesystem::path file = "current_file";
{
    // canonical path must exists, however, since we are just
    // about to create the file, we need to use weakly_canonical
    std::ofstream f(weakly_canonical(file));
    f << "Current content\n";
}

// Create a directory if it doesn't exist
std::filesystem::path backup_folder = "./backup";
if (!exists(backup_folder))
    create_directory(backup_folder);
```

Filesystem Operations

```
// Check for sufficient space
if (space(backup_folder).available < file_size(file))
    throw std::runtime_error("Not enough space for backup.");

// Create a "unique" filename in the backup folder
std::filesystem::path backup_file = backup_folder / file.filename();

// Copy the file to backup
copy(file, backup_file);

// Update the symlink to point to this backup
std::filesystem::path symlink = file.parent_path() / "current_backup";
if (exists(symlink))
    remove(symlink);
create_symlink(backup_file, symlink);
```

Revisiting I/O

Revisiting I/O

- In the previous example, we used a new type of stream, `std::ofstream`. Similar to `std::cin` and `std::cout`, files are also represented by streams.

```
{
    // Open for writing or create if file doesn't exist.
    std::ofstream out("data.txt");
    out << "Hello World!\n";
}    // out closes

{
    // Open for reading.
    std::ifstream in("data.txt");

    std::string line;
    std::getline(in, line);

    std::cout << "line == " << line << "\n";  // line == "Hello World!"
}
```

Revisiting I/O

- It is also possible to use an output stream that fills a `std::string`

```
std::string line;

{
    std::stringstream strm;
    strm << "Hello world!";
    line = strm.str();
}

std::cout << "line == " << line << "\n";    // line == "Hello World!"
```

I/O for your own Types

```
struct X
{
    int64_t value;
};

std::ostream& operator<<(std::ostream& out, X const& el)
{
    return out << el.value;
}

std::istream& operator>>(std::istream& in, X& el)
{
    return in >> el.value;
}
```

I/O for your own Types

```
{
    std::ofstream out("data.txt");

    X a{42};
    X b{7};
    out << a << " " << b;
}    // out closes

{
    std::ifstream in("data.txt");

    X a{0};
    X b{0};
    in >> a >> b;

    std::cout << "a.value == " << a << ", b.value == " << b
              << "\n";    // a.value == 42, b.value == 7
}
```

Containers, Algorithms & Iterators

Containers, Algorithms & Iterators

- The Standard Template Library is an extensible framework dealing with data in a C++ program.
- First, I will present the general idea, then the fundamental concepts, and finally examples of containers and algorithms.
- The key notions of sequence and iterator used to tie data together with algorithms (for general processing) are also presented.

- We can (already) write programs that are very similar independent of the data type used
  - Using an `int` isn't that different from using a `double`
  - Using a `std::vector<int>` isn't that different from using a `std::vector<string>`

Common Tasks

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the **N**th element)
  - By value (e.g., get the first element with the value **"Chocolate"**)
  - By properties (e.g., get the first elements where "**age < 64**")
- Add data
- Remove data
- Sorting and searching
- Simple numeric operations

Ideals

- We'd like to write common programming tasks so that we don't have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data
  - Finding a value in a `std::vector` isn't all that different from finding a value in a `std::list` or an array
  - Looking for a `std::string` ignoring case isn't all that different from looking at a `std::string` not ignoring case
  - Graphing experimental data with exact values isn't all that different from graphing data with rounded values
  - Copying a file isn't all that different from copying a vector

Ideals (continued)

- Code that's
  - Easy to read
  - Easy to modify
  - Regular
  - Short
  - Fast
- Uniform access to data
  - Independently of how it is stored
  - Independently of its type
- …

Ideals (continued)

- …
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
  - Retrieval of data
  - Addition of data
  - Deletion of data
- Standard versions of the most common algorithms
  - Copy, find, search, sort, sum, …

Examples

- Sort a vector of strings
- Find an number in a phone book, given a name
- Find the highest temperature
- Find all values larger than 800
- Find the first occurrence of the value 17
- Sort the telemetry records by unit number
- Sort the telemetry records by time stamp
- Find the first value larger than “Petersen”?
- What is the largest amount seen?
- Find the first difference between two sequences
- Compute the pair wise product of the elements of two sequences
- What’s the highest temperatures for each day in a month?
- What’s the top 10 best-sellers?
- What’s the entry for “C++” (say, in Google)?
- What’s the sum of the elements?

Generic Programming

- Generalize algorithms
  - Sometimes called “lifting an algorithm”
- The aim (for the end user) is
  - Increased correctness
    - Through better specification
  - Greater range of uses
    - Possibilities for re-use
  - Better performance
    - Through wider use of tuned libraries
    - Unnecessarily slow code will eventually be thrown away
- Go from the concrete to the more abstract
  - The other way most often leads to bloat

Lifting example (concrete algorithms)

```
// one concrete algorithm (doubles in array)
double sum(double array[], int n) {
    double s = 0;
    for (int i = 0; i < n; ++i)
        s = s + array[i];
    return s;
}

struct Node {
    Node* next; int data;
};

// another concrete algorithm (int's in list)
int sum(Node* first) {
    int s = 0;
    while (first != 0)
    {
        s += first->data;
        first = first->next;
    }
    return s;
}
```

Lifting Example (abstract the Data Structure)

```
// somehow parameterize with the data structure
int sum(data)
{
    int s = 0;                  // initialize
    while (not-at-end)
    {                           // loop through all elements
        s = s + get-value;      // compute sum
        get-to-next-data-element;
    }
    return s;     // return result
}
```

- We need three operations (on the data structure):
  - not at end
  - get value
  - get to next data element

Lifting Example (STL version)

```
// Concrete STL-style code  for a more general version of both algorithms
template <typename Iter, typename T>
T sum(Iter first, Iter last, T s) {
    while (first != last) {
        s = s + *first;
        ++first;
    }
    return s;
}

// 'Iter' should be an Input_iterator (supports ==, ++, *)
// 'T' should be something we can + and =, is the accumulator type
```

- Let the user initialize the accumulator:

```
float a[] = {1, 2, 3, 4, 5, 6, 7, 8};
double d = sum(a, a + std::size(a), 0.0);
```

Lifting Example

- Almost the standard library accumulate
  - Simplified a bit for terseness
- Works for
  - C arrays
  - `std::vector`'s
  - `std::lists`'s
  - `std::istream`'s
  - …
- Runs as fast as "hand-crafted" code
  - Given decent inlining
- The code's requirements on its data has become explicit
  - We understand the code better

Pattern: Iterator

- **Context:**
  - 1. An object (which we'll call the *container*) contains other objects (which we'll call *elements*).
  - 2. Clients (that is, methods that use the container) need access to the elements.
  - 3. The container should not expose its internal structure.
  - 4. There may be multiple clients that need simultaneous access.
- **Solution:**
  - 1. Define an iterator class that refers to one element at a time.
  - 2. Each iterator type needs to be able to keep track of the position of the previous and/or next element
  - 3. There are several variations of containers
    - Each exposes its own iterator classes
    - All iterators implement common interfaces
    - The client only needs to know the interface, not the concrete classes.

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP