Working with Types

Lecture 2

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

What is a Type?

Abstract

- What are types? What are objects?
- A pattern for regular types: `singleton`
  - Semi-regular `singleton`
  - Regular `singleton`
  - Totally ordered `singleton`
- Another useful regular type: `instrumented`

What is a ‘type’?

- A ‘type’ (of an object) defines the following things:
  - The amount of memory required to store all the data that is needed to support the operations valid for a type
  - The rules of how to interpret the bits in that memory as values in order to be able to make sense of the bit-salad
  - The set of values that are valid
  - The set of operations that are valid on those values
- Examples of types:
  - `int`, `double`, `float` (built-in types)
  - `token`, `token_stream`, `std::vector`, etc. (user-defined types)

What is an ’object’?

- An object is an instance of a type
  - Occupies memory
  - Has an optional name (is a variable)
  - Has a lifetime
- Objects in C++ don’t change their type
  - C++ is a type-safe language
  - C++ checks types and type compatibility at compile time
- Examples of objects:
  - `int i = 0;`
  - `token t('+');`
  - `std::vector<int> v = {1, 2, 3, 4, 5};`

Type Regularity

Regular Types

- Let's informally define what it means for a type to be 'Regular'
  - It behaves like an `int` (or any other built-in type)
- Regularity defines a set of properties a type should have
- Understanding regularity is important as it will allow us to understand what algorithms are allowed to do
  - Use only operations allowed for regular types
- Regular types are those that can be stored in standard containers (like `std::vector<T>`)
  - What properties must `T` have to be regular?
  - IOW, what properties must `T` have in order for it to be stored in a `std::vector<T>`
- We should be able to rely on `std::vector<T>` being regular if `T` is regular
- We will use **concepts** to describe those properties

Semiregular Types: Copy constructor

- `Semiregular` is a bit weaker than `Regular`
- We should be able to write:
  - Copy constructor (initializes `a`)
    - `T a(b);`
    - `T a = b;`
  - Both are equivalent, even the same, if `b` is of type `T`
- What are the semantics of this operation?
  - After this operation `a` should be equivalent to `b`
- What is equivalence?
  - A relation `R(a, b) = true` is equivalence, if it satisfies
    - symmetric: `R(a, b) <=> R(b, a)`
    - reflexive: `R(a, a)`
    - transitive: `R(a, b)` and `R(b, c) => R(a, c)`

Semiregular Types

- We actually want something significantly stronger. We want equality
- A copy is something which is equal to the original, but not identical to it
  - After `a` is copy-constructed from `b` then `a == b`, whatever the meaning of equality
  - After `a` is copy-constructed from `b` they have distinct identity markers.
    - In C++ the identity marker is usually the object's address: `&a != &b` (location in memory)
- All copy constructors must behave this way.
  - If somebody clever comes and says, "oh we're going to have semantics where we're going to have this shared thing".
  - Will it work? No. Copy has to construct a different thing.

Semiregular Types: Assignment

- Assignment operator:
  - `T a; a = b;`
- Construction (initialization) and assignment must be equivalent (lead to the same results):
  - `T a1(b) <=> T a2; a2 = b;` → `a1 == a2`
- Initialization creates an initial state for a new object
- Assignment first cleans up old state of an existing object and then initializes its new state
- In order for these operations to have correct semantics, the types involved have to have equality defined (`operator==()`)
  - How would you know otherwise if two instances are equal?

Semiregular Types: Destructor

- Even if you don't call destructors directly (the compiler does, though):
  - `~T();`
- Ends the lifetime of an object

C++ Class Anatomy

class.cpp class.cpp

```cpp
class MyNewType {
public:
    MyNewType();
    ~MyNewType();

public:
    void MemberFunction1();
    void MemberFunction2() const;
    static void StaticFunction();

public:
    MyNewType &operator+=(const MyNewType &other);
    std::ostream &operator<<(std::ostream &os, const MyNewType &obj);

private:
    int a_;
    std::vector<float> data_;
    MyType2 member_;
};
```

Class Definition

Constructors and Destructors

Member Functions

Operators

Data Members

Regular Types

- The concept `Regular` extends `Semiregular` with equality operators which are `==` and `!=`
- We should define `==` so that after constructing a copy, the original and the copy are equal
- `!=` should always behave like: `!(a == b)`
- Fundamentally equal is a symmetric function. It compares two things
  - We will implement it as a `friend` function, not as a member function

Total orderings

- The concept `TotallyOrdered` extends `Regular` by adding a comparison operator `<`
- operator `<` must obey the following mathematical properties:
  - Axiom 1: Anti-reflexive: `!(a < a)`
  - Axiom 2: Transitive: If `a < b` and `b < c` then `a < c`
  - Axiom 3: Anti-symmetric: If `a < b` then `!(b < a)`
  - Axiom 4: If `a != b` then `a < b` or `b > a`
- The semantics of `<` must be totally bound to the semantics of equality and related operations
  - The following should always be true, otherwise the world perishes.
    - `a >= b   --> !(a < b)`
    - `a > b    --> b < a`
    - `a <= b   --> !(b < a)`

Spaceship operator <=>

- C++20 introduced a simplified way of writing relational operators for user defined types
- Instead of implementing all relation operators (`<`, `<=`, `>`, `>=`, `==`, `!=`), you can implement a three-way comparison operator `<=>`
  - Returns `< 0`, if `a < b`
  - Returns `== 0`, if `a == b`
  - Returns `> 0`, if `a > b`
- The other relational operators are automatically synthesized by the compiler
- Simplest way is to define a member function for `X`:
  - `friend auto operator(X const&, X const&) = default;`
  - Will apply spaceship operator member-wise

Singleton

A Pattern for Regular Types

- We'll develop the simplest possible `Regular` (even `TotallyOrdered`) type: `singleton`
- The dictionary says: `singleton`, `pair`, `triple`, `quadruple`, etc.
  - A pair has two things, well a singleton has just one thing
- Can be used as a pattern (or "template") for any types you will want to create
  - It is the most simple class possible
    - It will have no (functionality oriented) code whatsoever
  - It is the most complete class possible
    - It will have all the language details about type creation that you need to know
  - It is a 'pure' regular type

Template Type Functions

- Singleton:

```
template <typename T>
struct singleton
{
    T value;
};
```

- `template <typename T>`
  - Why `template`?
    - We want to write something which takes one type and returns another type, i.e. a ‘type function’
    - In C++ the `template` mechanism is just that
- Simplest type function example
  - `int*`: i.e. get an `int` and return an `int*`
  - Transform one type into another type
- Singleton is a type function that takes a `T` and gives us a `singleton<T>`

Create new Types with Classes and Structs

- Classes are used to encapsulate data
  - along with methods to process them
- Every `class` or `struct` defines a new type
- Terminology:
  - Type or class to talk about the defined type
- A variable of such type is an instance of class (or an object)
- Classes allow C++ to be used as an Object Oriented Programming language
  - `std::string`, `std::vector`, etc. are all classes (predefined in the C++ standard library)

Compiler Generated Functions

- In C++, each user defined type has 6 special functions
  - Those are being generated by the compiler, if not explicitly provided
  - These functions are always available
- Here are the 6 functions
  - Default constructor
  - Destructor
  - Copy constructor
  - Copy assignment
  - Move constructor
  - Move assignment
- The special functions are being automatically used in certain situations
- The compiler generated functions simply apply its operation to all members of the type
- Unfortunately the spaceship operator is not automatically generated, you have to be explicit

Compiler Generated Functions

- Constructors are automatically used whenever a new instance of a user defined type is created (start lifetime of object)
  - Default constructor is used when no additional arguments are supplied:

```
singleton<int> s;
```

- Destructor is automatically called whenever an instance of a user defined type goes out of scope (ends the lifetime of an object)

- Copy constructor is used whenever a new instance of a user defined type is created and initialized from another instance of that type:

```
singleton<int> s1 = s;
```

- Copy assignment is used whenever an existing instance of a user defined type is assigned to another instance of that type:

```
singleton<int> s2; s2 = s1;
```

Compiler Generated Functions

- Any compiler generated special function by default invokes the corresponding special functions for all member data of the user defined type

  - Default constructor invokes default constructor of all members (in order of their definition)
  - Destructor invokes destructors of all members (in reverse order)
  - Etc.

Semi-regular `singleton`

- Let's implement support to make `singleton` Semiregular

```
struct singleton {
   // Semiregular:
    singleton() {}      // default constructor: could be implicitly declared sometimes
    ~singleton() {}     // destructor: could be implicitly declared

    singleton(singleton const& x) // copy constructor: could be implicitly declared
      : value(x.value)
    {
    }

    singleton& operator=(singleton const& x) // copy assignment operator: could be implicitly declared
    {
        value = x.value;
        return *this;
    }
};
```

Semi-regular singleton

- Let's implement support to make singleton Semiregular

```
// Semiregular:

singleton() = default;            // default constructor
~singleton() = default;           // destructor


// copy constructor
singleton(singleton const& x) = default;

// copy assignment
singleton& operator=(singleton const& x) = default;
```

Semi-regular `singleton`

- What are the semantics of the default constructor?
  - In this case you want whatever the default value of `T` is, to be constructed. The compiler will do this for us.
- The default constructor will always be synthesized by the compiler unless you have another constructor.
  - Always add it to avoid surprises!

Semi-regular `singleton`

- Should the destructor be virtual?
  - No! Why should it be?
  - Some people say ‘all destructors have to be virtual’ – they couldn’t be more wrong than that!
- Feel free to make singleton `final` to prevent people from deriving from it
  - There is no point in ever deriving from it anyways:

```
template <typename T>
struct singleton final
{
    // ...
};
```

Regular singleton

```cpp
// Regular
friend bool operator==(singleton const& x, singleton const& y)
{
    return x.value == y.value;
}
friend bool operator!=(singleton const& x, singleton const& y)
{
    return !(x == y);
}
```

- Recall that we decided not to define these as member functions
  - they are symmetric
  - `friend` functions inside the class declaration are not member functions
    - but still have all the access to all the members
  - More importantly this signature is nice. If you put it outside you discover you have to write an ugly thing

Equality and the three laws of thought

- The law of identity: `a == a`
  - Popeye the Sailor used to say, “I am, what I am”
- The law of non-contradiction:
  - You cannot have a predicate `P` be true and `!P` be true at the same time.
- The law of excluded middle:
  - Every predicate `P` must be either true, or false.

Exercise: Figure out a type that violates the law of identity

Totally ordered `singleton`

```cpp
// TotallyOrdered
friend bool operator<(singleton const& x, singleton const& y)
{
    return x.value < y.value;
}
friend bool operator>(singleton const& x, singleton const& y)
{
    return y < x;
}
friend bool operator<=(singleton const& x, singleton const& y)
{
    return !(y < x);
}
friend bool operator>=(singleton const& x, singleton const& y)
{
    return !(x < y);
}
```

Totally ordered singleton (C++20)

```cpp
// TotallyOrdered, synthesizes ==, !=, <, >, <=, >=
friend auto operator<=>(singleton const& x, singleton const& y)
{
    return x.value <=> y.value;
}
```

- Or even better:

```cpp
// TotallyOrdered, synthesizes ==, !=, <, >, <=, >=
friend auto operator<=>(singleton const& x, singleton const& y) = default;
```

- Spaceship operator `<=>`: should have been part of the language forever and should be synthesized by the compiler (same as 6 predefined functions)
  - Unfortunately it is not predefined

Concepts in C++ (since C++20)

- What requirements do we have to apply to T in order for `singleton<T>` to be valid?
  - C++20 introduced concepts allowing to constrain use of singleton

```
template <typename T>
  requires(std::regular<T> || std::semiregular<T> || std::totally_ordered<T>)
struct singleton final
{
    // ...
};
```

Exercise: Copy the file for `singleton` and modify it to write `pair`

- You might wonder how == will work, if you plug-in only a semiregular type T
  - In C++ templates, things don't have to be defined unless they are used
    - If `T` has no equality, `singleton<T>` will have copy constructor and assignment but no equality.
    - If `T` has an equality, then `singleton<T>` will have equality
    - Etc.

Instrumented

A performance measuring tool

instrumented<T>

- We will write a wrapper (adapter, decorator) class instrumented<T> which will take a type T and behave exactly like T
- We will be able to use instrumented<T> for any algorithm or container
  - It will behave normally, just like a T
  - In addition it will count all the operations that are applied to it
- Which operations should we count?
  - The ones specified by our concepts!
- T will be SemiRegular, Regular, or TotallyOrdered
  - Redefine all the operations: copy constructor, assignment, operator<, etc, adding code to count them

`instrumented<T>`

- For example:

```
std::vector<double> vec;
my_func(vec.begin(), vec.end());
```

- Could be replaced by:

```
std::vector<instrumented<double>> vec;
my_func(vec.begin(), vec.end());
```

- And it will count all operations
- Writing this particular class will teach to write `Regular` classes right.

instrumented<T>

- What to do with all the counts? Where do they get stored?
- We will define a base class to hold this data:

```
struct instrumented_base
{
    enum operations {
        n = 0, copy, assignment, destructor, default_constructor,
        equality, comparison, construction
    };

    static constexpr size_t number_ops = 8;
    static constexpr char const* counter_names[number_ops] = {
        "n", "copy", "assignment", "destructor", "default_constructor",
        "equality", "comparison", "construction"
    };
    static double counts[number_ops];
};
```

instrumented<T>

- Use this base class as:

```
template <typename T>
    requires(std::semiregular<T> || std::regular<T> || std::totally_ordered<T>)
struct instrumented : instrumented_base
{
    // ...
};
```

- Note that the base class does not change the size of `instrumented<T>`, i.e.

```
sizeof(instrumented<T>) == sizeof(T)
```

instrumented<T>

- Copy and paste the singleton.hpp file we created
- Replace the string `singleton` with `instrumented`
- In addition to existing operations, we'll add counting, e.g.:

```
instrumented(instrumented const& x)   // copy constructor
  : value(x.value)
{
    ++counts[copy];                   // ‘copy’ is a constant index
}

instrumented()                        // default constructor
{
    ++counts[default_constructor];    // ‘default_constructor’ is another constant index
}
```

Number of Unique Elements

- Counting operations and measuring execution time:
  - Using `std::set`

```
std::vector<instrumented<int>> v = {...};

std::set<instrumented<int>> set_of_ints(v.begin(), v.end());
std::println("{}", set_of_ints.size());
```

  - Using `std::sort` and `std::unique`:

```
std::sort(v.begin(), v.end());
std::println("{}", std::unique(v.begin(), v.end()) - v.begin());
```

Measuring Execution Time

- We will use a simple class `timer`:

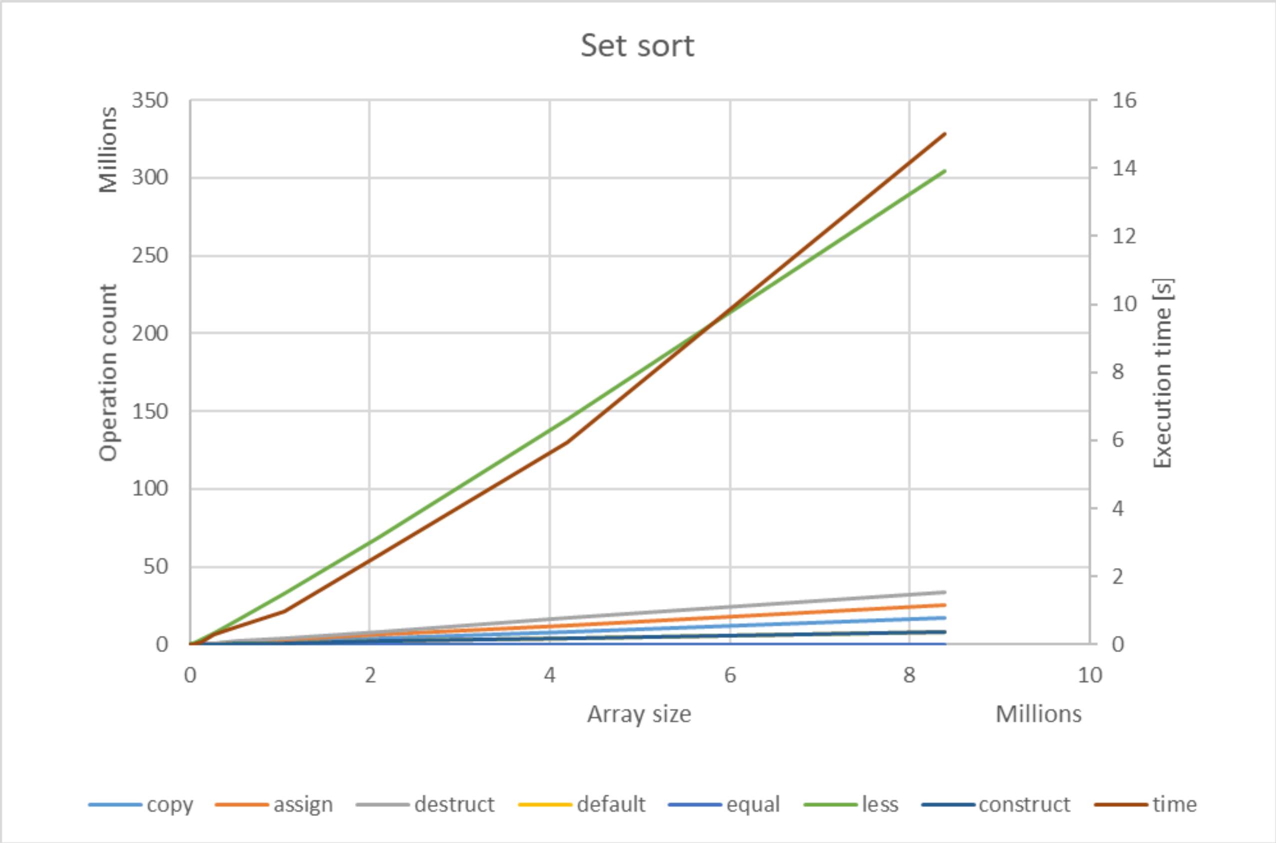
```
class timer {
private:
    using time_t = std::chrono::time_point<std::chrono::system_clock>;
    time_t start_time, stop_time;

public:
    timer() = default;

    time_t start()   { return (start_time = std::chrono::system_clock::now()); }
    time_t stop()    { return (stop_time  = std::chrono::system_clock::now()); }

    double elapsed() {
        auto diff = stop_time - start_time;
        return std::chrono::duration_cast<std::chrono::milliseconds>(diff).count();
    }
};
```

Measuring Execution Time

```
#include "timer.hpp"

int main(int argc, char* argv[]) {

    timer t;
    t.start();

    // do something that you would like t to measure the
    // execution time for

    t.stop();

    std::println("The code took {} milliseconds to execute", t.elapsed());
    return 0;
}
```

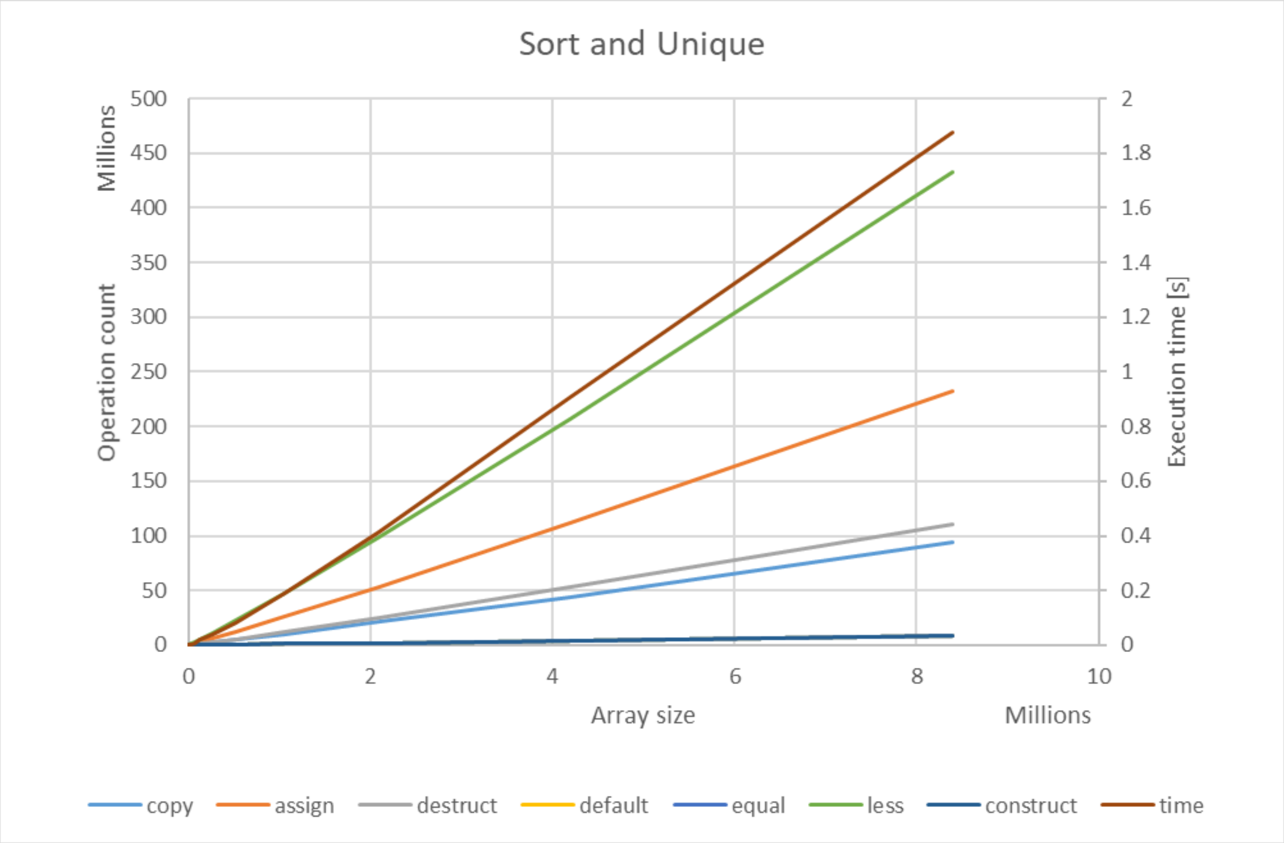
Using `std::set`

n	copy	assign	destruct	default	equal	less	construct	time
16	30	44	62	16	0	101	16	5.40E-06
32	57	82	121					
64	123	182	251					
128	252	376	508	1				
256	506	756	1018	2				
512	1021	1530	2045	5				
1024	2038	3052	4086	10				
2048	4090	6132	8186	20				
4096	8181	12266	16373	40				
8192	16375	24558	32759	81				
16384	32756	49128	65524	163				
32768	65522	98276	131058	327				
65536	131061	196586	262133	655				
131072	262130	393188	524274	1310				
262144	524283	786422	1048571	2621				
524288	1048560	1572832	2097136	5242				
1048576	2097134	3145692	4194286	10485				
2097152	4194292	6291432	8388596	20971				
4194304	8388590	12582876	16777198	41943				
8388608	16777197	25165786	33554413	83886				

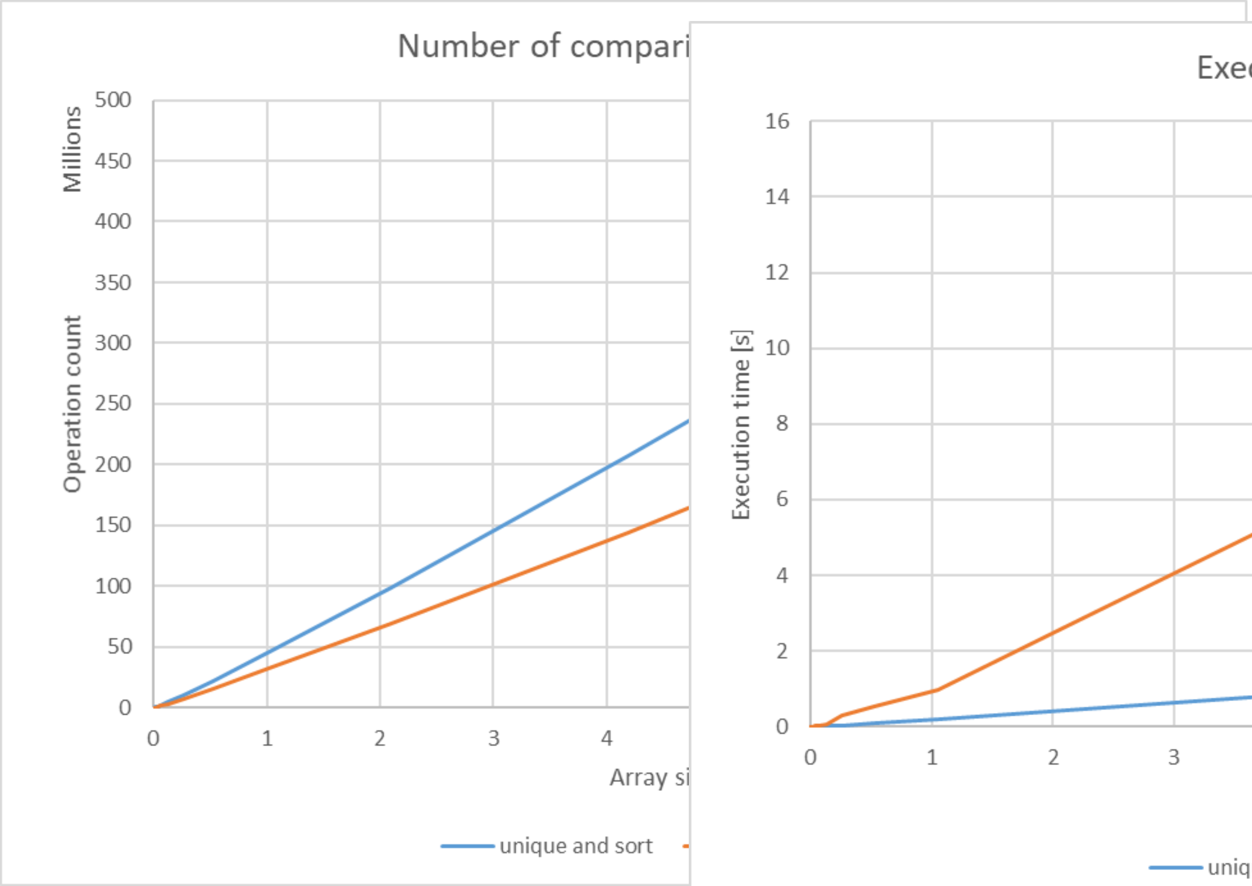


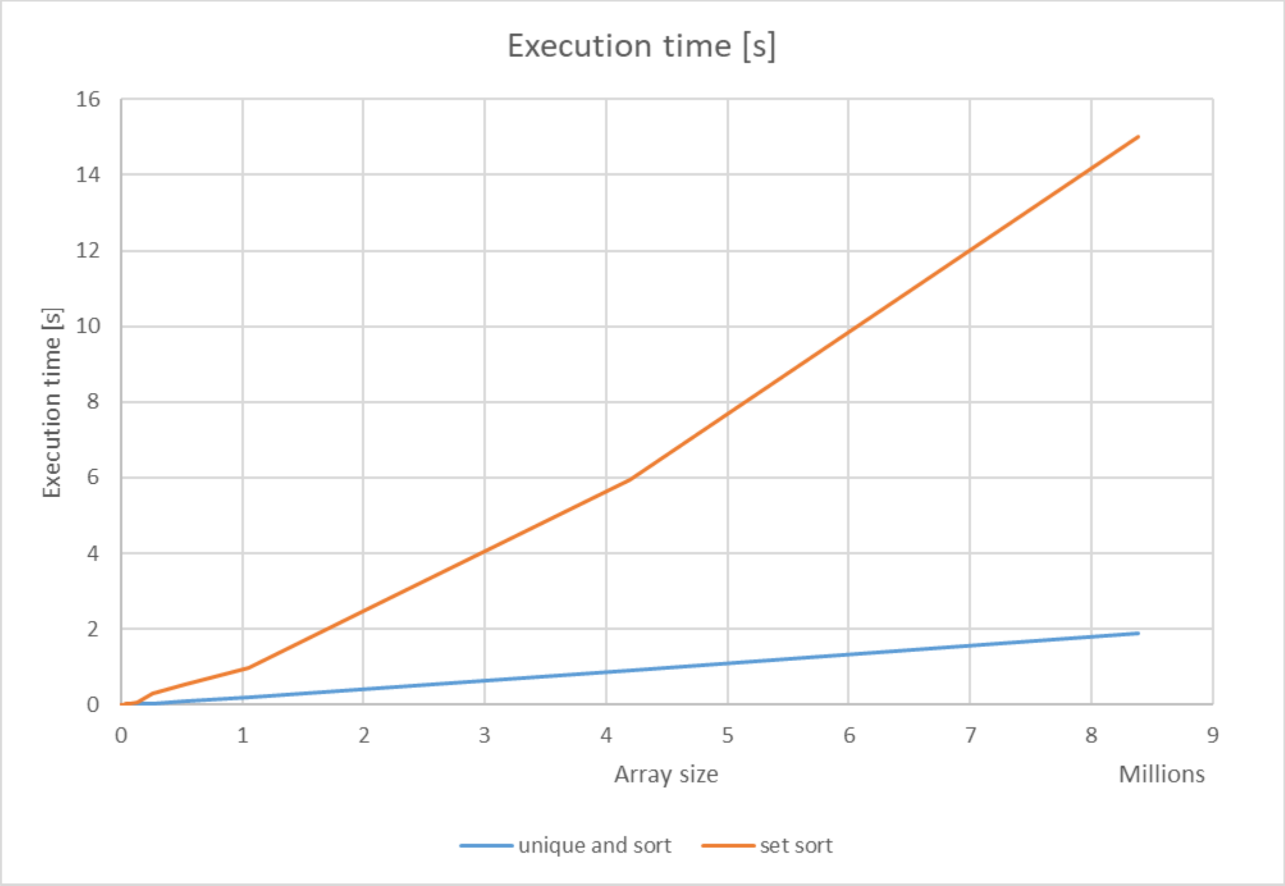
Using `std::sort` and `std::unique`

n	copy	assign	destruct	default	equal	less	construct	time
16	29	125	61	16	15	162	16	4.01E-06
32	61	421	125					
64	157	651	285					
128	404	1614	660	[illegible]				
256	856	3190	1368	[illegible]				
512	2200	7000	3224	[illegible]				
1024	4895	14949	6943	[illegible]				
2048	10202	31452	14298	[illegible]				
4096	23809	69595	32001	[illegible]				
8192	54365	151993	70749	[illegible]				
16384	104148	294590	136916	[illegible]				
32768	227532	630928	293068	[illegible]				
65536	512780	1374424	643852	[illegible]				
131072	1051039	2805207	1313183	[illegible]				
262144	2329354	6063902	2853642	[illegible]				
524288	4619934	12041526	5668510	[illegible]				
1048576	10067973	25735953	12165125	[illegible]				
2097152	21256236	53714098	25450540	[illegible]				
4194304	44364666	111139688	52753274	[illegible]				
8388608	93613867	232055273	110391083	[illegible]				



Number of unique elements





Conclusions

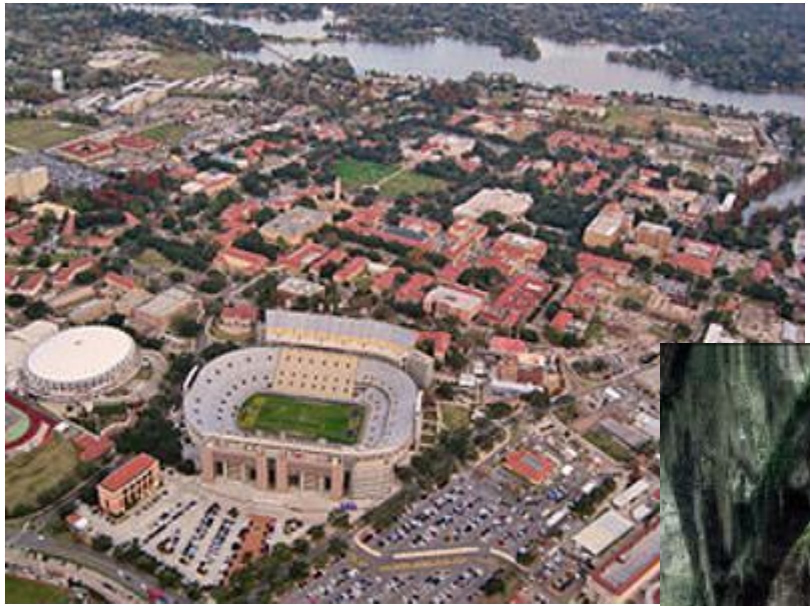
- Even if the number of operations is larger, the code may run faster
- Textbook solutions are often outdated
  - They are based on the understanding of how computers worked 15 years ago
- Understanding computer architecture is critically important in order to write efficient software
- Understanding Big-O complexity characteristics of algorithms (and data structure functionalities) is equally important
- All depends on the used data structures and how well those are aligned with how computers work
  - Always use `std::vector<T>`
  - If you think you can't use it, try again and find a way so you can

Exercise

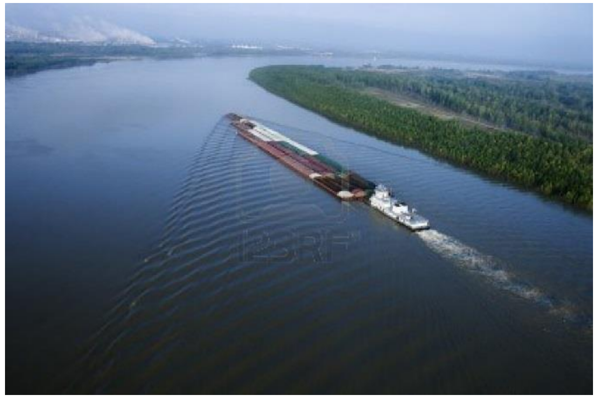
- Measure and compare the amount of operations and the overall execution time for
  - `std::sort`
  - `std::stable_sort`
- Explain what you're seeing

Summary

- We know that `singleton<T>` and `instrumented<T>` conform to the type requirements (concepts) that all standard algorithms and containers expect
  - They can be used anywhere it would be valid to use `T`
- This guarantees that these types can be used with all algorithms and containers
  - This will not change the semantics of the algorithms
- The understanding of what concepts are assumed to apply for a given function or data structure is important
  - Allows to formalize in what contexts a function or data structure is guaranteed to produce correct results
- If a function or data structure works with a type that conforms to a set of concepts
  - We know that it will work with any other type that conforms to those concepts as well



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP