Open4Tech Summer School 2021

So You Think You Can #
Building an app using Blockchain
A list< > of data structures you should add in your learning queue< >
Build a video call react app with WebRTC and Socket.io
Importance of good coding habits

Build cryptocurrencies exchange using React
Why we code
Getting a11y right
State of the Art Natural Language Processing for Noobs
Java Swing Crash Course

28 iunie - 16 iulie 2021
http://inf.ucv.ro/~ summer-school/

Departamentul de Informatică

CAPHYON

NETROM premium software

syncro soft

Society for Computing Technologies

	Luni	Marti	Miercuri	Joi	Vineri
	28 iunie	29 iunie	30 iunie	1 iulie	2 iulie
2-4pm	So You Think You Can # (hashing algorithms & containers)	So You Think You Can # (hashing algorithms & containers)	A list<> of data structures you should add in your learning queue<>	A list<> of data structures you should add in your learning queue<>	Building an app using Blockchain
4-6pm	Why we code	Why we code	Getting a11y right	Java Swing Crash Course	
	5 iulie	6 iulie	7 iulie	8 iulie	9 iulie
2-4pm	Build a video call react app with WebRTC and Socket.io	Build a video call react app with WebRTC and Socket.io	Build a video call react app with WebRTC and Socket.io	State of the Art Natural Language Processing for Noobs	State of the Art Natural Language Processing for Noobs
4-6pm					
	12 iulie	13 iulie	14 iulie	15 iulie	16 iulie
2-4pm	Build cryptocurrencies exchange using React	Build cryptocurrencies exchange using React	Code conventions: Importance of good coding habits	Code conventions: Importance of good coding habits	
4-6pm					

So You Think You Can

Hashing Algorithms and Containers

inf.ucv.ro/~summer-school

Victor Ciura
Principal Engineer
@ciura_victor

June 2021

⚠️ Warning

So You Think You Can

Not a C# workshop

Abstract

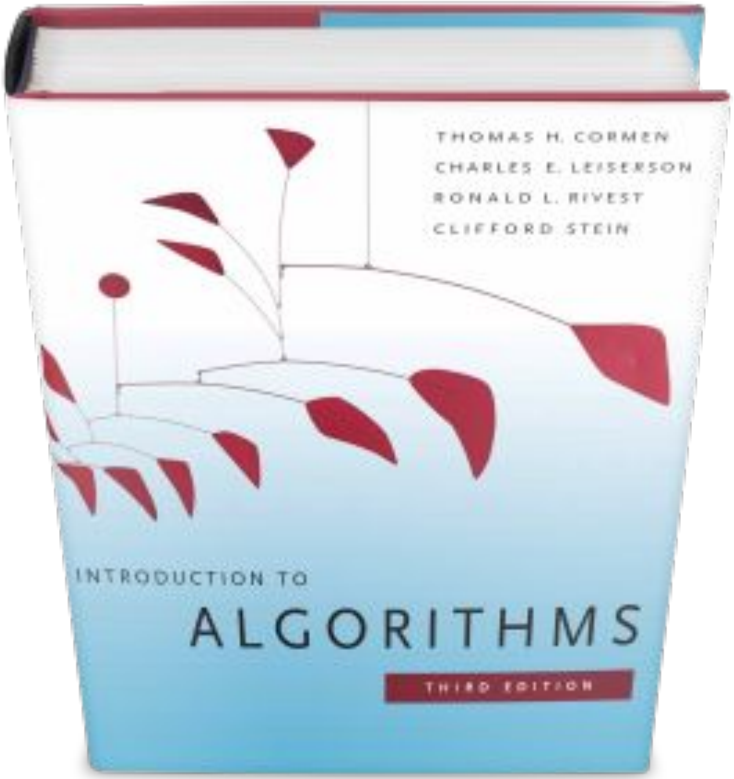
Most programming languages offer some kind of associative “arrays” or containers. They may be called differently: maps, dictionaries, hash-maps, unordered-maps, hash-tables, etc.

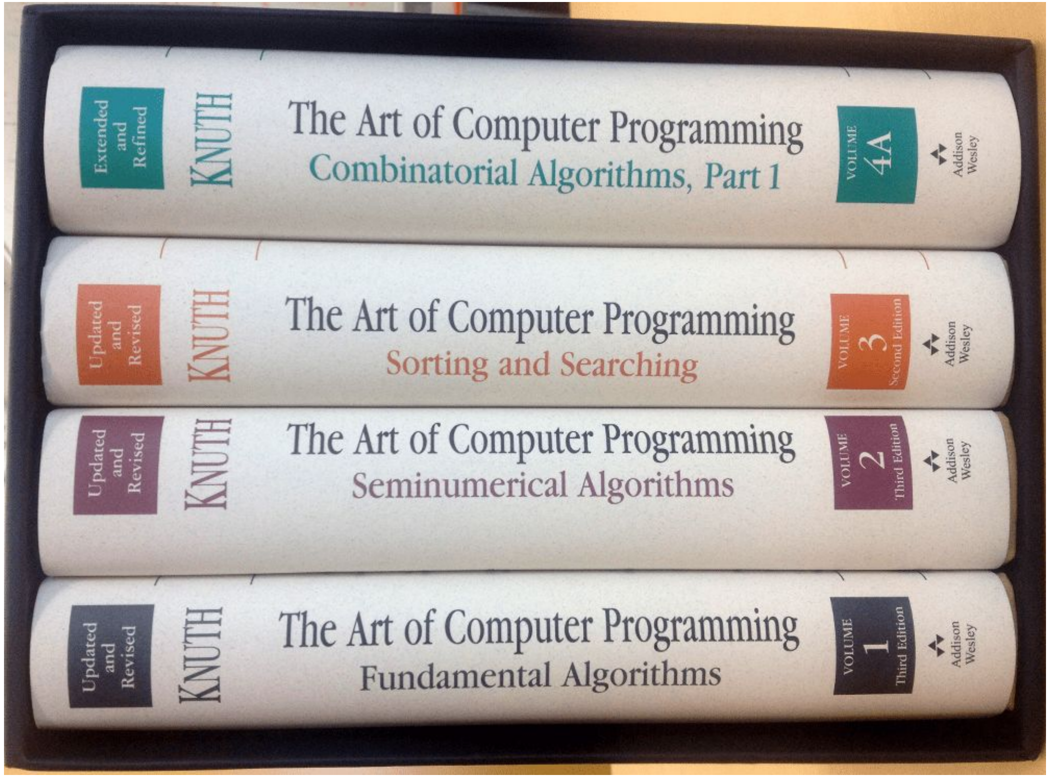
If you’ve never heard of them, this workshop is for you. If you’ve seen them before, but you’re not sure which one to use for a particular task, this workshop is for you. If you’re confident in using such fast lookup structures, great! But you’ll still be surprised by the details we’re going to cover in this lecture.

After this journey from the very basics of lookup data structures up to advanced hashing techniques, you’ll feel more confident when & how to use them effectively.

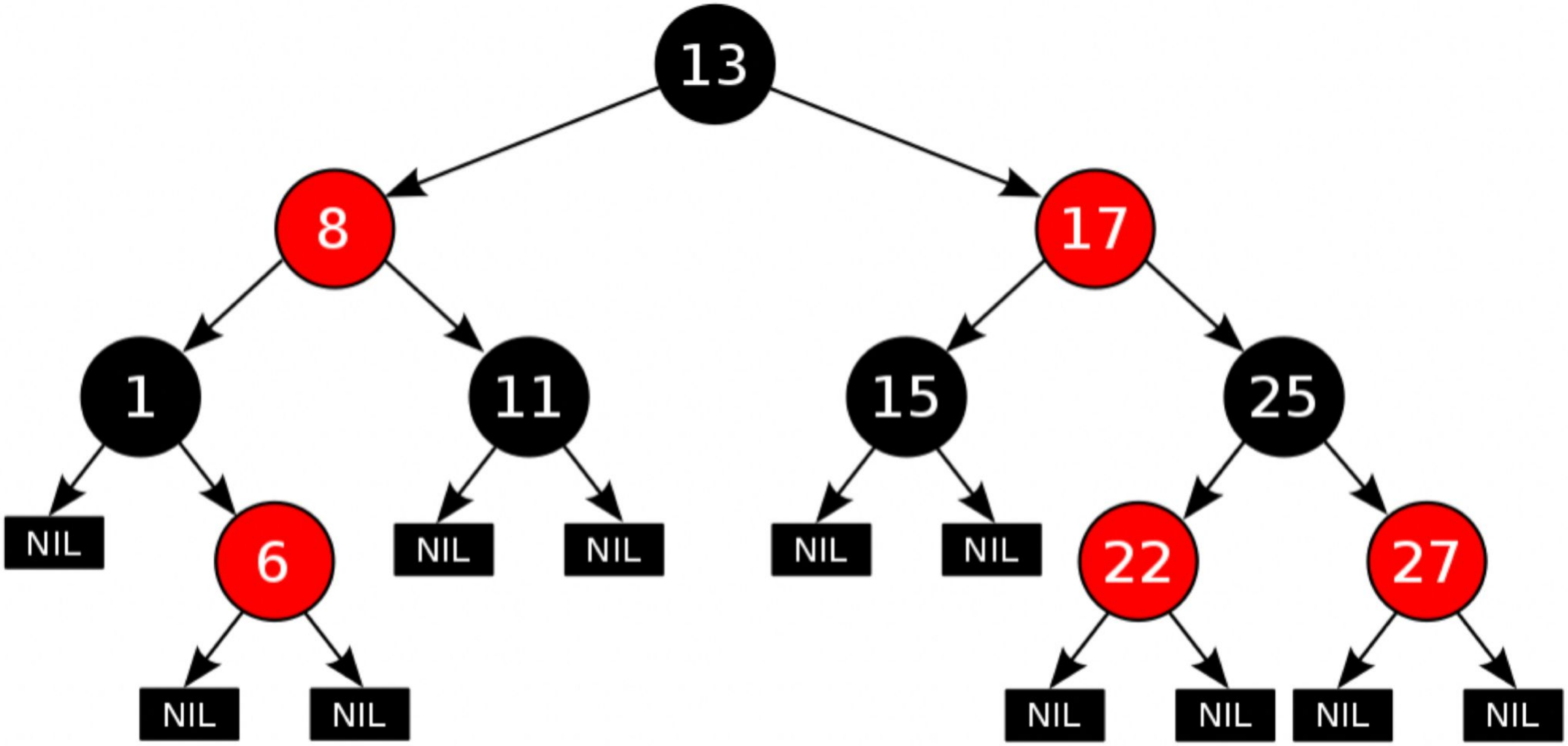
This workshop assumes familiarity with the C++ language, as we'll focus on hashing facilities in the standard library and beyond. We'll take a deep dive into hashing algorithms and hashed data structures, both in design and examples.

🎓 What they teach you





🎓 What they teach you



What they teach you

```
initial call to sort entire array Quicksort(A, 1, length[A])

Quicksort (A, p, r)
if p < r
   then q = Partition(A, p, r)
      Quicksort(A, p, q)
      Quicksort(A, q+1, r)

Partition (A, p, r)
   x = A[p]
   i = p-1
   j = r+1
   while TRUE
   do repeat j = j-1
         until A[j] <= x
      repeat i = i+1
         until A[i] >= x
      if i < j
         then exchange A[i], A[j]
         else return j
```

O(n log n)

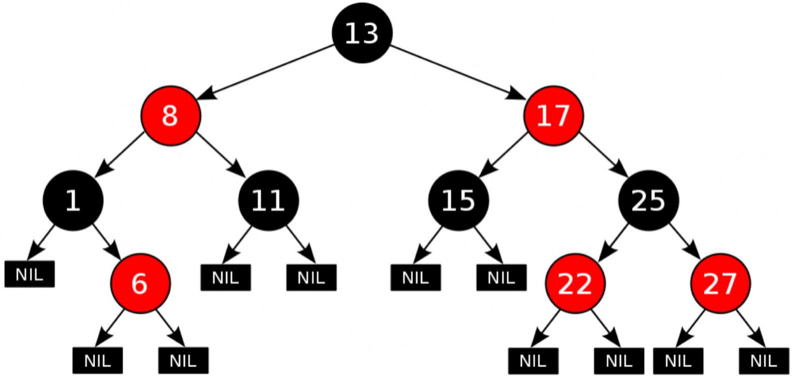
The Big-O

Algorithm	Data structure	Time complexity:Best	Time complexity:Average	Time complexity:Worst	Space complexity:Worst
Quick sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(n)$
Merge sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Heap sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Smooth sort	Array	$O(n)$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bogo sort	Array	$O(n)$	$O(n\ n!)$	$O(\infty)$	$O(1)$

wikipedia.org/wiki/Computational_complexity_theory

What about Data Structures ? 📦

Data structures along with the operations they provide, also have complexity guarantees



STL Containers Big-O cheat-sheet

C++ STL	insert @end	insert @pos	erase @end	erase @pos	find	sort	iterator	comment
vector	O(1)	O(dist(pos,end))	O(1)	O(dist(pos,end))	O(n)	O(n*log(n))	RandomAccess	array
dequeue	@begin/@end O(1)	O(dist(pos,begin/end))	@begin/@end O(1)	O(dist(pos,begin/end))	O(n)	O(n*log(n))	RandomAccess	
list	O(1)	O(1)	O(1)	@pos O(1); @key O(n)	O(n)	O(n*log(n))	Bidirectional	doubly linked
stack	O(1) push()	-	O(1) pop()	-	O(n)	-	same as container	adaptor<dequeue, list, vector>
queue	O(1) push()	-	O(1) pop() @begin	-	O(n)	-	same as container	adaptor<dequeue, list>
set/map	-	O(log(n))	-	@pos O(1); @key O(log(n)+count(key))	O(log(n))	sorted	Bidirectional	red-black tree (balanced BST)
unordered_set/ unordered_map	-	avg O(1); worst O(n)	-	@pos avg O(1) worst O(n); @key O(count(key))	avg O(1); worst O(n)	-	Forward	hash_set/hash_map
priority_queue	push() O(log(n))	-	pop() O(log(n))	-	top() O(1)	-	RandomAccess	adaptor<vector, dequeue> => constant time extraction of the largest (default) element, at the expense of logarithmic insertion
make_heap(range)	push_heap() O(2*log(n))	-	pop_heap() O(2*log(n))	-	max is first	O(n*log(n))	RandomAccess	constructs a max heap in the range

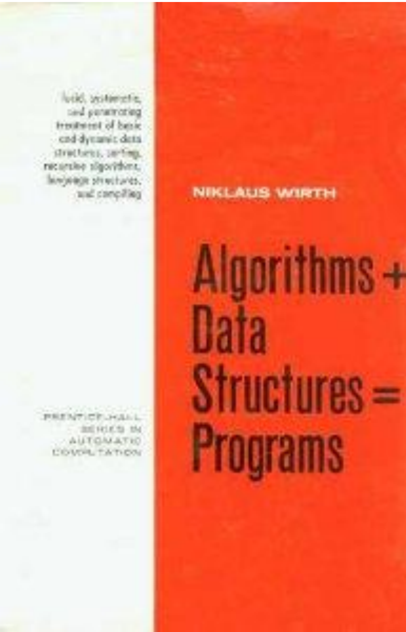
The difference between **Efficiency** and **Performance**

Why do we care ?

Because: *"Software is getting slower more rapidly than hardware becomes faster."*

"A Plea for Lean Software" - Niklaus Wirth

Efficiency	**Performance**
the amount of work you need to do	how fast you can do that work
governed by your algorithm	governed by your data structures



ℹ️ **Efficiency** and **performance** are *not necessarily dependant* on one another

What about Performance ? 🚀

How **fast** can the CPU execute **each step** from the algorithms.

This is mostly determined by the native (CPU) **data types** used

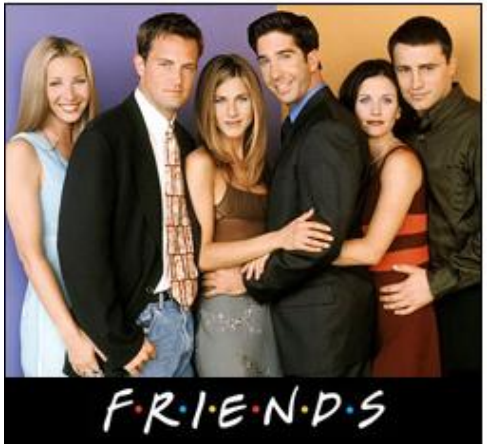
and your choice of **data structures**.

🎓 What they teach you

That's all great and still relevant,

but...

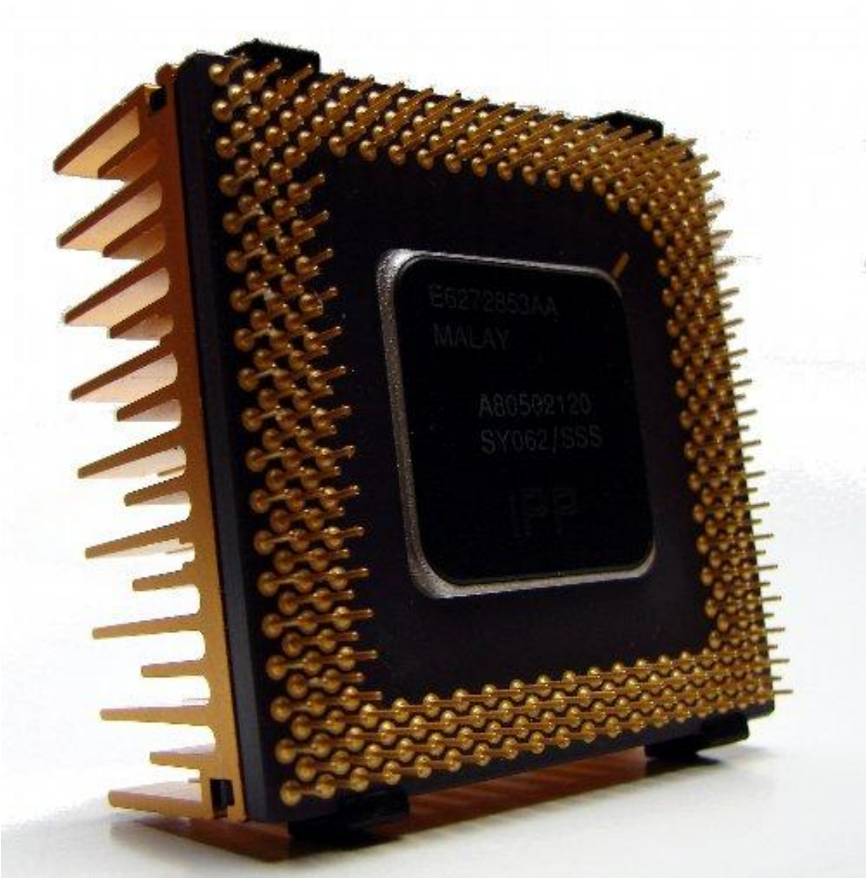
The 90s happened...



Seinfeld

The 90s happened...

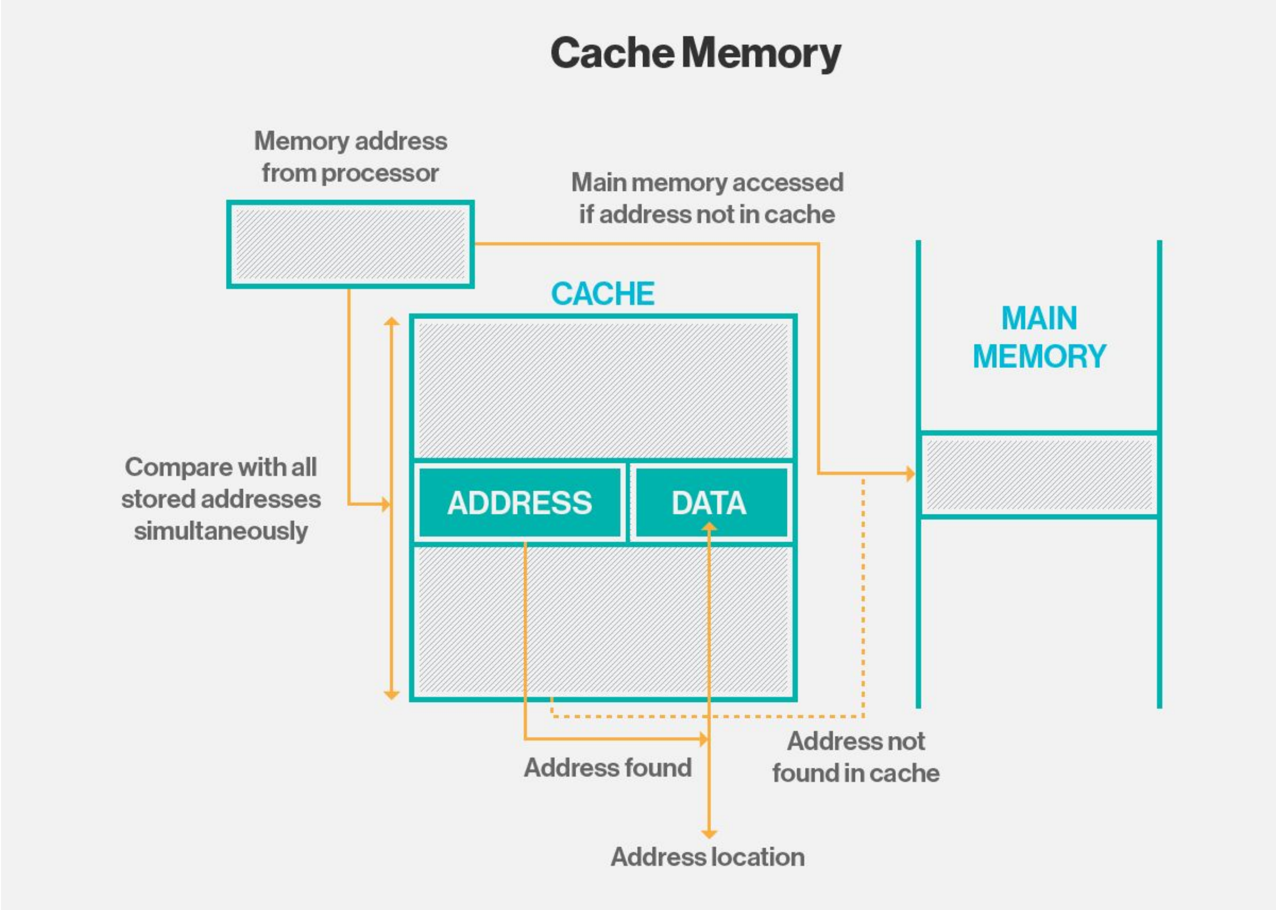
The 90s happened...



The 90s happened...

Cache

The 90s happened...

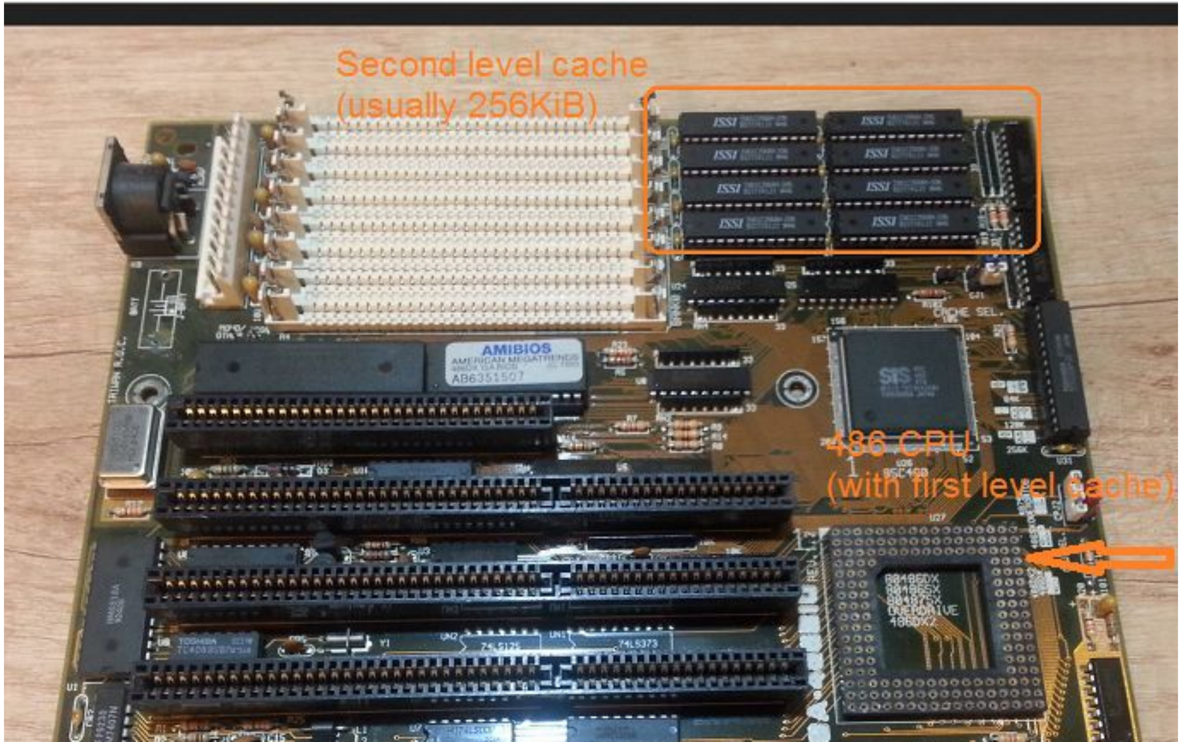


The 90s happened...

80486 (1989)

This is the first CPU of this generation which has some **cache** on the CPU.

It is a 8KB **unified** cache which means it is used for *data* and *instructions*.



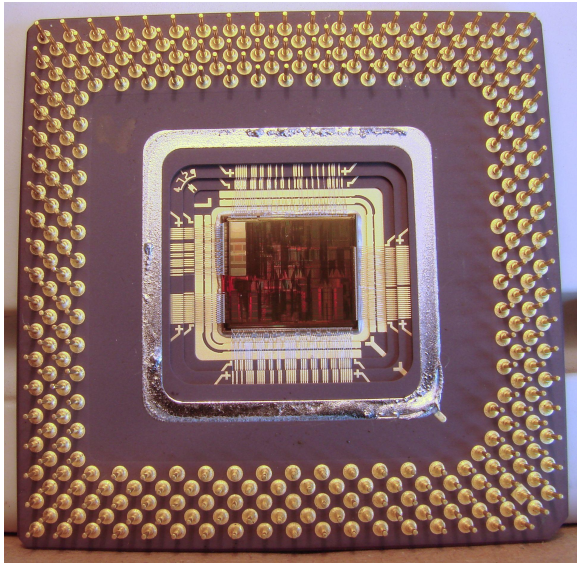
The 90s happened...

`80586` (1993)

The 586 or Pentium-1 uses a a **split** level 1 cache.
8 KB each for data and instructions.

The cache was split so that the *data* and *instruction* caches could be individually tuned for their specific use.

You still have a small yet very fast 1st cache near the CPU, and a larger but slower 2nd cache on the *motherboard*.



The 90s happened...

80686 (1995)

The 686 or Pentium Pro chip, depending on the model, had a 256Kb, 512KB or 1MB on board cache.

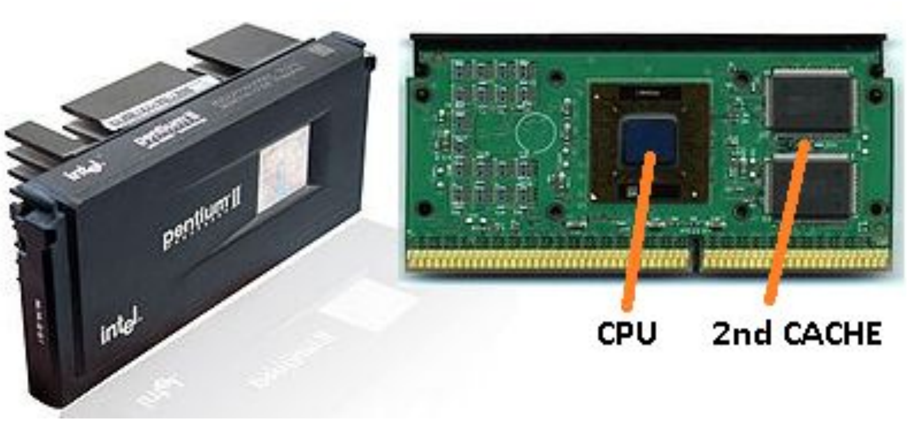
Half the space in the chip is used by the cache.

The 90s happened...

Pentium 2 (1997)

For economy reasons the 2nd cache is **not** in the CPU.

CPU package is on a PCB with separate chips for:
- CPU (and 1st cache)
- 2nd cache

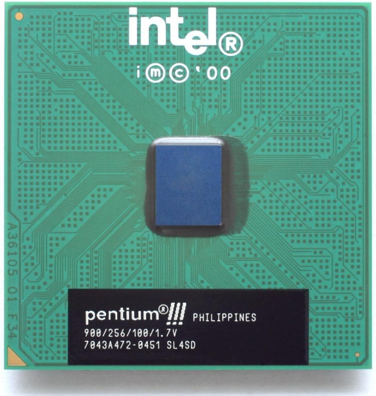


The 90s happened...

`Pentium III` (1999)
`Pentium 4` (2000)

As technology progresses and we start put create chips with smaller components it gets financially possible to put the 2nd cache back in the actual CPU die.

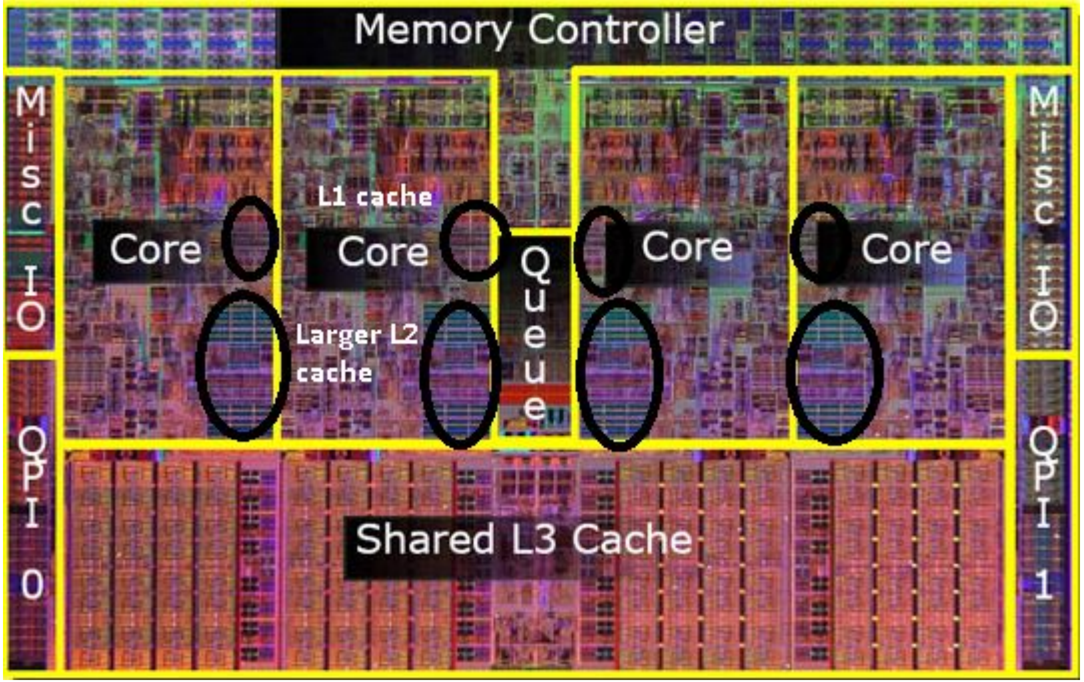
However there is still a split:
Very fast 1st cache snuggled up to the CPU. With one 1st cache per CPU core and a larger but less fast 2nd cache next to the core.



L1 and L2 caches not enough…

=> **L3 cache**

Nehalem 1st gen Core i7 series (2008)



Cache Latency

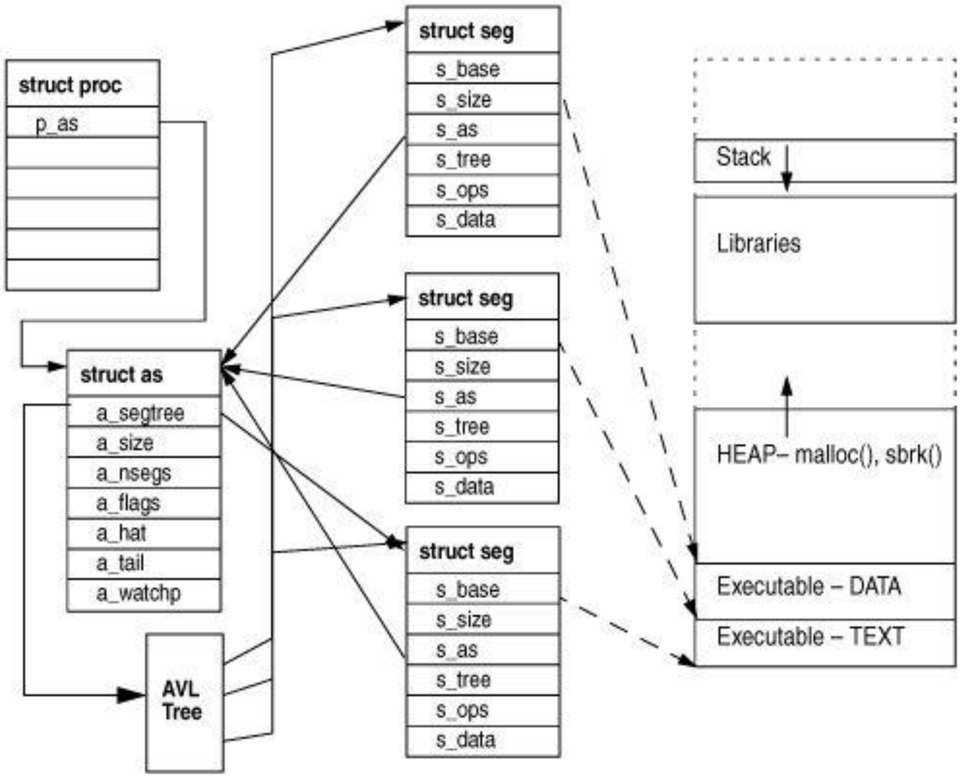
```
Core i7 Xeon 5500 Data Source Latency

local  L1 CACHE hit,                              ~4 cycles (   2.1 -  1.2 ns )
local  L2 CACHE hit,                             ~10 cycles (   5.3 -  3.0 ns )
local  L3 CACHE hit, line unshared               ~40 cycles (  21.4 - 12.0 ns )
local  L3 CACHE hit, shared line in another core ~65 cycles (  34.8 - 19.5 ns )
local  L3 CACHE hit, modified in another core    ~75 cycles (  40.2 - 22.5 ns )

local  DRAM                                                   ~60 ns
remote DRAM                                                  ~100 ns
```

- Memory Layout
- Memory Access Patterns

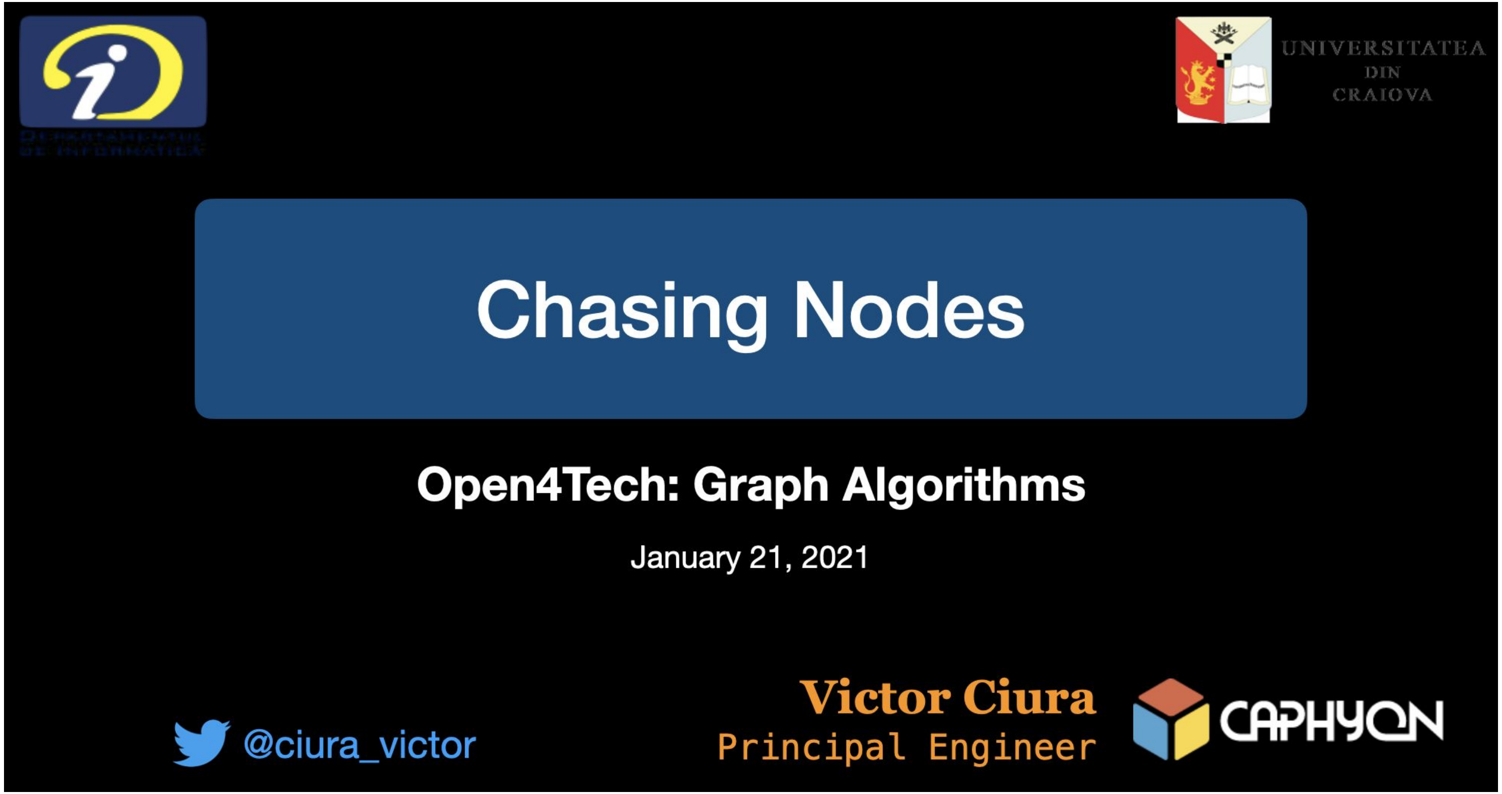
Container access patterns

Jumping around through memory,

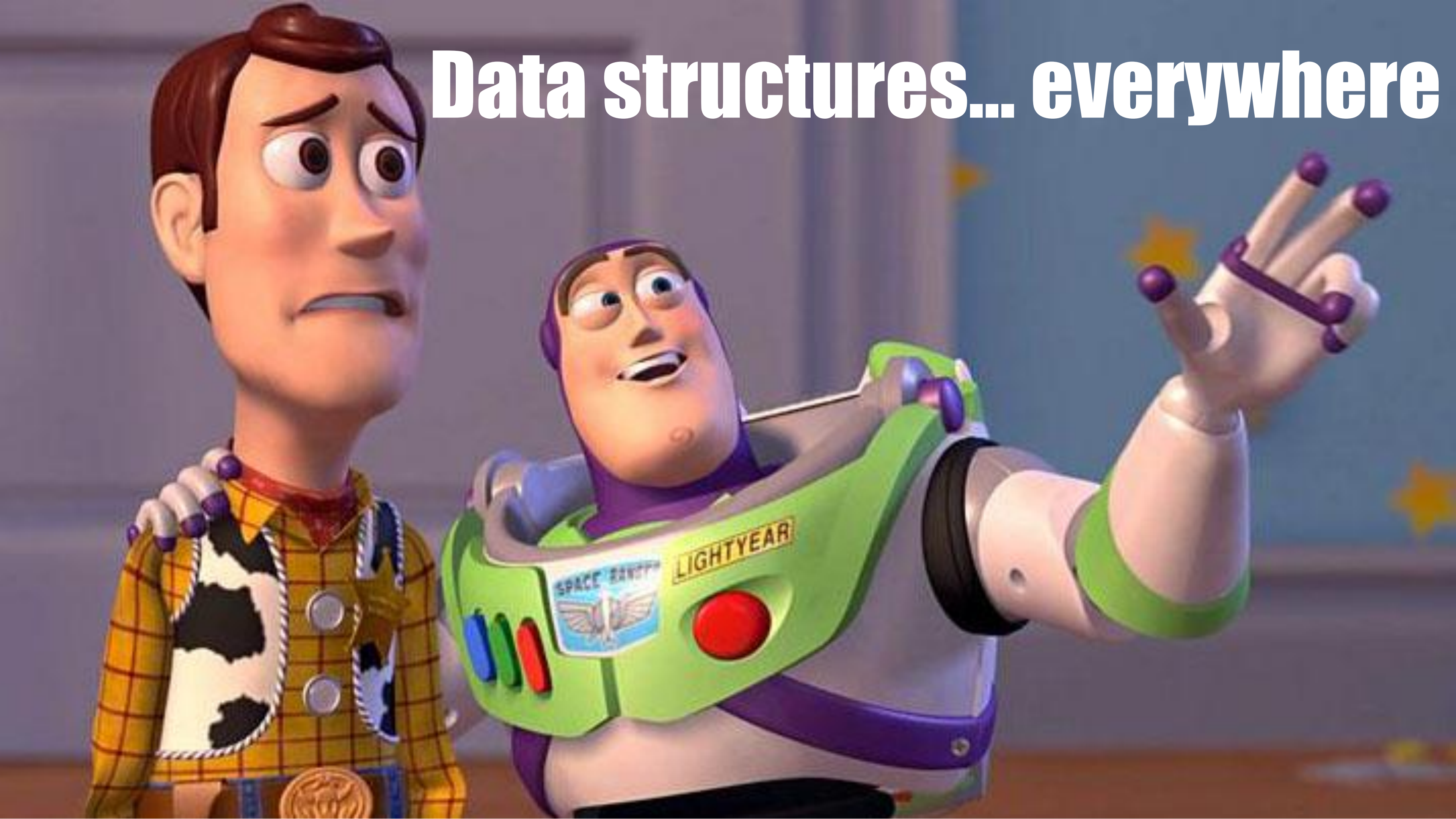
chasing pointers...



Further Study



ciura.ro/presentations/2021/Open4Tech/ChasingNodes.pdf


Data structures... everywhere
LIGHTYEAR

STL

array
vector
deque
forward_list
list

set
map
multiset
multimap

unordered_set
unordered_map
unordered_multiset
unordered_multimap

stack
queue
priority_queue

cppreference.com/w/cpp/container

A List<> Of Data Structures You Should Add In Your Learning Queue<>

In this workshop we'll explain the mechanics behind data structures, and we'll deep dive into the meaning and usage of the most common ones.

Let's try together to get some insights in some data structures and their pitfalls.

We'll discover, by lots of examples, the strengths and weaknesses of each data structure and find good use cases for:

- Vector and List
- Stack and Queue
- Hash Table
- Graphs and Trees
- Heap.

Difficulty: *intermediate*

Format: *2 days x 2h*

Trainer: Nicolae Telechi

Caphyon, Senior Software Developer

	Luni	Marti	Miercuri	Joi	Vineri
	28 iunie	29 iunie	30 iunie	1 iulie	2 iulie
2-4pm	So You Think You Can # (hashing algorithms & containers)	So You Think You Can # (hashing algorithms & containers)	A list<> of data structures you should add in your learning queue<>	A list<> of data structures you should add in your learning queue<>	Building an app using Blockchain
4-6pm	Why we code	Why we code	Getting a11y right	Java Swing Crash Course	

www.caphyon.ro/open4tech-2021-a-list-of-data-structures-you-should-add-in-your-learning-queue/

90% of situations, a great choice*

`std::array`

`std::vector`

* when performance matters

Today’s focus

Most programming languages offer some kind of **associative** “arrays” or containers.

They may be called differently:

- maps
- dictionaries
- hash-maps
- unordered-maps
- hash-tables

Today's focus

unordered_set

unordered_map

unordered_multiset

unordered_multimap

Hash Functions & Hash Tables

A hash function is any function that can be used to map data of *arbitrary size* to data of ***fixed size*** **(hash code)**.

Hash functions are used in **hash tables**, to quickly locate a data record given its **search key**.

Hash Functions & Hash Tables

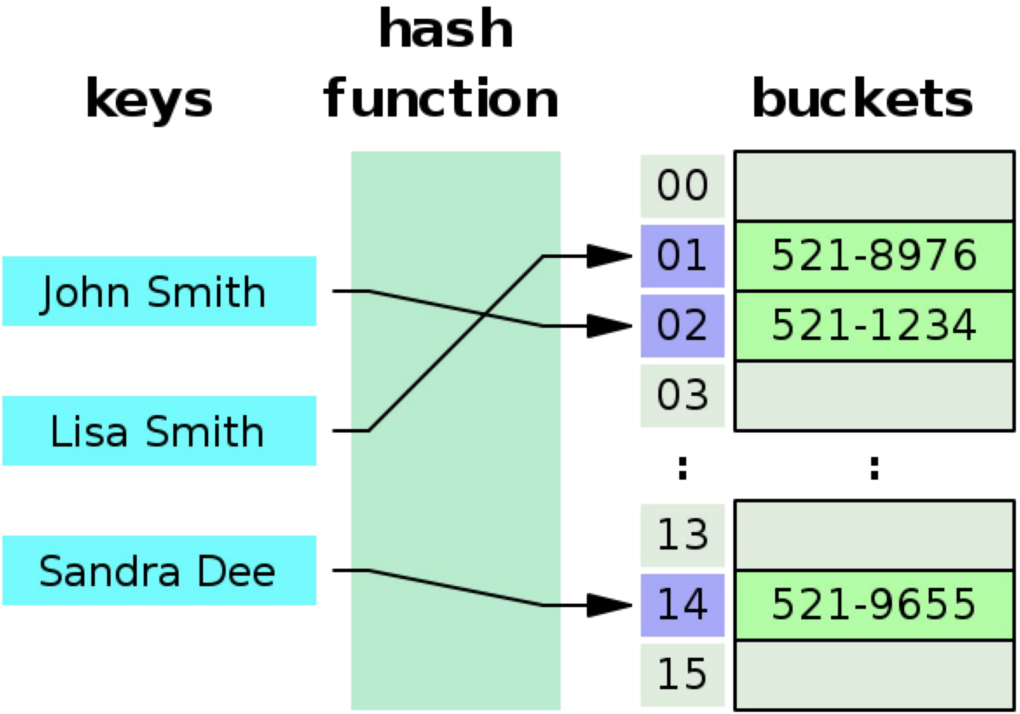
The hash function is used to map the search key to an **index**; the index gives the place in the hash table where the corresponding record should be stored/found.

The **domain** of a hash function (the set of possible keys) is larger than its **range** (the number of different table indices), and so it will map several <u>different keys</u> to the <u>same index</u>.

Hash Functions & Hash Tables

Each slot (**bucket**) of a hash table is associated with a set of records, rather than a single record.

Visualize it



Hash Function Properties

❖ **Determinism**

A hash procedure must be deterministic – meaning that for a given input value it must always generate the <u>same hash</u> value.

Hash Function Properties

- **Uniformity**

    A good hash function should map the expected inputs as evenly as possible over its output range.

    That is, every hash value in the output range should be generated with roughly the <u>same probability</u>.

Hash Function Properties

❖ **Defined range**

It is often desirable that the output of a hash function have fixed size.

If, for example, the output is constrained to 32-bit integer values, the hash values can be used to index into an array (eg. hash tables).

Hash Function Properties

❖ **Non-invertible**

In *cryptographic* applications, hash functions are typically expected to be practically non-invertible, meaning that it is not realistic to <u>reconstruct</u> the input datum from its hash value alone, without spending great amounts of computing time.

Questions

- How should one <u>combine</u> hash codes from your bases and data members to create a “good” hash function?

- How does one know if you have a good hash function?

- If somehow you knew you had a bad hash function, how would you change it for a type built out of several bases and/or data members?

How does one hash this class?

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
  // ...
};
```

std::hash<Key>

Defined in header <functional>

```
std::size_t h = std::hash<std::string>{}(firstName);
```

- Accepts a single parameter of type Key
- Returns a value of type size_t that represents the hash value of the parameter
- Does not throw exceptions when called
- If k1 and k2 are equal => std::hash<Key>()(k1) == std::hash<Key>()(k2)
- If k1 and k2 are different, the ***probability*** that std::hash<Key>()(k1) == std::hash<Key>()(k2) should be very small, approaching 1.0/std::numeric_limits<size_t>::max()

std::hash<Key>

Standard specializations for ***basic*** types:

```
template< class T > struct hash<T*>;

template<> struct hash<bool>;
template<> struct hash<char>;
template<> struct hash<signed char>;
template<> struct hash<unsigned char>;
template<> struct hash<char16_t>;
template<> struct hash<char32_t>;
template<> struct hash<wchar_t>;
template<> struct hash<short>;
template<> struct hash<unsigned short>;
template<> struct hash<int>;
template<> struct hash<unsigned int>;
template<> struct hash<long>;
template<> struct hash<long long>;
template<> struct hash<unsigned long>;
template<> struct hash<unsigned long long>;
template<> struct hash<float>;
template<> struct hash<double>;
template<> struct hash<long double>;
```

Standard specializations for ***library*** types:

```
std::hash<std::string>
std::hash<std::wstring>
std::hash<std::unique_ptr>
std::hash<std::shared_ptr>
std::hash<std::bitset>
//...
```

```cpp
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
// ...
  std::size_t hash_code() const
  {
    std::size_t k1 = std::hash<std::string>{}(firstName);
    std::size_t k2 = std::hash<std::string>{}(lastName);
    std::size_t k3 = std::hash<int>{}(age);

    return hash_combine(k1, k2, k3);  // what algorithm is this?
  }
};
```

Is this a good hash strategy?

What if we wanted to use another hash algorithm?

boost::hash_combine

```
template <class T>
inline void hash_combine(std::size_t & seed, const T & v)
{
    std::hash<T> hasher;
    seed ^= hasher(v) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}
```

The magic number is supposed to be 32 “random” bits:
- each is equally likely to be 0 or 1
- with no simple correlation between the bits

A common way to find a pattern of such bits is to use the binary expansion of an *irrational number*.

In this case, that number is the reciprocal of the **golden ratio**:

φ = (1 + sqrt(5)) / 2
2^32 / φ = **0x9e3779b9**

http://stackoverflow.com/questions/35985960/c-why-is-boosthash-combine-the-best-way-to-combine-hash-values

FNV-1A

```
std::size_t fnv1a(void const * key, std::size_t len)
{
  std::size_t h = 14695981039346656037u;

  unsigned char const * p = static_cast<unsigned char const*>(key);
  unsigned char const * const e = p + len;
  for (; p < e; ++p)
    h = (h ^ *p) * 1099511628211u;

  return h;
}
```

The FNV hash was designed for fast hash-table and checksum use (not cryptography).

https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function

Hash with FNV-1A

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
// ...
  std::size_t hash_code() const
  {
    std::size_t k1 = fnv1a(firstName.data(), firstName.size());
    std::size_t k2 = fnv1a(lastName.data(), lastName.size());
    std::size_t k3 = fnv1a(&age, sizeof(age));

    return hash_combine(k1, k2, k3);  // what algorithm is this?
  }
};
```

Ok, but our algorithm is still “polluted” by the combine step...

Anatomy Of A Hash Function

1. Initialize internal state.
2. Consume bytes into internal state.
3. Finalize internal state to result_type (usually size_t).

Anatomy Of A Hash Function

```
std::size_t fnv1a(void const * key, std::size_t len)
{
  std::size_t h = 14695981039346656037u; ⇐ initialize internal state

  // consume bytes into internal state:
  unsigned char const * p = static_cast<unsigned char const*>(key);
  unsigned char const * const e = p + len;
  for (; p < e; ++p)
    h = (h ^ *p) * 1099511628211u;

  return h; ⇐ finalize internal state to size_t
}
```

Repackaging this algorithm
to make the three stages separately accessible

```
class fnv1a
{
  std::size_t h = 14695981039346656037u;     ⇐ initialize internal state
public:

  // consume bytes into internal state
  void operator()(void const * key, std::size_t len) noexcept
  {
    unsigned char const * p = static_cast<unsigned char const*>(key);
    unsigned char const * const e = p + len;
    for (; p < e; ++p)
      h = (h ^ *p) * 1099511628211u;
  }

  explicit operator size_t() noexcept   ⇐ finalize internal state to size_t
  {
    return h;
  }
};
```

```cpp
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  std::size_t hash_code() const
  {
    fnv1a hasher;

    hasher(firstName.data(), firstName.size());
    hasher(lastName.data(), lastName.size());
    hasher(&age, sizeof(age));

    return static_cast<std::size_t>(hasher); // no more hash_combine() !!!
  }
};
```

🔶 The same technique can be used with almost every existing hashing algorithm.

Now we are using a “pure” FNV-1A algorithm for the entire data structure.

Combining Types

```
class Sale
{
  Customer customer;
  Product  product;
  Date     date;

public:

  std::size_t hash_code() const
  {
    std::size_t h1 = customer.hash_code();
    std::size_t h2 = product.hash_code();
    std::size_t h3 = date.hash_code();

    return hash_combine(h1, h2, h3); // OMG, it’s back :(
  }
};
```

How do we use just FNV-1A for the entire class?

hash_append()

Proposal by:

Howard Hinnant, Vinnie Falco, John Bytheway

N3980 / 2014-05-24

hash_append()

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  std::size_t hash_code() const
  {
    fnv1a hasher;

    hasher(firstName.data(), firstName.size());
    hasher(lastName.data(), lastName.size());
    hasher(&age, sizeof(age));

    return static_cast<std::size_t>(hasher); // no more hash_combine() !!!
  }
};
```

hash_append()

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  friend void hash_append(fnv1a & hasher, const Customer & c)
  {
    hasher(c.firstName.data(), c.firstName.size());
    hasher(c.lastName.data(), c.lastName.size());
    hasher(&c.age, sizeof(c.age));
  }
};
```

Let some other piece of code *construct* and *finalize* fnv1a.
Customer only *appends* to the state of fnv1a.

hash_append()

```
class Sale
{
  Customer customer;
  Product  product;
  Date     date;

public:

  friend void hash_append(fnv1a & hasher, const Sale & s)
  {
    hash_append(hasher, s.customer);
    hash_append(hasher, s.product);
    hash_append(hasher, s.date);
  }

};
```

Types can recursively build upon one another's hash_append() to build up state in fnv1a object.

hash_append()

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  friend void hash_append(fnv1a & hasher, const Customer & c)
  {
    hash_append(hasher, c.firstName);
    hash_append(hasher, c.lastName);
    hash_append(hasher, c.age);
  }
};
```

Primitive and std-defined types can be given hash_append() overloads
=> simplified & uniform interface

hash_append() / Abstracting the algorithm

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  template<class HashAlgorithm>
  friend void hash_append(HashAlgorithm & hasher, const Customer & c)
  {
    hash_append(hasher, c.firstName);
    hash_append(hasher, c.lastName);
    hash_append(hasher, c.age);
  }
};
```

If all hash algorithms use a *uniform interface*, we can swap any hasher into our data type.

hash_append() / Primitives

For **primitive types** that are ***contiguously hashable***
we can just send their bytes to the hash algorithm in hash_append().

Eg.

```
template <class HashAlgorithm>
void hash_append(HashAlgorithm & hasher, int i)
{
  hasher(&i, sizeof(i));
}

template <class HashAlgorithm, class T>
void hash_append(HashAlgorithm & hasher, T * p)
{
  hasher(&p, sizeof(p));
}
```

hash_append()

A complicated class is ultimately made up of **scalars** located in discontiguous memory.

hash_append() appends each byte to the HashAlgorithm state by ***recursing*** down into the data structure to find the scalars.

Prerequisites:

- Every type has a hash_append() overload
- The overload will either call hash_append() on its bases and members, or it will send bytes of its memory to the HashAlgorithm
- No type is aware of the concrete HashAlgorithm type.

How to use hash_append()

```
HashAlgorithm hasher;

hash_append(hasher, my_type);

return static_cast<size_t>(hasher);
```

Wrap the whole thing up in a conforming hash functor

```
template <class HashAlgorithm>
struct GenericHash
{
  using result_type = typename HashAlgorithm::result_type;

  template <class T>
  result_type operator()(const T & t) const noexcept
  {
    HashAlgorithm hasher;
    hash_append(hasher, t);
    return static_cast<result_type>(hasher);
  }
};
```

```
unordered_set<Customer, GenericHash<fnv1a>> my_set;
```

Change Hashing Algorithms

```
unordered_set<Sale, GenericHash<fnv1a>> my_set;

unordered_set<Sale, GenericHash<SipHash>> my_set;

unordered_set<Sale, GenericHash<Spooky>> my_set;

unordered_set<Sale, GenericHash<Murmur>> my_set;

unordered_set<Sale, GenericHash<CityHash>> my_set;
```

It becomes trivial to <u>experiment</u> with different hashing algorithms to optimize performance, minimize collisions.

Open4Tech Summer School 2021

So You Think You Can #
Building an app using Blockchain
A list< > of data structures you should add in your learning queue< >
Build a video call react app with WebRTC and Socket.io
Importance of good coding habits

Build cryptocurrencies exchange using React
Why we code
Getting a11y right
State of the Art Natural Language Processing for Noobs
Java Swing Crash Course

28 iunie - 16 iulie 2021
http://inf.ucv.ro/~ summer-school/

DEPARTAMENTUL DE INFORMATICĂ

CAPHYON

NETROM premium software

syncro soft

Society for Computing Technologies

	Luni	Marti	Miercuri	Joi	Vineri
	28 iunie	29 iunie	30 iunie	1 iulie	2 iulie
2-4pm	So You Think You Can # (hashing algorithms & containers)	So You Think You Can # (hashing algorithms & containers)	A list<> of data structures you should add in your learning queue<>	A list<> of data structures you should add in your learning queue<>	Building an app using Blockchain
4-6pm	Why we code	Why we code	Getting a11y right	Java Swing Crash Course	
	5 iulie	6 iulie	7 iulie	8 iulie	9 iulie
2-4pm	Build a video call react app with WebRTC and Socket.io	Build a video call react app with WebRTC and Socket.io	Build a video call react app with WebRTC and Socket.io	State of the Art Natural Language Processing for Noobs	State of the Art Natural Language Processing for Noobs
4-6pm					
	12 iulie	13 iulie	14 iulie	15 iulie	16 iulie
2-4pm	Build cryptocurrencies exchange using React	Build cryptocurrencies exchange using React	Code conventions: Importance of good coding habits	Code conventions: Importance of good coding habits	
4-6pm					

Part II

So You Think You Can

Hashing Algorithms and Containers

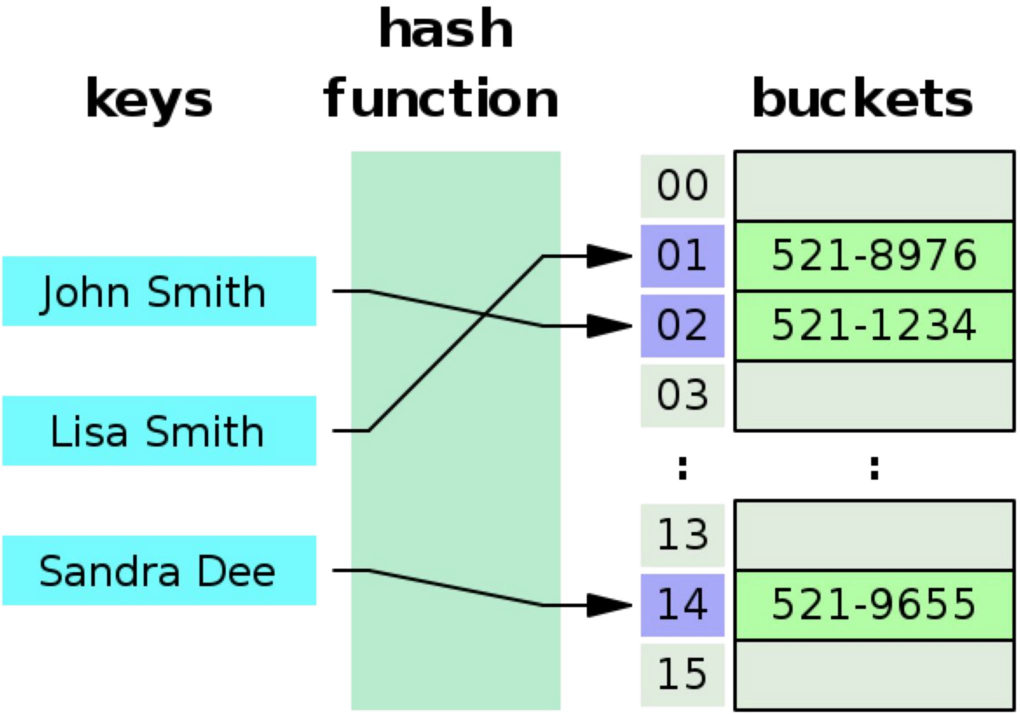
inf.ucv.ro/~summer-school

Victor Ciura
Principal Engineer

@ciura_victor

June 2021

Recap from **Part 1**



Hash Function Properties

- Determinism
- Uniformity
- Defined range
- Non-invertible

Anatomy Of A Hash Function

1. Initialize internal state.
2. Consume bytes into internal state.
3. Finalize internal state to result_type (usually size_t).

Repacking this algorithm to make the three stages separately accessible

```
class fnv1a
{
  std::size_t h = 14695981039346656037u;     ⇐ initialize internal state
public:

  // consume bytes into internal state
  void operator()(void const * key, std::size_t len) noexcept
  {
    unsigned char const * p = static_cast<unsigned char const*>(key);
    unsigned char const * const e = p + len;
    for (; p < e; ++p)
      h = (h ^ *p) * 1099511628211u;
  }

  explicit operator size_t() noexcept   ⇐ finalize internal state to size_t
  {
    return h;
  }
};
```

hash_append()

```
class Customer
{
  std::string firstName;
  std::string lastName;
  int         age;
public:
  // ...

  friend void hash_append(fnv1a & hasher, const Customer & c)
  {
    hasher(c.firstName.data(), c.firstName.size());
    hasher(c.lastName.data(), c.lastName.size());
    hasher(&c.age, sizeof(c.age));
  }
};
```

Let some other piece of code *construct* and *finalize* fnv1a.
Customer only *appends* to the state of fnv1a.

hash_append()

```
class Sale
{
  Customer customer;
  Product  product;
  Date     date;

public:

  friend void hash_append(fnv1a & hasher, const Sale & s)
  {
    hash_append(hasher, s.customer);
    hash_append(hasher, s.product);
    hash_append(hasher, s.date);
  }

};
```

Types can recursively build upon one another's hash_append() to build up state in fnv1a object.

Wrap the whole thing up in a conforming hash functor

```
template <class HashAlgorithm>
struct GenericHash
{
  using result_type = typename HashAlgorithm::result_type;

  template <class T>
  result_type operator()(const T & t) const noexcept
  {
    HashAlgorithm hasher;
    hash_append(hasher, t);
    return static_cast<result_type>(hasher);
  }
};
```

```
std::unordered_set<Customer, GenericHash<fnv1a>> my_set;
```

Change Hashing Algorithms

```
unordered_set<Sale, GenericHash<fnv1a>> my_set;

unordered_set<Sale, GenericHash<SipHash>> my_set;

unordered_set<Sale, GenericHash<Spooky>> my_set;

unordered_set<Sale, GenericHash<Murmur>> my_set;

unordered_set<Sale, GenericHash<CityHash>> my_set;
```

It becomes trivial to <u>experiment</u> with different hashing algorithms to optimize performance, minimize collisions.

std::hash<Key>

Defined in header <functional>

```
std::size_t h = std::hash<std::string>{}(firstName);
```

- Accepts a single parameter of type Key
- Returns a value of type size_t that represents the hash value of the parameter
- Does not throw exceptions when called
- If k1 and k2 are equal => std::hash<Key>()(k1) == std::hash<Key>()(k2)
- If k1 and k2 are different, the ***probability*** that std::hash<Key>()(k1) == std::hash<Key>()(k2) should be very small, approaching 1.0/std::numeric_limits<size_t>::max()

std::hash<Key>

Standard specializations for ***basic*** types:

```
template< class T > struct hash<T*>;

template<> struct hash<bool>;
template<> struct hash<char>;
template<> struct hash<signed char>;
template<> struct hash<unsigned char>;
template<> struct hash<char16_t>;
template<> struct hash<char32_t>;
template<> struct hash<wchar_t>;
template<> struct hash<short>;
template<> struct hash<unsigned short>;
template<> struct hash<int>;
template<> struct hash<unsigned int>;
template<> struct hash<long>;
template<> struct hash<long long>;
template<> struct hash<unsigned long>;
template<> struct hash<unsigned long long>;
template<> struct hash<float>;
template<> struct hash<double>;
template<> struct hash<long double>;
```

Standard specializations for ***library*** types:

```
std::hash<std::string>
std::hash<std::wstring>
std::hash<std::unique_ptr>
std::hash<std::shared_ptr>
std::hash<std::bitset>

//...
```

</Recap>

`std::hash<Key>`

Standard specializations for ***library*** types:

```
std::hash<std::string>
std::hash<std::wstring>

std::hash<std::unique_ptr>
std::hash<std::shared_ptr>

std::hash<std::bitset>

//...
```

Exploring string hash tables
<code walk-through>

What we want:

"A hash table mapping string keys (case-insensitive) to some custom data type."

Starting point:

```
template<
  class Key,
  class T,
  class Hash     = std::hash<Key>,
  class KeyEqual = std::equal_to<Key>, ⇐ Why is this part of the interface? Why not Key::operator==()
  class Alloc    = std::allocator< std::pair<const Key, T> >
>
class std::unordered_map;
```

What we need:

- A custom hash functor for case-insensitive strings
- A custom comparator functor, to compare strings ignoring character case

Exploring string hash tables
<code walk-through>

```
template <class Type, class StringType = std::basic_string<Type>>
struct BasicStringHash
{
  using HashedType = StringType;

  size_t operator()(const HashedType & aStr) const
  {
    std::hash<HashedType> hasher;  ⇐ we can use any hashing algorithm
    return hasher(aStr);
  }

  bool operator()(const HashedType & aStr1, const HashedType & aStr2) const
  {
    return aStr1 < aStr2;
  }

  struct KeyEquality
  {
    bool operator()(const HashedType & aStr1, const HashedType & aStr2) const
    {
      return aStr1 == aStr2;
    }
  };
};
```

Exploring string hash tables
<code walk-through>

```
typedef BasicStringHash<char>     StringHash;
typedef BasicStringHash<wchar_t>  StringHashW;
```

Eg.

```
        std::unordered_map<wstring, TYPE, StringHashW, StringHashW::KeyEquality>
```

Exploring string hash tables
<code walk-through>

Case-insensitive hashes

```
template <class Type, class StringType = std::basic_string<Type>>
struct BasicStringHashI
{
  using HashedType = StringType;

  size_t operator()(const HashedType & aStr) const
  {
    // make a lower-case copy of the input string
    HashedType lowerStr(aStr);
    ToLower(const_cast<Type *>(lowerStr.c_str()));

    std::hash<HashedType> hasher;
    return hasher(lowerStr);
  }

  bool operator()(const HashedType & aStr1, const HashedType & aStr2) const
  {
    return CompareI(aStr1, aStr2) < 0;
  }

  //...

};
```

Exploring string hash tables
<code walk-through>

Case-insensitive hashes

```
template <class Type, class StringType = std::basic_string<Type>>
struct BasicStringHashI
{
  //...

  struct KeyEquality
  {
    bool operator()(const HashedType & aStr1, const HashedType & aStr2) const
    {
      return CompareI(aStr1, aStr2) == 0;
    }
  };

private:
  static void ToLower(char    * aStr) {  ::CharLowerA(aStr);  }
  static void ToLower(wchar_t * aStr) {  ::CharLowerW(aStr);  }

  static int CompareI(const string & aStr1, const string & aStr2)
  {  return ::lstrcmpiA(aStr1.c_str(), aStr2.c_str());  }

  static int CompareI(const wstring & aStr1, const wstring & aStr2)
  {  return ::lstrcmpiW(aStr1.c_str(), aStr2.c_str());  }

};
```

Exploring string hash tables
<code walk-through>

```
typedef BasicStringHashI<char>     StringHashI;
typedef BasicStringHashI<wchar_t>  StringHashWI;
```

Eg.

```
        std::unordered_map<wstring, TYPE, StringHashWI, StringHashWI::KeyEquality>
```

Case-insensitive hashes

Demo

Show me the code!

Research Topic for You

Optimal file-path (case-insensitive)

hash functor for std::unordered_map<>

open4tech@caphyon.com

FilePath Hasher

Special type of case-insensitive hash: **file-paths** hash map.

(a hasher for the string representation of file paths, in a ***case-insensitive*** file system)

What we want:

```
std::unordered_map<FilePath, TYPE, FilePathHash, FilePathHash::KeyEquality>
```

Where **FilePath** encapsulates a std::**wstring** plus file specific methods.

What issues do we have with regular **StringHashWI** for file paths ?

```
std::unordered_map<FilePath, TYPE, StringHashWI, StringHashWI::KeyEquality>
```

FilePath Hasher

Requirements:

- The operations which have to be fast are ***insertions*** and ***searches***
- Fast deletions would be desirable as well, but they are not a mandatory requirement
- A tradeoff between faster search time and slower insert time is accepted (if a faster search time can be achieved by slowing the insertion time a little bit)
- **`FilePathHash`** should be a drop-in replacement for **`StringHashWI`** (same API)
- **Benchmarks** for your FilePath hasher, having as baseline **`StringHashWI`**

So You Think You Can

Hashing Algorithms and Containers

inf.ucv.ro/~summer-school

Victor Ciura
Principal Engineer

@ciura_victor

June 2021