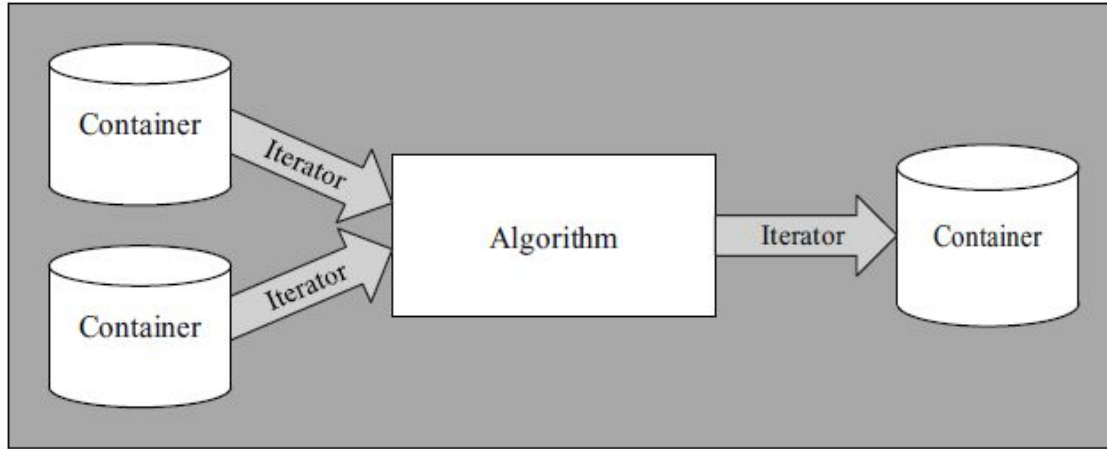


C++ Course 6: Containers + Miscellanea 2

2020 by Oleksiy Grechnyev

Standard Template Library (STL)



1. **Container** : A data class : **vector, array, map, set, ...**
2. **Iterator** : An object which iterates over a container (sort of a smart pointer)
3. **Algorithm**: A polymorphic algorithm : **sort, find, reverse, transform, copy, move ...**
4. **Function Objects + Lambda expressions**: Often used as arguments in algorithms

STL = Object Oriented + Generic + Functional programming

STL containers

Sequence Containers:

Array:



Vector:



Deque:



List:

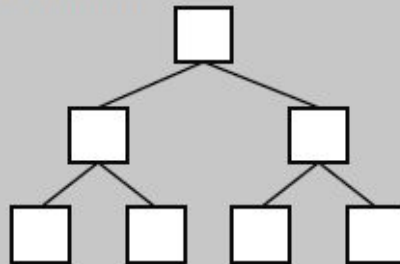


Forward-List:

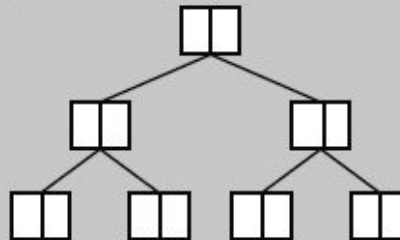


Associative Containers:

Set/Multiset:

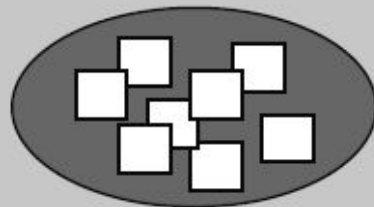


Map/Multimap:

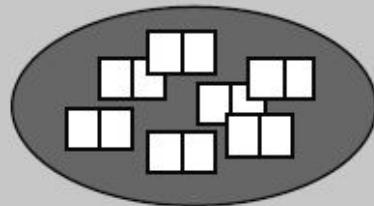


Unordered Containers:

Unordered Set/Multiset:



Unordered Map/Multimap:



Built-in arrays (C-arrays) : Don't use them!

```
int a[12]; // Array of size 12
double b[3] = {0.1, 1.2, 2.3};
string weapons[] = {"Sword", "Axe", "Bow"}; // Size can be omitted if initialized
constexpr int SIZE = 1024*1024*16; // SIZE must be constexpr
char buffer[SIZE];
char cString[] = "This is a null-terminated C-string"; // Automatically ended with \0
double matrix[10][10]; // Multidimensional
```

Arrays are indexed by the `[]` operator, starting from 0, no range checks (seg fault !!!)

```
for (int i=0; i<12; ++i)
```

```
    a[i] = i*i;
```

Arrays can be implicitly converted to pointers.

```
char * message = cString;
```

Size of an array (number of elements), does not work with pointers !

```
int size = sizeof a / sizeof (int);
```

Extra slides : C-arrays

What's wrong with built-in arrays? They are C legacy.

1. They cannot be copied.
2. They cannot be returned from a function.
3. They cannot be passed to a function by value. Converted to pointer !!!
`void fun(int a[]) {...}` // Converted to `int *` pointer !
`void fun(int a[37]) {...}` // Converted to `int *` pointer, size is ignored !
4. No `size()` method !
5. Must be of fixed size.
6. Array types in templates and containers are trouble.
7. Pointer is NOT an array. No range `for`, `begin()`, `end()` for pointers !

Don't use arrays, use *container classes* instead !

Exception: Array of constants:

```
const string names[] = {"Karen", "Lucia", "Anastasia", "Margaret", "Alice"};
```

std::array : C++ array of fixed (compile time) size

```
array<string, 5> aS1{"Karen", "Lucia", "Anastasia", "Margaret", "Alice"};
array<int, 100> al;
al.fill(17); // Fill with the value 17
constexpr int SIZE = 1024*1024*16;
array<double, SIZE> aD; // Size must be constexpr (compile time) !
auto aStr = std::experimental::make_array("Red", "Green", "Blue");
```

Possible implementation of **array**: thin wrapper around C-array:

```
template <typename T, size_t SIZE>
class array{
public:
    ... // Methods, operators
private:
    T myData[SIZE]; // The build-in array
}; // Note: std::array does not allocate any heap memory (stack overflow risk).
```

Creating std::array : more options

You can use type alias:

```
using SArray = array<string, 5>;  
SArray aS1{"Karen", "Lucia", "Anastasia", "Margaret", "Alice"}; // List creation  
SArray aS2 = {"Maria", "Nel", "Sophia", "Clair", "Mirage"}; // This is also OK  
SArray aS3;  
aS3 = aS1;           // Copy array (copies data, not reference !)  
aS1.swap(aS2);       // Swap arrays  
swap(aS1, aS2);      // Swap arrays (the same)
```

Get a raw pointer to the data (underlying built-in array):

```
string * rawData = aS1.data();
```

Create an std::array object out of a built-in array (NOT from pointer !):

```
string a[] = {"Maria", "Nel", "Sophia", "Clair", "Mirage"};  
auto aS4 = std::experimental::to_array(a);
```

std::array of funny types

Pointers (but NOT references) :

```
int i1 = 13, i2 = 17, i3 = 666;
```

```
array <int *, 3> aPtr{&i1, &i2, &i3};           // Pointers
```

```
array <const int *, 3> aCPtr{&i1, &i2, &i3};    // Pointers to const
```

Constants : Some other containers (e.g. **std::vector**) don't allow this:

```
array<const string, 5> cNames{"Maria", "Nel", "Sophia", "Clair", "Mirage"};
```

unique_ptr or **shared_ptr** objects (fine with other containers):

```
array<unique_ptr<int>, 2> uAr {  
    make_unique<int>(17),  
    make_unique<int>(666)  
};
```

Built-in arrays of fixed size:

```
array<int[17], 3> aa;
```


Indexing std::array

```
array <int, 12> a;
```

a.at(i) : Element **i** (Checks boundaries, throws **std::out_of_range**)

a[i] : Element **i** (Faster, no checks, seg fault !)

a.front() : First element

a.back() : Last element

a.size() : Number of elements

Set/modify array elements:

```
for (int i = 0; i < a.size(); ++i)
```

```
    a.at(i) = i*i;
```

Print the array:

```
for (int i = 0; i < a.size(); ++i)    // Using []
```

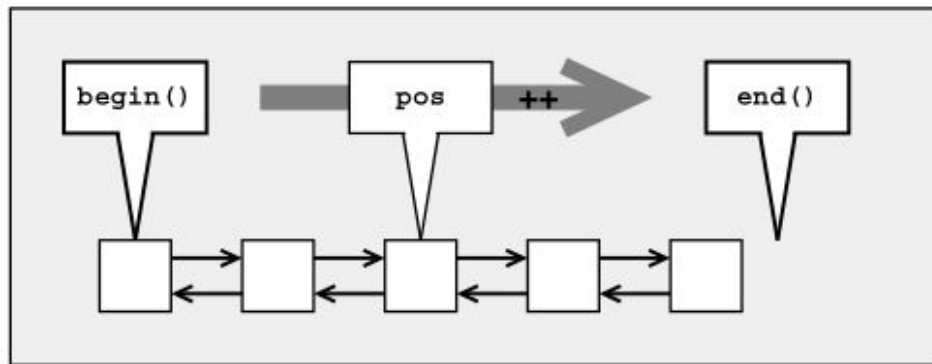
```
    cout << a[i] << " ";
```

```
for (int elem : a)    // Using range for
```

```
    cout << elem << " ";
```

Iterators

- *Iterators* are pointer-like objects that iterate over container elements.
- This includes containers without numerical index (**std::set**, **std::list**).
- Iterators have operators `*` and `->` defined (like pointers). ***iter** is the container element the iterator points at.
- All iterators support operators `++`, `==`, `!=`, `=`.
- Some iterators support operators `--`, `+`, `-`, `<`, `>`, `+=`, `-=`.
- Often iterators are pointers in disguise, but NOT for tree/hash sets/maps !



begin() and end()

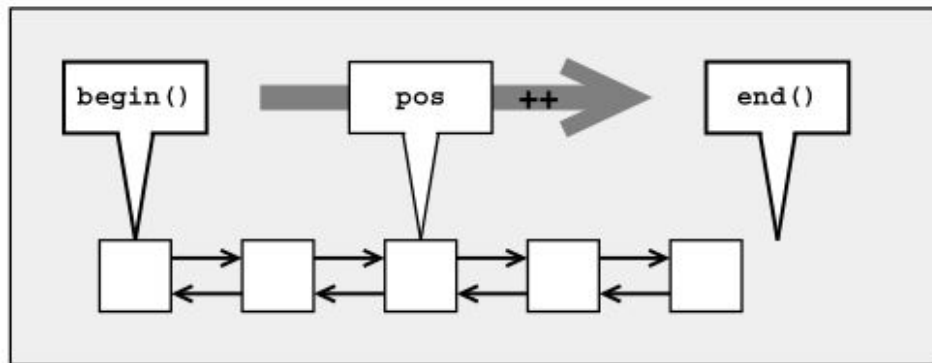
a.begin() or **begin(a)** is the iterator to the first element of a container **a**.

a.end() or **end(a)** is the iterator *past the last* element of a container **a**.

a.end() is not a valid element!

a.begin() == a.end() for an empty container.

Only the functional form **begin(a)**, **end(a)** can be used for C-arrays !



const and reverse iterators, for loop

Normal version : `a.begin()`, `a.end()`

const version : `a.cbegin()`, `a.cend()`

Reverse version : `a.rbegin()`, `a.rend()` all except `forward_list`

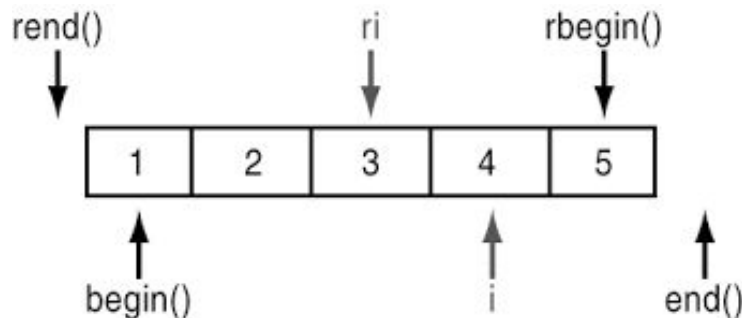
Reverse **const** version : `a.crbegin()`, `a.crend()` all except `forward_list`

Iterators in a **for** loop (same syntax for all containers !):

```
for (auto it = a.begin(); it != a.end(); ++it)
```

```
    *it *= *it; // Square each element of the container
```

Range **for** is based on iterators! Use range **for** whenever possible!



Iterator arithmetics (std::array, std::vector, ...)

Index to iterator:

```
auto it = a.begin() + index;
```

Iterator to index:

```
size_t index = it - a.begin();
```

More operations:

```
int diff = it2 - it1;    // Distance between two iterators
if (it1 < it2) ..        // Compare two iterators
it += 5;                 // Move forward by 5 elements
it -= 2;                 // Move back by 2 elements
it = a.end() - 1;        // Element before last
```

Iterator classes of `std::array` (usually `auto` is used):

```
array::iterator, array::const_iterator,
array::reverse_iterator, array::const_reverse_iterator
```

Templates to print any container (even built-in array !)

With range **for** (uses iterators under the hood !):

```
template <typename C>
void print(const C & c){
    for (const auto & e : c) // Duck typing ! Will not compile if C is not a container !
        cout << e << " ";
    cout << endl;
}
```

With iterators:

```
template <typename C>
void print2(const C & c){
    for (auto it = begin(c); it != end(c); ++it)
        cout << *it << " ";
    cout << endl;
}
```

Iterators, ranges, and algorithms

C++ algorithms use a range given by 2 iterators : **first**, **last**.

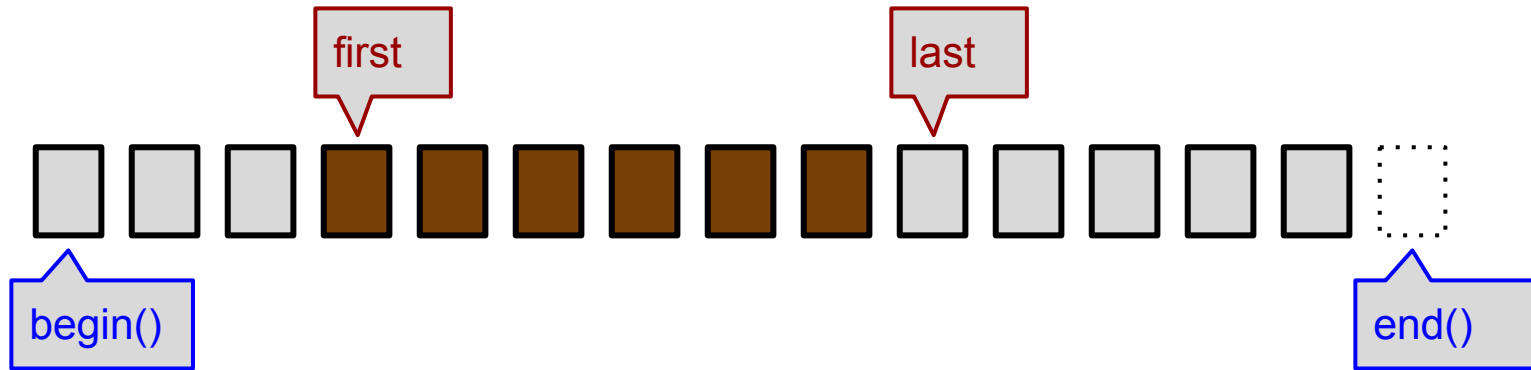
first : The first element of the range

last : The element past the last

Use **begin()**, **end()** to include the entire container in the range.

The algorithms are *polymorphic templates* (work with any container !)

This is basically a *slice* (a bit ugly syntax though)



Algorithms 1

Sort a container.

```
sort(first, last);  
sort(a.begin(), a.end());    // The entire container  
sort(begin(a), end(a));      // This form can be used for built-in arrays
```

Reverse a container.

```
reverse(first, last);
```

Random order.

```
shuffle(first, last, rng);    // rng = Random Number Generator
```

Find min and max elements (Returns iterator !).

```
auto minEl = min_element(first, last);  
auto maxEl = max_element(first, last);  
cout << "min = " << *minEl << ", max = " << *maxEl << endl;
```


Algorithms 2

Fill with a value : **fill**(first, last, value);

fill(a.begin(), a.end(), 13); // Fill container **a** with value 13

Copy elements. : **copy**(first, last, dest_first);

copy(a1.begin(), a1.end(), a2.begin()); // Copies **a1** to **a2**

Generate with a function. **generate**(first, last, fun);

int **n** = 0;

generate(a1.begin(), a1.end(), [&n]() {return n++;}); // 0, 1, .., 11

Apply a function to each element: **for_each**(first, last, fun);

for_each(a.begin(), a.end(), [](int &n){n*=3;}); // Multiply each element by 3

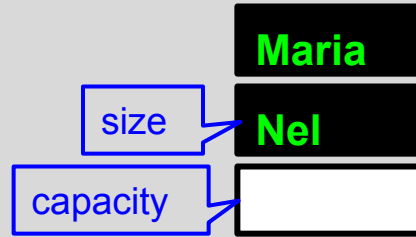
Extra slides: Algorithms

std::vector : A dynamic array. THE C++ container.

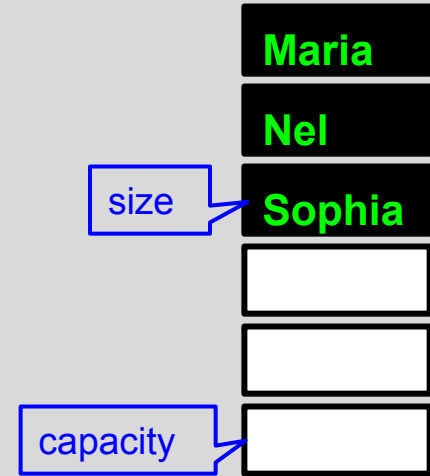
```
vector<string> vS;
```

```
string *data;  
int size;  
int capacity;
```

HEAP



Before growth



After growth

size Number of objects in container

capacity Number of reserved slots in the heap. It grows automatically if needed!

Creating std::vector

```
vector<int> v1;           // Empty vector
v1.push_back(17);        // Add elements to an empty vector
v1.push_back(19);
v1.push_back(26);
vector<int> v2(10);       // vector of 10 elements
vector<int> v3(10, 13);   // vector of 10 elements equal to 13
vector<int> v4{10, 13};   // vector of two elements : 10, 13
vector<int> v5 = {2, 3, 7, 11, 13, 17, 19, 23}; // List assignment constructor
vector<string> vS{"Maria", "Nel", "Sophia", "Clair", "Mirage"}; // List constructor
move() and swap() are very good for vectors !
```

Fast and slow operations:

Fast: access elements with `[]`, `move()` and `swap()`

Medium: access elements with `.at()`, adding to the end (`push_back`, `emplace_back`)

Slow: insert/delete in the middle

How to fill std::vector with data?

Indexing : Default Ctor, string Ctor, move assignment :

```
vector<Tjej> vT(5); // Default constructor 5 times !!!  
for (int i=0; i < 5; ++i)  
    vT.at(i) = Tjej("Tjej #" + to_string(i));
```

push_back : string Ctor, move Ctor :

```
vector<Tjej> vT; // Empty vector  
for (int i=0; i < 5; ++i)  
    vT.push_back(Tjej("Tjej #" + to_string(i)));
```

emplace_back : string Ctor. New objects are constructed in-place !

```
vector<Tjej> vT; // Empty vector  
for (int i=0; i < 5; ++i)  
    vT.emplace_back("Tjej #" + to_string(i));
```

But what the hell is going on (Tjej = logging class) ???

```
Ctor Tjej #0
Ctor Tjej #1
Move Ctor Tjej #0
Dtor
Ctor Tjej #2
Move Ctor Tjej #0
Move Ctor Tjej #1
Dtor
Dtor
Ctor Tjej #3
Ctor Tjej #4
Move Ctor Tjej #0
Move Ctor Tjej #1
Move Ctor Tjej #2
Move Ctor Tjej #3
Dtor
Dtor
Dtor
Dtor
```

std::vector capacity growth

```
vector<int> v;  
for (int i = 0; i <= 40; ++i) {  
    cout << "size = " << v.size() << ", capacity = " << v.capacity() << endl;  
    v.push_back(i);  
}
```

```
size = 0, capacity = 0  
size = 1, capacity = 1  
size = 2, capacity = 2  
size = 3, capacity = 4  
size = 4, capacity = 4  
size = 5, capacity = 8  
...  
size = 9, capacity = 16  
...  
size = 17, capacity = 32  
...  
size = 33, capacity = 64  
...
```

size vs capacity

SIZE operations:

<code>v.size();</code>	// Get size
<code>v.clear();</code>	// Delete all elements
<code>v.resize(17);</code>	// Change size (delete elements or create empty ones)

CAPACITY operations:

<code>v.capacity();</code>	// Get capacity
<code>v.reserve(1000);</code>	// Reserve storage (grow in size to given capacity)
<code>v.shrink_to_fit();</code>	// Trim capacity to size

Use `reserve()` before `push_back` / `emplace_back` !

Is growth a COPY or MOVE operation ?

If move constructor is **noexcept** : prefer MOVE !

Otherwise prefer COPY !

Don't forget **noexcept** in your move Ctor!

Extra slides: Insert and delete elements in the middle (slow !)

deque, set, unordered_set, multiset, unordered_multiset

std::deque : Like **vector**, but with fast **push_front()**, **emplace_front()**, **pop_front()**
std::set : Tree set. Sorted. Uses **operator<** or **less()** to compare.
std::unordered_set : Hash set. Unsorted. Uses function **hash()** .
std::multiset, **std::unordered_multiset** : Multisets can keep multiple copies.

```
set<int> s{1, 22, 2, 3, 19, 1, 3, 8, 12, 19, 22}; // Repeated, unsorted
```

Contains : 1 2 3 8 12 19 22

Look for element in the set:

```
int i = s.count(22); // Returns 0 or 1 (or number of copies)  
auto pos = s.find(22); // Returns iterator or s.end() if not found
```

Insert and delete:

```
s.insert(5);  
s.emplace(7);  
s.insert({11, 13, 17, 19, 23});  
s.erase(8);
```


map operations

Check if a key exists:

```
m1.count("Nel Zelphe"); // Returns 0 or 1
```

Access an element by key:

```
auto pos = m1.find("Nel Zelphe"); // Returns iterator, dereference if needed !
```

```
string s1 = m1["Maria Traydor"]; // Does not work on const map!
```

```
string s2 = m1.at("Nel Zelphe"); // Throws exception if key is not found
```

Delete an entry (by iterator):

```
m1.erase(pos);
```

Delete an entry (by key):

```
m1.erase("Mirage Koas");
```

Number of entries in a map:

```
m1.size();
```

Extra slide: Template to print a map

C strings and C++ strings (std::string)

C-strings: 0-terminated string : **char*** or **char []**

C-string literal: **"Hello"** : type **const char [7]**

C++ strings: **std::string** . Use this, don't use C-strings!

C++ string literals (C++ 14) : **"World"**s**** : Warning! Temporary objects, like **std::string(...)** !

H	e	l	l	o	\0
---	---	---	---	---	----

basic_string<T> : A vector-like container for primitive types only, **char_traits**

Note : C++ strings are *mutable* (unlike Java, Python, ...) !

using string = basic_string<char>; : UTF-8 string (works with IO streams)

using u16string = basic_string<char16_t>; : UTF-16 string

using u32string = basic_string<char32_t>; : UTF-32 string

using wstring = basic_string<wchar_t>; : Don't use this

Strings are real simple, see the EXTRA SLIDES

Do it yourself: C string operations, Regular Expressions

std::string 101

Create **std::string** from C-string literal :

```
string s1("Big [REDACTED]");
```

Concatenate with **+**, **+=** :

```
s1 += " Gun";
```

Create a 0-terminated C-string (**const char ***) from **std::string** :

```
const char * c1 = s1.c_str();
```

Find a substring, then remove it :

```
string s2 = "[REDACTED] ";
```

```
int pos = s1.find(s2);
```

```
if (pos != string::npos){
```

```
    // Remove the substring from s1 if found. std::string is mutable !
```

```
    s1.erase(pos, s2.size());
```

```
}
```

std::string_view (C++ 17)

std::string_view is a **std::string**-like interface to *existing* bytes in memory

std::string_view works like a slice (pointer + length) to *existing* string, has no text of its own !

Create from a C-string:

```
const char * c1 = "Take a look to the sky just before you die";
```

```
string_view sv1(c1); // Whole 0-terminated string
```

```
string_view sv2(c1 + 12, 10); // Substring
```

Create from a **std::string** :

```
string s3{"It is the last time you will"};
```

```
string_view sv3(s3); // Whole string
```

```
string_view sv4 = string_view(s3).substr(10, 9); // Substring
```

Warning ! The underlying **string** must be kept alive while you use **string_view**!

```
string_view sv(string("Error !!! Dangling pointer to a temporary !!!"));
```

Do it yourself: **std::valarray**

C++ and unicode : use UTF-8 ! And no locales !

1. Your code (*.h, *.cpp) must be in UTF-8 (string literals !).
2. Use **string** (not **wstring** !) for strings in UTF-8.
3. Use **cin**, **cout**, **ifstream**, **ofstream** with files in UTF-8.
4. Works fine with files, linux console.
5. Some trouble with windows console:
 Output: type **chcp 65001** in the console
 Input: I could not fix
6. Could be fixed with windows API if really needed.
7. GUI libraries have their own unicode support, e.g. **ustring** in gtkmm, **QString** in Qt.
8. Use C++ 11 **u16string** and **char16_t** if needed. UTF8 <-> UTF16 conversion!
9. Ignore the forum posts with locales, **wchar**, **wstring**, **wmain()**, **_tmain()**.

```
cout << "Український текст із літерами rГ !" << endl; // UTF-8 string literals
cout << "Svenska bokstäver ÅäÖöÄä !" << endl;
cout << "Hiragana : あ , い , う , え , お " << endl;
```

Using UTF-16 : char16_t and u16string

char16_t is a type for a UTF-16 character

```
char16_t c1 = u'İ', c2 = 0x456;
```

u16string is basic_string<char_16_t> :

```
u16string us2 = u"İİ€€ΓΓÅåÖöÄä"; // UTF-8 string literal converted to UTF-16
```

```
u16string us3{0x414, 0x456, 0x432, 0x43a, 0x430}; // Numerical UTF-16 values
```

```
u16string us4{u'İ', u'ж', u'a', u' ', u'å', u'ö', u'ä'}; // List of UTF-16 chars
```

UTF-8 <-> UTF-16 conversion: from_bytes(), to_bytes() (deprecated ? WTF ???):

```
wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> cvt; // Converter object
```

```
string s1 = "Український текст!"; // UTF-8 string
```

```
u16string us1 = cvt.from_bytes(s1); // Convert UTF-8 to UTF-16 !
```

```
cout << "us1 = " << cvt.to_bytes(us1) << endl; // Convert UTF-16 to UTF-8 !
```

```
for (char16_t c : us2) // Iterate over UTF-16 chars
```

```
    cout << cvt.to_bytes(c) << " " << hex << (int)c << dec << endl;
```

Time your code execution

Time your code with `system_clock` or `high_resolution_clock` :

```
auto t1 = std::chrono::system_clock::now(); // Time point t1
```

// Some code you want to time

```
auto t2 = std::chrono::system_clock::now(); // Time point t2
```

We want duration in milliseconds (Note: uses `std::chrono::duration` template):

Difference `t2 - t1` is `std::chrono::duration` in some unknown time units. Cast it to ms!

```
int dMs = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
```

Where `std::chrono::milliseconds` is `duration` (template instantiation) in ms `int` units.

The method `count()` returns numerical value of the `duration` object.

I prefer duration in `double` seconds:

```
using DSeconds = std::chrono::duration<double>; // Double duration of 1 second units
```

```
double dS = DSeconds(t2 - t1).count(); // No need for cast here, because double is 'exact'
```

Extra slides: C++ date and time, ratio, duration

Random numbers

Create random number generator from a seed:

```
mt19937 mt(seed);
```

Or use system time as a seed:

```
mt19937 mt(time(NULL));
```

Create a distribution:

```
uniform_int_distribution<int> uiD(-2, 4); // Integer -2 to 4 INCLUSIVE
```

Use this distribution with the random engine:

```
for (int i = 0; i < 20; ++i)  
    cout << uiD(mt) << endl;
```

Pseudorandom engines (templates with parameters) :

linear_congruential_engine, mersenne_twister_engine, subtract_with_carry_engine

Many pre-configured types: **mt19937**, **mt19937_64** (good), **minstd_rand** (fast)

Random distributions

Uniform integer distribution from $n1$ to $n2$ inclusive :

```
uniform_int_distribution<int> uiD(n1, n2);
```

Uniform real distribution from a to b :

```
uniform_real_distribution<double> urD(a, b);
```

Normal (Gaussian) distribution with mean and sigma :

```
normal_distribution<double> nD(mean, sigma);
```

Random boolean values with probability p :

```
bernoulli_distribution bD(p);
```

MANY other distributions !

Do it yourself: C random numbers: `rand()`, `srand()`

Library of the day: RapidJSON (Very Fast Header-only JSON parser)

Let us parse a JSON file hero.json :

```
{
    "name" : "Reimi Saionji",
    "age" : 19,
    "weapons" : ["Short Bow", "Eldarian Bow", "Torch Bow", "Hunting Bow",
"Earthsoul Bow"]
}
```

In CMakeLists.txt :

```
...
# Copy the file to build directory at the "cmake .." stage
file(COPY hero.json DESTINATION .)

# RapidJSON
find_package(RapidJSON REQUIRED)
message("RAPIDJSON_INCLUDE_DIRS = ${RAPIDJSON_INCLUDE_DIRS}")
include_directories(${RAPIDJSON_INCLUDE_DIRS})
...
```

Library of the day: RapidJSON (Very Fast Header-only JSON parser)

```
ifstream inFile("hero.json");           // Open json file
rapidjson::IStreamWrapper isw(inFile);  // Create rapidjson wrapper
rapidjson::Document d;                  // Create the document (DOM)
if (d.ParseStream(isw).HasParseError()) // Parse the stream
    throw runtime_error("Parse Error !");

if (d.HasMember("name") && d["name"].IsString()) // Read a field
    cout << "name = " << d["name"].GetString() << endl;
if (d.HasMember("age") && d["age"].IsInt())      // int or double ?
    cout << "age = " << d["age"].GetInt() << endl;
else if (d.HasMember("age") && d["age"].IsDouble())
    cout << "age = " << d["age"].GetDouble() << endl;

if (d.HasMember("weapons") && d["weapons"].IsArray()) { // Array
    cout << "weapons =\n";
    for (const rapidjson::Value & v : d["weapons"].GetArray() ) {
        if (v.IsString())
            cout << v.GetString() << endl;
    }
}
```

Thank you for your attention !

title

text

References and Pointers to array

Do not confuse with pointer to array *element*. Array type must be of fixed size !

```
int a[12];
```

```
int (& aRef1) [12] = a;
```

```
int (* aPtr1) [12] = &a;
```

Create a type alias:

```
using ArrayType = int [12];
```

```
ArrayType & aRef2 = a;
```

```
ArrayType * aPtr2 = &a;
```

Template to get array size:

```
template <typename T, size_t SIZE>
```

```
size_t getArraySize(const T (&) [SIZE]) { return SIZE; }
```

Arrays can be printed with a range for:

```
for (int i : a) cout << i << " ";
```

Dynamic arrays (pointers actually)

How do we create an array of dynamic size?

We cannot, use pointers instead:

```
int size = 1024;           // Not constexpr
int * data = new int[size];
for (int i=0; i<size; ++i)
    data[i] = i*i;          // We can use operator[] with pointers
delete [] data;             // Array delete
```

It is possible to use `unique_ptr` (but not `shared_ptr` !):

```
unique_ptr<int[]> data2(new int[size]);
```

Pointers are not real arrays!

You cannot use range `for`, `begin()`, `end()`, or our `getArraySize()` for pointers !

Absolutely no way to tell the size !

`sizeof(data)` returns pointer size (8 bytes) and not array size !

Pointers as array iterators

A C-style Array (NOT class) :

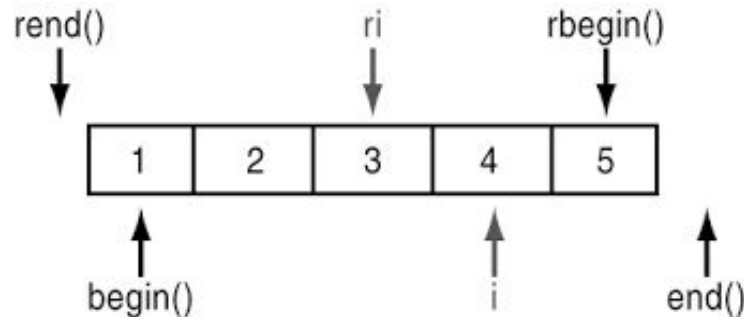
```
const string names[] = {"Karen", "Lucia", "Anastasia", "Margaret", "Alice"};
```

Print it with pointers :

```
const string * eit = names + 5;    // Position just after the last element
for (const string * it = names; it != eit; ++it)
    cout << *it << " ";
```

Or using C++ iterator style (only works with real array, not pointers !):

```
for (auto it = begin(names); it != end(names); ++it)
    cout << *it << " ";
```



Algorithms 3

Find first occurrence of an element. Returns last if not found.

find(first, last, value);

Find example:

```
array<int, 12> a{0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121};
```

```
it1 = find(a.begin(), a.end(), 16);
```

```
it2 = find(a.begin(), a.end(), 64);
```

```
if (it1 == a.end() || it2 == a.end()) // Check if found
    throw runtime_error("Not found !!!");
```

```
if (it1 > it2) // Check the order !
    swap(it1, it2);
```

```
reverse(it1, it2 + 1); // Reverse from 16 to 64 inclusive !
```

Dangers of using iterators

When using algorithms, **last** must be reachable from **first** by operator **++** !

sort(first, last);

For example:

++ ++ ++ ++ ++ first == last

Otherwise BAD error! Difficult to diagnose!

array<int, 12> a1, a2;

...

sort(a1.begin(), a1.end());

// OK

sort(a1.end(), a1.begin());

// Error ! Wrong order !

sort(a1.begin(), a2.end());

// Error ! Iterators of different arrays !

insert()/emplace() in the middle of vector

```
vector<int> v{1, 2, 3, 4, 5};

auto pos = find(v.cbegin(), v.cend(), 3); // Find position of 3
pos = v.insert(pos, 17);                  // Insert BEFORE 3
pos = v.insert(pos, {21, 22});            // Insert list before 17
pos = v.emplace(pos, 33);                 // Emplace BEFORE 21
// insert() returns iterator to the 1st new element !

// 1 2 33 21 22 17 3 4 5

for (auto it = v.cbegin(); it != v.cend(); ++it)
    if (*it > 20 && *it < 30)
        it = v.insert(it, 49) + 1;        // Insert 49 BEFORE 21, 22
// 1 2 33 49 21 49 22 17 3 4 5

// it -> 21
// insert 17 before 21, it -> 17
// +1 : it -> 21
// ++it : it -> 22
```

erase() : delete elements

```
vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

auto pos = find(v.cbegin(), v.cend(), 2);      // Find 2
pos = v.erase(pos);                            // Erase 2, pos -> 3
pos = v.erase(pos, pos+3);                    // Erase 3, 5, 6
// erase() returns iterator to the element AFTER erased

// 0 1 6 7 8 9

v.assign({0, 1, 2, 3, 4, 5, 6, 7, 8, 9});      // Just like Ctor

// Delete all even numbers in a for loop
for (auto it = v.cbegin(); it != v.cend(); ++it)
    if (*it % 2 == 0)
        it = v.erase(it) - 1;                // -1 to negate ++it

// 1 3 5 7 9
```

Templates to print a map

With range for :

```
template <typename M>
void printMap(const M & m){
    for (const auto & e : m)
        cout << e.first << " : " << e.second << endl;
}
```

With iterators:

```
template <typename M>
void printMap2(const M & m){
    for (auto it = m.begin(); it != m.end(); ++it)
        cout << it->first << " : " << it->second << endl;
}
```

Creating strings : constructors, assign

```
string s0;           // Empty string
```

From literals, char and C-strings :

```
string s1("Bastard Sword");           // "Bastard Sword"  
string s2 = "Heavy Crossbow";         // "Heavy Crossbow"  
string s3(18, 'Z');                   // "ZZZZZZZZZZZZZZZZZZZZ"  
string s4("Mary Had a Little Lamb", 8); // "Mary Had" (length)  
string s5("Mary Had a Little Lamb" + 5, 12); // "Had a Little"
```

From strings object:

```
string s6(s1, 8);                     // "Sword" (start pos)  
string s7(s2, 6, 5);                  // "Cross" (start pos, length)
```

assign() : change an existing strings object:

```
s3.assign(s2, 6, 5);                  // "Cross"
```

String operations

substr() : Substring (works just like constructors from **string**) :

```
string s1 = "Take a look to the sky just before you die";
```

```
string s2 = s1.substr(7);                // "look to the sky just before you die"
```

```
string s3 = s1.substr(7, 11);           // "look to the"
```

Length of a **string** :

```
s1.size() == s1.length() == 42
```

Convert to a C-string (0-terminated) :

```
const char * cS1 = s1.c_str();
```

The temporary C-string lives only as long as **s1** is alive and not modified !

Raw data (might be not 0-terminated in theory, in reality identical to **c_str()**) :

```
const char * raw = s1.data();
```


Container operations

Modify with a range for :

```
for (char & c : s)  
    c = toupper(c);
```

Print with iterators :

```
for (auto it = s.cbegin(); it != s.cend(); ++it)  
    cout << *it;  
cout << endl;
```

Sort using algorithm:

```
sort(s.begin(), s.end());
```

capacity, reserve, shrink_to_fit

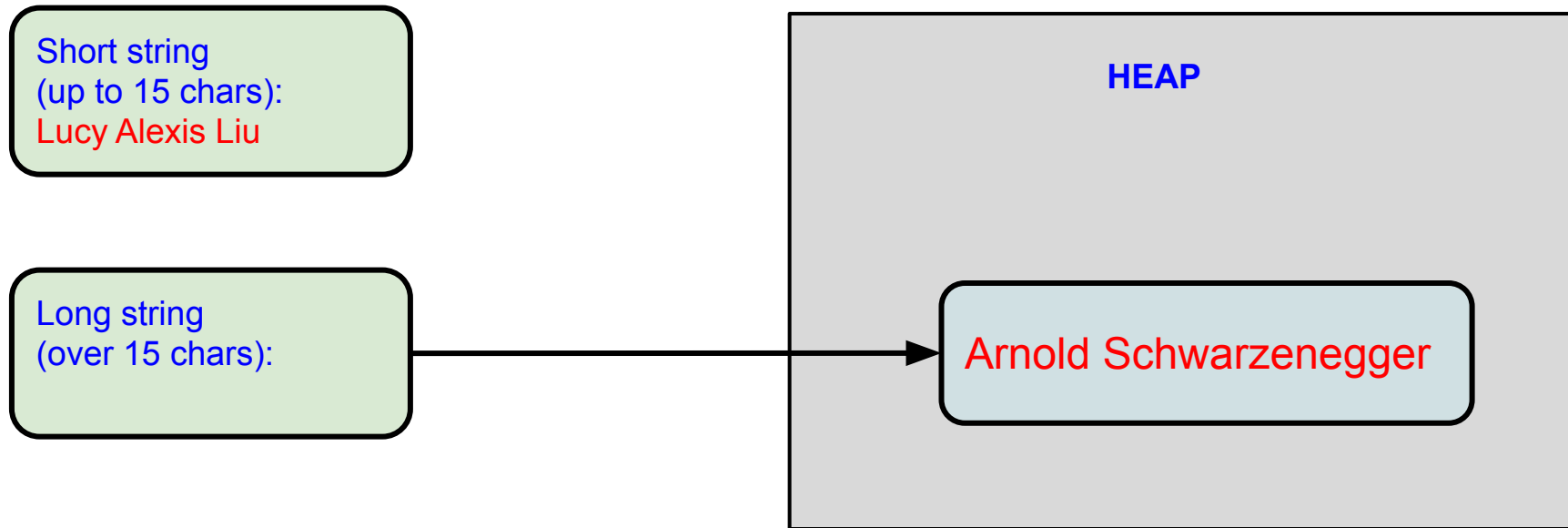
Capacity operations:

```
s.capacity();           // return capacity
s.reserve(100);         // reserve capacity
s.shrink_to_fit();      // Trim capacity to size
for (int i = 0; i < 65; ++i){
    cout << "size = " << s.size() << " , capacity = " << s.capacity() << endl;
    s.push_back('Z');
} // Growth : 15, 30, 60, 120 ...
```

Size operations:

```
s.size();               // return size, also s.length()
s.empty();              // return true if empty
s.clear();              // return size
s.resize(27);           // resize
s.resize(127, 'Z');     // resize filling with 'Z'
```

In-place and heap strings



Short strings are stored in the **string** object (initial/minimal **capacity()** = 15)

Long strings are stored in the HEAP

insert(), erase() a substring

Position-based `insert()` , syntax like constructor and `assign()` :

```
string s1 = "Lucy Liu";  
s1.insert(5, "Alexis ");           // "Lucy Alexis Liu"  
s1.insert(0, string("One Gorgeous Two"), 4, 9); // "Gorgeous Lucy Alexis Liu"  
s1.insert(s1.size(), 3, '!');      // "Gorgeous Lucy Alexis Liu!!!"
```

Iterator-based `insert()` , container syntax :

```
auto pos = s1.begin() + 9;  
pos = s1.insert(pos, '?') + 1;      // "Gorgeous ?Lucy Alexis Liu!!!"  
string s2(" Deadly ");  
s1.insert(pos, s2.cbegin(), s2.cend()); // "Gorgeous ? Deadly Lucy Alexis Liu!!!"
```

Position and iterator-based `erase()`:

```
s1.erase(0, 18);                    // "Lucy Alexis Liu!!!"  
pos = s1.begin() + 5;  
s1.erase(pos, pos + 7);             // "Lucy Liu!!!"  
s1.erase(8);                        // "Lucy Liu"
```

Concatenate: +, +=, append()

Concatenate with operator+ :

```
string s1 = string("One ") + "Two " + "Three";  
string s2 = "One " + string("Two ") + "Three";  
string s3 = "One " + ("Two " + string("Three"));  
string s4 = "One " + "Two " + string("Three");  
string s5 = "One " + "Two " + "Three";
```

Which lines are OK ? Which are errors?

Concatenate: +, +=, append(), replace()

Concatenate with **operator+** :

```
string s1 = string("One ") + "Two " + "Three";           // OK
string s2 = "One " + string("Two ") + "Three";           // OK
string s3 = "One " + ("Two " + string("Three"));         // OK
string s4 = "One " + "Two " + string("Three");           // Error !!!
string s5 = "One " + "Two " + "Three";                   // Error !!!
```

Concatenate with **append()** and **operator+=** :

```
string s = "Alpha ";           // "Alpha "
s.append("Beta ");             // "Alpha Beta "
s.append("Gamma Delta ", 6);    // "Alpha Beta Gamma "
s += "Epsilon ";               // "Alpha Beta Gamma Epsilon "
```

Modify with **replace()** : works like **erase()** + **insert()** :

```
s.replace(6, 4, "OMEGA"); // "Alpha OMEGA Gamma Epsilon "
```

find()

`find()` returns position of type `string::size_type` or `string::npos` if not found:

```
string s("Gorgeous ? Deadly Lucy Alexis Liu!!!");
```

Search for substring from left or right:

```
s.find("Alex")           // 23 : Gorgeous ? Deadly Lucy Alexis Liu!!!  
s.find("Alexander", 5);  // string::npos : starting position = 5  
s.find(" L")             // 17 : Gorgeous ? Deadly Lucy Alexis Liu!!!  
s.rfind(" L")            // 29 : Gorgeous ? Deadly Lucy Alexis Liu!!!
```

Search for any of the characters:

```
s.find_first_of(".,?!;") // 9,  Gorgeous ? Deadly Lucy Alexis Liu!!!  
s.find_last_of(".,?!;")  // 35, Gorgeous ? Deadly Lucy Alexis Liu!!!  
s.find_first_not_of(".,?!;") // 0,  Gorgeous ? Deadly Lucy Alexis Liu!!!  
s.find_last_not_of(".,?!;") // 32, Gorgeous ? Deadly Lucy Alexis Liu!!!
```

Search string with iterators and algorithms

returns iterators:

```
string s("Gorgeous ? Deadly Lucy Alexis Liu!!!");
```

Find a character:

```
find(s.cbegin(), s.cend(), 'L')           // 17 : Gorgeous ? Deadly Lucy Alexis Liu!!!
```

Find with a lambda expression:

```
find(s.cbegin(), s.cend(), [](char c)->bool{  
    return set<char>{'?', '!', ':', ',', '.', ';'}.count(c);  
})           // 9,  Gorgeous ? Deadly Lucy Alexis Liu!!!
```

Search for a substring:

```
const string s2("Alex");           // 23 : Gorgeous ? Deadly Lucy Alexis Liu!!!  
search(s.cbegin(), s.cend(), s2.cbegin(), s2.cend());
```

Search for a first occurrence of a character:

```
const string s3(".,?!;");           // 9,  Gorgeous ? Deadly Lucy Alexis Liu!!!  
find_first_of(s.cbegin(), s.cend(), s3.cbegin(), s3.cend());
```


Comparing strings:

Compare with `operator==` :

`string("Mary Ann") == string("Mary Ann")` // OK

`string("Mary Ann") == "Mary Ann"` // OK

`"Mary Ann" == string("Mary Ann")` // OK

`"Mary Ann" == "Mary Ann"` // Compares pointers, not strings !!!!

Compare with `compare()` : Returns number <0 , 0 , or >0 :

`string("abcd").compare("abce")` // <0

`string("abcd").compare("abc")` // >0

Compare two substrings (result == 0):

`string("Alpha Two Three Tango").compare(6, 9, string("One Two Three Four"), 4, 9)`

Number-string conversion

to_string(0.123456789) // "0.123456789"

String to int: **stoi**(std::string& str, size_t* pos = 0, int base = 10)

stoi("101") // 101

size_t st;

stoi("101", &st) // 101, st == 3 (Number of chars read)

stoi("101", nullptr, 2) // 5, binary

stoi("101", nullptr, 5) // 26, base 5

stoi("101", nullptr, 8) // 65, base 8

stoi("101", nullptr, 16) // 257, base 16

stoi("101", nullptr, 0) // 101, base 10 (auto base)

stoi("0101", nullptr, 0) // 65, base 8

stoi("0x101", nullptr, 0) // 257, base 16

Other types: **stof()**, **stod()**, **stold()**, **stol()**, **stoll()**, **stoul()**, **stoull()**

String streams: istream, ostream, stringstream

We can use strings as IO streams with operators <<, >>

Use **str()** (getter and setter) to access the underlying string

Useful for sophisticated string formatting

```
istream iss("13.98 17.32");
ostream oss;
double a, b;
iss >> a >> b;
oss << "a = " << a << " , b = " << b << " , a*b = " << a*b << endl;
cout << "oss.str() = " << oss.str(); // Contents of oss
```

If we want to reuse **iss** -- Если мы хотим снова использовать **iss** :

```
iss.str("3.0 7.0"); // Change the string in iss
iss.clear();        // To avoid failure on EOF !
```

We need **clear()** to clear the EOF bit !

Time and Date in C++

C++ time:

duration

clock

time_point

C time:

Time in seconds: **time_t, time()**

Execution time in milliseconds: **clock_t, clock()**

Calendar: **tm, localtime(), gmtime()**

Print: **ctime(), asctime(), strftime()**

Print (C++): **put_time**

Alternatives:

Boost or HowardHinnant/date

ratio : compile time rational number (fraction n/d)

ratio<n, d> : compile time rational number (fraction n/d)

using **R1** = **ratio**<1, 100>; // 1/1000

Template with static members only, do not create objects of this type

Numerator **R1::num** , Denominator **R1::den**

The fraction is reduced:

ratio<25, 15> // 5/3

ratio<100, -10> // -10/1

ratio_add<**ratio**<1, 2>, **ratio**<1, 3>> // 5/6

ratio_multiply<**ratio**<1, 2>, **ratio**<1, 3>> // 1/6

ratio_greater<**ratio**<1, 2>, **ratio**<1, 3>>::**value** // true

Predefined ratios:

atto, **femto**, **pico**, **nano**, **micro**, **milli**, **centi**, **deci**

deca, **hecto**, **kilo**, **mega**, **giga**, **tera**, **peta**, **exa**

std::chrono::duration

`duration<Rep, Period>` is a template for time intervals

Rep is a numerical type (`int`, `unsigned long long`, `double`)

Period is a time unit represented as **ratio** of seconds

```
using DMinutes = duration<double, ratio<60>>;    // 60/1
using DSeconds = duration<double>;                // 1/1
using DDays = duration<double, ratio<60*60*24>>;  // 60*60*24/1
using DHours = duration<double, ratio<60*60>>;    // 60*60/1
```

Examples from cppreference.com :

```
constexpr auto year = 31556952ll;    // seconds in average Gregorian year
using Shakes = duration<int, ratio<1, 100000000>>;
using Jiffies = duration<int, centi>;    // centi = 1/100
using Microfortnights = duration<float, ratio<14*24*60*60, 1000000>>;
using Nanocenturies = duration<float, ratio<100*year, 10000000000>>;
```

std::chrono::duration operations

Predefined durations :

nanoseconds, microseconds, milliseconds, seconds, minutes, hours

Declaring variables :

```
seconds s148(148);           //148 int seconds  
minutes m1(1);               //1 int minute  
DSeconds ds1_3(1.3);         //1.3 double seconds
```

Adding and subtracting durations (uses common denominator !):

```
auto dur1 = minutes(1) + seconds(3) - milliseconds(247);
```

Using literals (operator""h etc.):

```
using namespace std::chrono_literals;  
auto dur2 = 1h + 10min + 42s;  
auto dur3 = 1s + 234ms + 567us + 890ns;
```

count() , duration_cast

`count()` returns numerical value of a duration :

```
minutes m15(15);    // 15 minutes  
m15.count();        // Returns 15 (in minutes !)
```

`duration_cast<D>` casts to a duration type `D` :

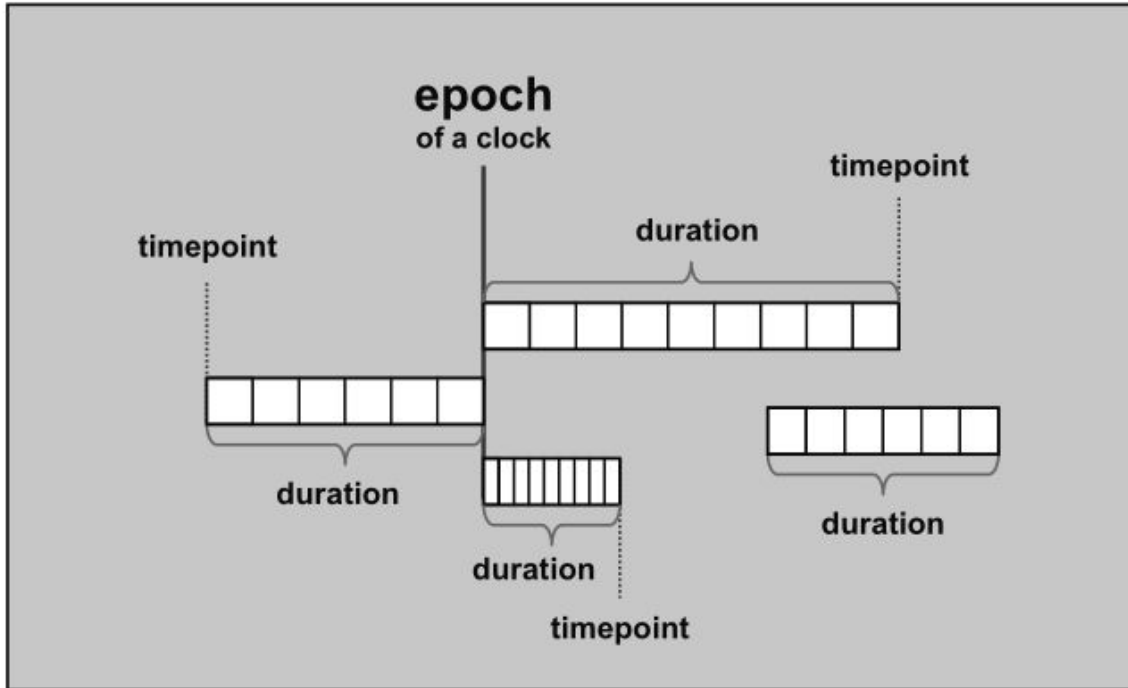
```
cout << "148 seconds = " << DMinutes(s148).count() << " DMinutes" << endl;  
cout << "148 seconds = " << duration_cast<minutes>(s148).count() << " minutes" << endl;  
cout << "1.3 seconds = " << duration_cast<milliseconds>(ds1_3).count() <<  
    " milliseconds" << endl;  
cout << "dur1 = " << milliseconds(dur1).count() << " milliseconds" << endl;  
cout << "dur2 = " << seconds(dur2).count() << " seconds" << endl;  
cout << "dur3 = " << DSeconds(dur3).count() << " DSeconds" << endl;
```

`duration_cast` is needed if there is a *precision loss*

Note: float/double are treated as exact, which they are not, no need for cast

Clocks

system_clock : Normal clock
steady_clock : Never adjusted
high_resolution_clock : Shortest time unit



Time execution of a method

```
auto t1 = high_resolution_clock::now();           // time_point 1
int result = fun(17);                             // Method to time
auto t2 = high_resolution_clock::now();           // time_point 2
```

```
nanoseconds dNS = duration_cast<nanoseconds>(t2-t1);
using DSeconds = duration<double>;
DSeconds dS = duration_cast<DSeconds>(t2-t1);
```

```
cout << "Timing(nanoseconds) : " << dNS.count() << endl;
cout << "Timing(seconds) : " << dS.count() << endl;
```

Sleep for a time interval:

```
this_thread::sleep_for(milliseconds(2600));
```

C time routines and calendars

time_t Integer type to store time in seconds since epoch (1970)

```
time_t t1 = time(nullptr);           // C function to get time
```

Get **time_t** from a C++ **time_point** :

```
system_clock::time_point tP2 = system_clock::now();    // auto can be used  
time_t t2 = system_clock::to_time_t(tP2);  // Convert to time_t
```

Different ways to print a **time_t** variable:

```
cout << "put_time(localtime()) : " << put_time(localtime(&t1), "%c %Z") << endl;  
cout << "put_time(gmtime()) : " << put_time(gmtime(&t1), "%c %Z") << endl;  // GMT !  
  
cout << "asctime(localtime()) : " << asctime(localtime(&t1));  
cout << "ctime : " << ctime(&t1);           // Short for asctime(localtime(&t1))  
cout << "asctime(gmtime()) : " << asctime(gmtime(&t1));  // GMT !
```

tm : a C structure for time+date

localtime(), gmtime() return ***tm** :

```
tm tM1 = *localtime(&t1);      // Copy from static buffer to tM1

cout << "put_time(&tM1) = " << put_time(&tM1, "%c %Z") << endl;

cout << "tM1.tm_year = " << tM1.tm_year << endl;
cout << "tM1.tm_mon = " << tM1.tm_mon << endl;
cout << "tM1.tm_mday = " << tM1.tm_mday << endl;
cout << "tM1.tm_hour = " << tM1.tm_hour << endl;
cout << "tM1.tm_min = " << tM1.tm_min << endl;
cout << "tM1.tm_sec = " << tM1.tm_sec << endl;
cout << "tM1.tm_wday = " << tM1.tm_wday << endl;
cout << "tM1.tm_yday = " << tM1.tm_yday << endl;
cout << "tM1.tm_isdst = " << tM1.tm_isdst << endl;
```

tm : a C structure for time+date

localtime(), **gmtime()** return ***tm** :

```
put_time(&tM1) = 10/03/17 16:59:13 FLE Daylight Time
tM1.tm_year = 117
tM1.tm_mon = 9
tM1.tm_mday = 3
tM1.tm_hour = 16
tM1.tm_min = 59
tM1.tm_sec = 13
tM1.tm_wday = 2
tM1.tm_yday = 275
tM1.tm_isdst = 1
```

Use external libraries such as Boost or HowardHinnant/date !