C++ Course 8: Strings. Time and date. Random numbers.

By Oleksiy Grechnyev

C strings and C++ strings

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

String literal: Строковые литералы: "**Hello**" : type **const char [7]**

H	**e**	**l**	**l**	**l**	**o**	**\0**

C++ strings: String classes. Строковые классы.

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

using string = basic_string<char>;	: UTF-8 string (works with IO streams)
u16string = basic_string<char16_t>;	: UTF-16 string
u32string = basic_string<char32_t>;	: UTF-32 string
wstring = basic_string<wchar_t>;	: Don't use this

Use C++ strings, not C strings!

Используйте C++ строки, не C строки!

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');                                   // "ZZZZZZZZZZZZZZZZZZ"
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) : Преобразовать в C-строку:

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

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

Временная C-строка живет пока **s1** жива и не модифицирована !

Raw data (might be not 0-terminated !) :

```
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

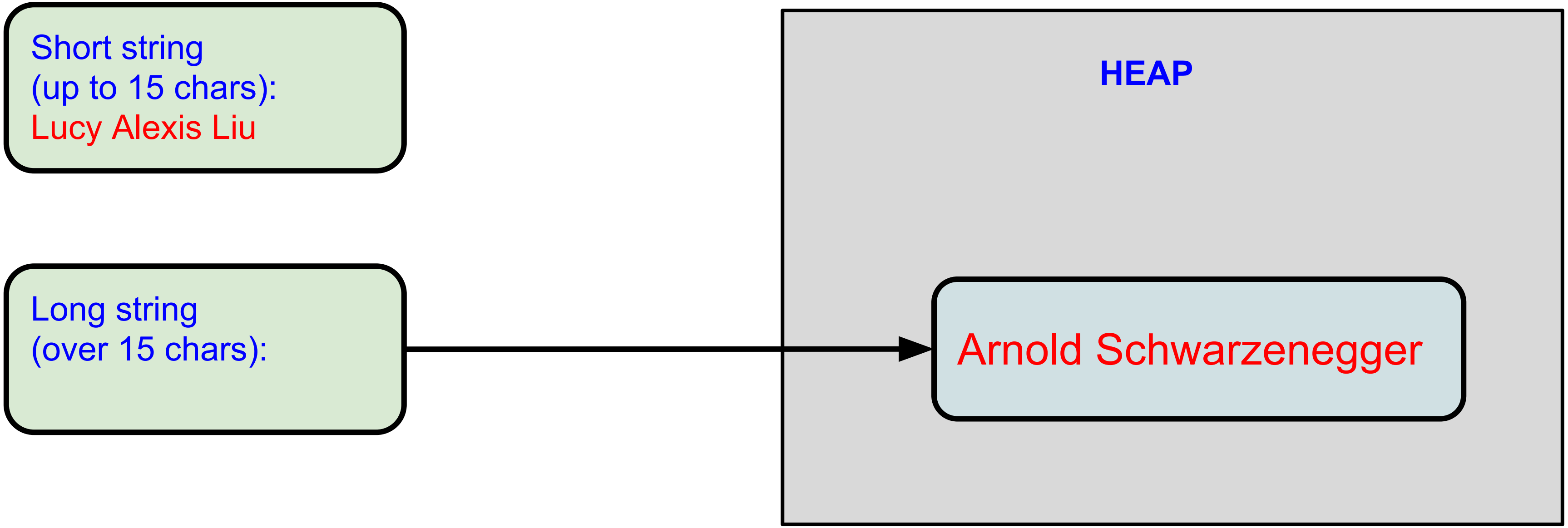
Capacity operations:

```
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 string
(up to 15 chars):
Lucy Alexis Liu

Long string
(over 15 chars):

HEAP

Arnold Schwarzenegger

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

Короткие строки хранятся в объекте **string** (начальный/минимальный **capacity()** = 15)

Long strings are stored in the HEAP

Длинные строки хранятся в хипе

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: **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: istringstream, ostringstream, stringstream

To use strings as streams -- Использовать строки как потоки

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

```
istringstream iss("13.98  17.32");
ostringstream 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 !

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.
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**.
8. Use C++ 11 **u16string** and **char16_t** if needed. UTF8 <-> UTF16 conversion!

```
cout << "Український текст із літерами ґҐ !"  << endl;
cout << "Svenska bokstäver ÅåÖöÄä !" << 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'Ї', c2 = 0x456;
```

u16string is **basic_string<char_16_t>** :

```
u16string us2 = u"ЇїЄєҐґÅåÖöÄä";                  // UTF-8 string literal converted to UTF-16
u16string us3{0x414, 0x456, 0x432, 0x43a, 0x430};  // Numerical UTF-16 values
u16string us4{u'Ї', 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()** :

```
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 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> : рациональное число времени компиляции (дробь num/den)

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

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

Numerator (числитель) **R1::num** , Denominator (знаменатель) **R1::den**

The fraction is reduced: дробь упрощается:

```
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

duration<Rep, Period> -- это template (шаблон) для промежутков времени

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

Period is a time unit represented as **ratio** of seconds (единица времени в секундах)

```
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, 1000000000>>;
```

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** :

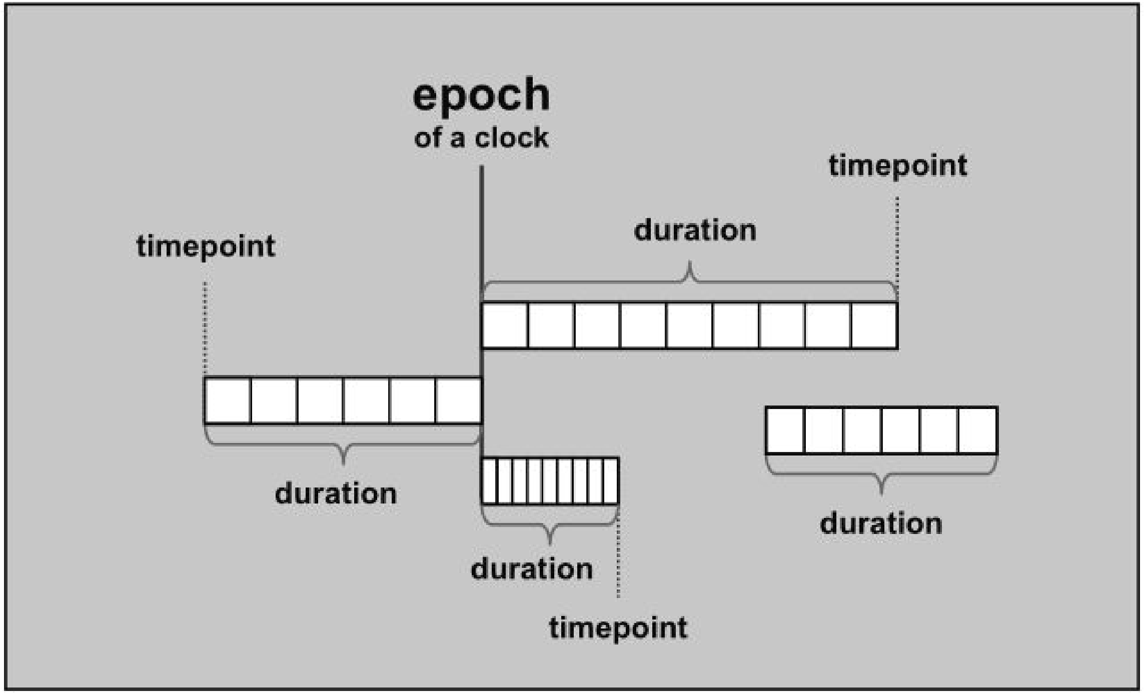
```
cout << "148 seconds = " << DMinutes(s148).count() << " DMinutes" << endl;
cout << "148 seconds = " << duration_cast<minutes>(s148).count() << " minutes" << endl;
cout << "1.3 seconds = " << duration_cast<milliseconds>(ds1_3).count() <<
                   " milliseconds" << endl;
cout << "dur1 =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

duration_cast необходим если есть *потеря точности* !

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;
```

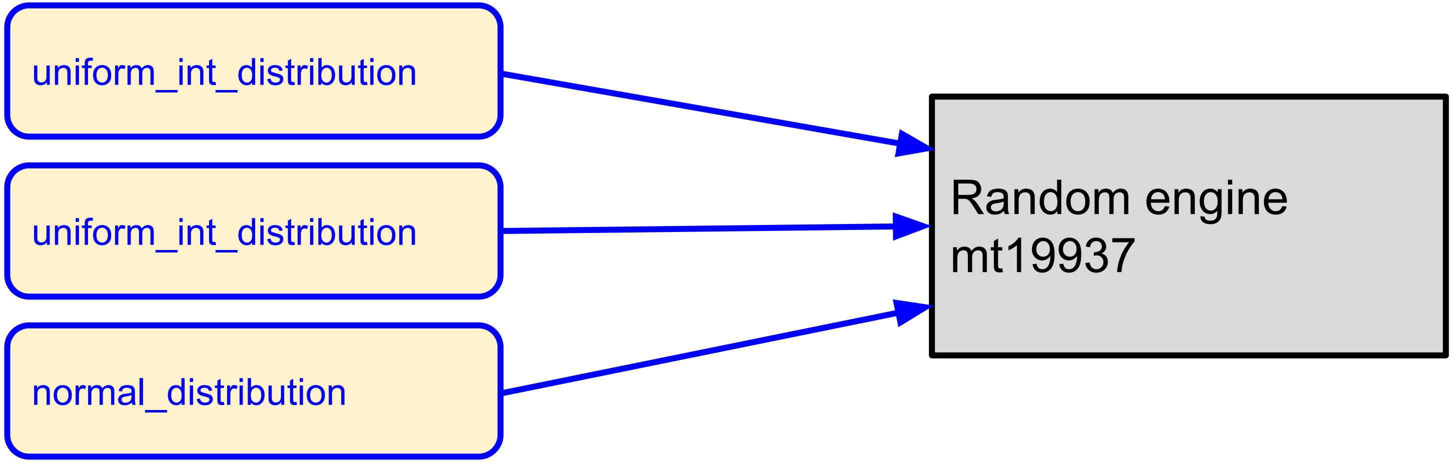
tm : a C structure for time+date

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 !

Random numbers



Random engine: pseudorandom number generator

Random engine: движок который генерирует псевдослучайные числа

Distribution: Wrapper which gives type, range and distribution law (e.g 1 to 10)

Distribution: Обертка, которая дает тип, диапазон и закон распределения

Random engines

Pseudorandom algorithms (templates with parameters) :

linear_congruential_engine, mersenne_twister_engine, subtract_with_carry_engine

Many predefined types: **mt19937, mt19937_64** (good), **minstd_rand** (fast)

Seeding with **random_device** (NOT random in MinGW !!!) :

```
mt19937 mt(random_device{}());
```

Seeding with time :

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

mt is the random engine variable. Use it for your distributions:

```
uniform_int_distribution<int> uiD(-2, 4);    // Integer -2 to 4 inclusive
for (int i = 0; i < 20; ++i)
        cout << uiD(mt) << endl;
```

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 !

Thank you for your attention !

title

text