Time handling in C (1)

In classical C the reading of current time from the system clock is performed as follows:

```
#include "time.h"
time_t now; // time_t is specified by typedef, in Visual Studio it is is a 64-bit integer
time(&now); // the number of seconds since January 1, 1970, 0:00 UTC
```

To get the current date and time understandable for humans use the standard *struct tm*:

```
struct tm // do not declare it in your code, it is already declared by time.h
{
   int tm_sec;   // seconds after the minute - [0, 60] including leap second
   int tm_min;   // minutes after the hour - [0, 59]
   int tm_hour;  // hours since midnight - [0, 23]
   int tm_mday;  // day of the month - [1, 31]
   int tm_mon;   // months since January - [0, 11], attention: January is with index 0
   int tm_year;  // years since 1900, attention, not from the birth of Christ
   int tm_wday;  // days since Sunday - [0, 6], attention: Sunday is with index 0, Monday 1
   int tm_yday;  // days since January 1 - [0, 365]
   int tm_isdst; // daylight savings time flag
};
```

To fill this struct:

```
struct tm now_tm;
localtime_s(&now_tm, &now);
```

Time handling in C (2)

Example:

```
printf("Today is %d.%d.%d\n",
  now_tm.tm_mday, now_tm.tm_mon + 1, now_tm.tm_year + 1900);
```

Function *asctime_s* converts the *struct tm* to string:

```
char buf[100];
asctime_s(buf, 100, &now_tm);
printf("%s\n", buf); // prints like Thu Jan 23 14:26:42 2020
```

but here we cannot set the format. Better is to use function *strftime*, for example:

```
strftime(buf, 100, "%H:%M:%S %d-%m-%Y", &now_tm);
printf("%s\n", buf);   // prints according to Estonian format 14:26:42 23-01-2020
```

The complete reference of *strftime* is on http://www.cplusplus.com/reference/ctime/strftime/

The attributes of *struct tm* may be modified. For example, if we want to know what date is after 100 days, do as follows:

```
struct tm future_tm = now_tm;
future_tm.tm_mday += 100; // add 100 days
time_t future = mktime(& future_tm); // convert back to time_t
localtime_s(&future_tm, &future); // convert once more to struct tm
asctime_s(buf, 100, &future_tm);
printf("%s\n", buf); // prints like Sat May 2 15:26:42 2020
```

Time handling in C++

In C++ we have more powerful but complicated tools:

```
#include <chrono> // see https://en.cppreference.com/w/cpp/chrono.html
using namespace std::chrono; // do not forget!
```

Namespace *chrono* includes several concepts:

- *duration*
- *timepoint*
- *clock*
- *date*
- *timezone*

The last two of them are extensions introduced in C++ v. 20.

Rational numbers (1)

In mathematics, a rational number can be expressed as fraction *a / b*, where *a* is called as numerator and *b* as denominator. The decimal expansion of a rational number may have finite number of digits like *1.234*. But it may also have endless number of digits in which a sequence of digits is repeating over and over, like *7 / 3 = 2.33333…..*

An irrational number like *sqrt(2), π, e* has also endless decimal expansion, but without repeating.

Problems with endless rational numbers:

```
double x = 2.3333333; // actually in specification this expression is written as 7 / 3
double y = x * 3; // we get 6.9999999, but the correct value is 7
```

To get results of calculations that are as exact as possible, we need to use template *ratio*:

```
typedef <numerator_as_integer_constant, denominator_as_integer_constant> ratio_name;
```

The denominator has default value 1. Examples:

```
#include <ratio> // see http://www.cplusplus.com/reference/ratio/ratio/
const int numerator = 7, denominator = 3; // must be constant expression
typedef ratio<numerator, denominator> test1;
typedef ratio<7, 3> test2;
```

To access numerator and denominator, use public members *num* and *den*, for example:

```
cout << test1::num << ' ' << test1::den<< endl;
```

Rational numbers (2)

The following expression is for adding ratios:

```
typedef ratio_add<addend_1 _as_ratio, addend_2_ as_ratio> ratio_name;
```

Example:

```
typedef ratio<7, 3> test1;
typedef ratio<5, 6> test2;
typedef ratio_add<test1, test2> sum;
cout << sum::num << ' ' << sum::den << endl; // prints 19 6
```

ratio_subtract, *ratio_multiply* and *ratio_divide* are similar.

An *integral_constant* is a standard class (better to say *struct*) template that stores the type and constant value. For example, *integral_constant<bool, true>* stores a boolean value *true* and *integral_constant<int, 100>* stores integer *100*. It has two members: *type* and *value*.

To compare two ratios write expression:

```
typedef ratio_equal<ratio_1, ratio_2> integral_constant_name;
```

The results is *integral_constant<bool, true>* or *integral_constant<bool, false>*

Example:

```
typedef ratio<7, 3> test1;
typedef ratio<5, 6> test2;
typedef ratio_equal<test1, test2> res;
cout << (res::value ? "Equal" : "Not equal") << endl;
```

Rational numbers (3)

ratio_not_equal, ratio_less, ratio_less_equal, ratio_greater, ratio_greater_equal are similar.

All the ratio templates are evaluated at compile time. The values for numerator and denominator cannot be calculated at run time, for example:

```
int x;
cin >> x;
typedef ratio <x, 2> test; // error
```

There are no C++ operations between rational numbers and integers or doubles. So, if we have

```
typedef ratio<5, 6> test2;
```

and we want to multiply it with 2, we need to write

```
typedef ratio<2, 1> test3; // or simply ratio<2>
typedef ratio_multiply<test2, test3> test4;
cout << test4::num << ' ' << test4::den << endl; // prints 5 3
```

C++ has several predefined ratios, for example *nano* (i.e. 1 / 1e9), *micro* (i.e. 1 / 1e6), *milli* (i.e. 1 / 1e3), *kilo* (i.e. 1e3 / 1), *mega* (i.e. 1e6 / 1), etc.

Duration (1)

Template *duration* (see https://en.cppreference.com/w/cpp/chrono/duration.html) represents an interval between two timepoints:

```
template<typename T1, typename T2> class duration { ………….. };
```

Here *T1* is used for variable storing the number of ticks (*int, long int, double*, etc.) and *T2* is for ratio presenting the period of one tick in seconds. The default value for *T2* is *ratio<1, 1>* (or simply *ratio<1>*). Examples:

```
duration<long int> d1; // ratio has default value, it means that tick is one second
duration<long int, ratio<60, 1> > d2; // tick is one minute
duration <long long int, ratio<1, 10>>  d3; // tick is one tenth of second
```

Predefined ratios like *nano* or *micro* often help to define durations:

```
duration <long int, micro>  d4; // tick is one microsecond
```

Constructor without parameters does not initialize the number of ticks, consequently variable *d1*, *d2*, *d3* and *d4* are not ready to use.

```
duration <long int, milli>  d5(1000); // now the duration is 1000 ms
```

Method *count* returns the value of ticks, for example:

```
cout << d5.count() << endl;
```

There are no methods to set a new value ticks.

Duration (2)

There are several typedefs for typical durations. Examples:

```
hours d1(24);               // declares time interval 24 hours
minutes d2(10);             // declares time interval 10 minutes
seconds d3(20);             // declares time interval 20s
milliseconds d4(1500);      // declares time interval 1500ms
microseconds d5(1500);      // declares time interval 1500µs
nanoseconds d6(1500);       // declares time interval 1500ns
days d7(31);                // declares time interval 31 days (from C++ v.20)
weeks d8(1);                // declares time interval 1 week (from C++ v.20)
months d9(3);               // declares time interval 3 months (from C++ v.20)
years d10(100);             // declares time interval 100 years (from C++ v.20)
```

Duration has a large set of operator functions for arithmetics and comparison. The operands may be of different types. Examples:

```
milliseconds d1(1000);
milliseconds d2(2000);
milliseconds d3 = d1 + d2; // get time interval 3000ms
cout << d3 << endl; // prints 3000ms
cout << boolalpha << (d1 > d2) << endl;  // prints false
milliseconds d4 = d3 + 1000; // error, operations between durations and integers
                             // are not defined
```

Duration (3)

```
milliseconds d1(1000);
milliseconds d2(2000);
microseconds d3 = d1 + d2; // different types !
cout << d3 << endl; // prints 3000000us
```

If the implicit cast is not possible, the *duration_cast* operator solves the problem:

```
milliseconds d4 = d3; // error
milliseconds d4 = <duration_cast>(d3); // correct
cout << d4 << endl; // prints 3000ms
```

But

```
milliseconds d5(100);
seconds d6 = duration_cast<seconds>(d5); // formally correct
cout << d6 << endl; // prints 0s because in typical durations the number of ticks is integral
duration<double> d7 = d5;
cout << d7 << endl; // prints 0.1s: correct
```

Read more on https://en.cppreference.com/w/cpp/chrono/duration.html

Duration (4)

Suppose we have a long duration measured in seconds. To analyse it, in most cases we need to recalculate it into format hh:mm:ss. For that C++ v.20 has a helper class *hh_mm_ss*.

Examples:

```
seconds dsec = measure_process_duration();  // some function is called, result is 7456 s
hh_mm_ss hms1(dsec);  // create object from class hh_mm_ss
cout << "Process duration was " << hms1.hours() << ":" << hms1.minutes() << ":" <<
hms1.seconds() << endl;   // prints 2h:4min:16s
```

```
milliseconds dmsec = measure_sorting_duration();  // result is 745454924 ms
hh_mm_ss hms2(dmsec);
cout << "Sorting needed " << hms2.seconds() << " " << hms2.subseconds() << endl;
  // prints 14s 924ms
```

To compare two objects from class *hh_mm_ss* turn them to duration:

```
if (hms1.to_duration() < hms2.to_duration()) { …. }
```

To covert object from class *hh_mm_ss* to string with C++ v.20 formatting

```
cout << format("{:%R}", hms1) << endl;  // prints 02:04
cout << format("{:%T}", hms1) << endl;  // prints 02:04:16
```

Read more on https://en.cppreference.com/w/cpp/chrono/hh_mm_ss.html

https://en.cppreference.com/w/cpp/chrono/hh_mm_ss/formatter.html

Clock and time_point (1)

C++ v.11 defines 3 clocks:

- *system_clock* represents timepoints associated with the computer usual real-time clock. See https://en.cppreference.com/w/cpp/chrono/system_clock.html
- *steady_clock* guarantees that it never gets adjusted.
- *high_resolution_clock* represents the clock with the shortest possible tick period. In Visual Studio equivalent with the *system_clock*.

C++ v.20 expands the list of clocks:

- *utc_clock* for Coordinated Universal Time
- *tai_clock* for International Atomic Time
- *gps_clock* for GPS time
- ………..

In this course we deal only with the *system_clock.* To read the current time from the system clock use method now():

```
system_clock::time_point now = system_clock::now();
```

Turn attention, that a *time_point* is always associated with a clock:

```
time_point<system_clock> t; // correct
system_clock::time_point t; // correct
time_point t; // error, clock not specified
```

Clock and time_point (2)

Actually, *time_point* (see https://en.cppreference.com/w/cpp/chrono/time_point.html) is a template:

```
template<typename T1, typename T2> class time_point { ………….. };
```

Here *T1* is used for clocks (*system_clock,* etc.) and *T2* for duration, i.e. the interval between the current moment and the epoch (or origin, 01.01.1601 in case of Windows, 01.01.1970 in case of Linux). Its value is actually the duration from the epoch (measured in 100ns units in case of Windows and seconds in case of Linux).

For example, if we write:

```
time_point<system_clock, duration<long long int, ratio<1, 1> > > t;
```

then *t* measures the number of seconds from epoch, the value is retrieved from system clock. Theoretically we may declare timepoints in many different ways but actually the duration parameters (epoch and tick period) are built into clock. Consequently, each clock must have its own standard for timepoint:

To know which ratio is used in the duration of system clock, write the following snippet:

```
cout << system_clock::period().num << " " <<  system_clock::period().den << endl;
```

On the instructor's computer the result was *1 10000000*.

To know what is the type for ticks in the duration of your system clock, write the following code snippet:

```
cout << typeid(system_clock::rep).name() << endl;
```

On the instructor's computer the result was *__int64*.

Clock and time_point (3)

For many people it is more convenient to continue with C time handling tools:

```
system_clock::time_point now = system_clock::now();
time_t now_t = now; // convert to time_t
struct tm now_tm;
localtime_s(&now_tm, &now_t);
struct tm future_tm = now_tm;
future_tm.tm_mday += 100; // add 100 days
time_t future_t = mktime(& future_tm);
```

There is a standard function std::put_time to create from *struct tm* time strings for *iostream* and *sstream*:

```
#include <iomanip>
cout << put_time(&future_tm, "%d-%m-%Y %H:%M:%S") << endl;
```

or

```
stringstream sout;
sout << put_time(&future_tm, "%d-%m-%Y %H:%M:%S") << endl;
cout << sout.str() << endl;
```

See more from http://www.cplusplus.com/reference/iomanip/put_time/

To turn back to C++ tools:

```
system_clock::time_point future = system_clock::from_time_t(future_t);
```

Clock and time_point (4)

Timepoint has a set of operator functions for arithmetics and comparison. The only operation between two timepoints is subtraction, its result is a duration:

```
system_clock::time_point start = system_clock::now();
int i;
cin >> i; // to introduce a pause
system_clock::time_point end = system_clock::now();
auto diff = end - start;
cout << typeid(diff).name() << endl;
```

the result is *class std::chrono::duration<__int64,struct std::ratio<1,10000000> > ,* i.e. the type of duration presenting the difference between two timepoints is the same as the duration in system_clock::time_point.

We may convert implicitly the difference into nanoseconds but not to milliseconds or seconds:

```
nanoseconds dn = diff;
duration<double> ds = diff;
cout << dn << endl; // prints 4487655200ns
cout << ds << endl; // prints 4.48766s
```

Clock and time_point (5)

It is possible to add to timepoint a duration as well as subtract a duration from it:

```
system_clock::time_point now = system_clock::now();
cout << now << endl; // prints 2025-08-22 11:43:10.4304195
system_clock::time_point after_an_hour = now + hours(1);
cout << after_an_hour << endl; // prints 2025-08-22 12:43:10.4304195
system_clock::time_point before_an_hour = now - hours(1);
cout << after_an_hour << endl; // prints 2025-08-22 10:43:10.4304195
```

Template function *floor<>()* truncates the timepoint to a precision of days or seconds:

```
system_clock::time_point in_days = floor<days>(now);
cout << in_days << endl; // prints 2025-08-22 00:00:00.000000
system_clock::time_point in_seconds = floor<seconds>(now);
cout << in_seconds << endl; // prints 2025-08-22 10:43:10.000000
```

Remember that timepoint is template<typename T1, typename T2> class time_point { ... }; where *T1* is the clock and *T2* is the duration from the epoch.

```
time_point<system_clock, days> t1 = time_point_cast<days>(now);
    // t1 presents the number of days from epoch
cout << t1 << endl; // prints 2025-08-22
time_point<system_clock, seconds> t2 = time_point_cast<seconds>(now);
   // t2 presents the number of seconds from epoch
cout << t2 << endl; // prints 2025-08-22 10:43:10
```

Clock and time_point (6)

To simply the work of code writers, several aliases are introduced:

- *sys_time* is the alias of *time_point<system_clock, duration>*
- *sys_days* is the alias of *time_point<system_clock, days>*
- *sys_seconds* is the alias of *time_point<system_clock, seconds>*

So, we may write simply:

```
sys_time now = system_clock::now();
sys_days now_days = timepoint_cast<days>(now);
sys_seconds now_seconds = timepoint_cast<seconds>(now);
```

Calendar (1)

C++ v.20 provides several classes for operating with calendar. The full list is on page https://en.cppreference.com/w/cpp/chrono.html. The most important of them are:

```
year y { 2026 };   // allowed values -32767 … +32767
cout << static_cast<int>(y) << endl;  // get the value wrapped into object
cout << "This is" << (y.is_leap() ? " " : " not") << " a leap year" << endl;  // leap year?

month m1  { 2 }; // allowed values 1 … 12
cout << static_cast<unsigned int>(m1) << endl;  // get the value wrapped into object
month m2  { February };  // constants January, February etc. are defined in namespace chrono
cout << static_cast<unsigned int>(m1) << endl; // prints 2

day d1  { 15 }; // allowed values 1 … 31
cout << static_cast<unsigned int>(d1) << endl;  // get the value wrapped into object

weekday wd1  { 1 }; // allowed values 0 … 6, Sunday is 0
cout << wd1.c_encoding() << endl;  // get the value wrapped into object, prints 1
cout << wd1.iso_encoding() << endl; // prints 1
cout << wd1 << endl; // prints Mon
weekday wd2 { Sunday };  // constants Sunday, Monday etc. are defined in chrono
cout << wd2.c_encoding() << endl;  //  prints 0
cout << wd2.iso_encoding() << endl; // prints 7
                          // see https://en.cppreference.com/w/cpp/chrono/weekday/encoding.html
cout << wd2 << endl; // prints Sun
```

Calendar (2)

All the presented classes have method *ok()* to check the correctness of value. For example:

```
month m { 13 };  // error not detected
cout << boolalpha << m.ok() << endl; // prints false
```

There are also operator functions for comparison and:

- to add to and subtract from *year* duration in *years*
- to add to and subtract from *month* duration in *years* or *months*
- to add to and subtract from *days* and *weekdays* duration in *days*
- to increment and decrement
- to find the difference as duration in days, months or days

Examples:

```
year y1 { 2026 };
y1 += years(2);
cout << static_cast<int>(y1) << endl;  // prints 2028
year y2 = y1 + years(2);
cout << static_cast<int>(y2) << endl;  // prints 2030
month m1 { June };
month m2 = m1 + months(3);
cout << static_cast<unsigned int>(m2) << endl; // prints 9
year y3 { 2025 }, y4 { -44 }; // the year of Julius Caesar's assassination
years from_Caesar_death = y3 – y4;  // 2069
```

Calendar (3)

If we write const *3.14*, it is stored as double value. To store it as float on 4 bytes, we need to write *3.14F* or *3.14f*. Similarly, constant *5UL* is handled as unsigned long long int.

In namespace *chrono 2026y* means that this is constant *year* with value *2026* and *30d* means that it is constant *day* with value *30*:

```
auto y1 { 2025y };
auto d1 { 30d };
```

Only lowercase *y* and *d* are allowed, *m* is not defined.

Class *year_month_day* (see https://en.cppreference.com/w/cpp/chrono/year_month_day.html) is a good tool to operate with calendar dates.

The 4 variants to define 21-Aug-2025 are:

```
year_month_day ymd1 { 2025y, August, 21d }, ymd2{ 2025y / August / 21d },
ymd3 { August / 21d / 2025y }, ymd4 { 21d / August / 2025y };
```

There are operator functions for comparison and to add to and subtract from *year_month_day* duration in *years* or *months* (but not days). Unfortunately, there is no possibility to find the difference between two objects *year_month_day*.

Methods *year()*, *month()* and *day()* are to retrieve the components of *year_month_day*. Method *ok()* checks the correctness.

Calendar (4)

Class *year_month_day* has one more constructor:

```
sys_time now = system_clock::now();
sys_days now_days = timepoint_cast<days>(now);
year_month_day today { now_days };
```

Thus we have converted the time read from clock into object of class *year_month_day*.

For backward conversion class *year_month_day* has operator overloader for casting into *sys_days*. Therefore:

```
year_month_day ymd { 2025y, August, 21d };
sys_days = ymd;
```

For comparing two objects from class *year_month_day* there are only two operators: == and <=>. Example:

```
auto res = ymd12 <=> ymd2;
if (res == strong_ordering::less) { …. }
```

Time zone

Method *system_clock::now()* returns UTC time, not the local time.

```
cout << system_clock::now << endl; // prints 2025-08-25 12:29:16.2318576
                                   // but the wristwatch shows 15:29
```

To get the local time you may use tools from C (see the previous slides). Az an alternative, use IANA time zone database copied into C++ v.20 library. Remark that on compilers for small systems this database may be nor available.

```
sys_time sys_time_utc = system_clock::now();
cout << sys_time_utc << endl;
try
{  // must be in try-catch block
   const time_zone* pEst = locate_zone("Europe/Tallinn");
   local_time est_time = pEst->to_local(sys_time_utc);
   cout << est_time << endl;
}
catch (const std::runtime_error& e)
{
   cout << "Time zone error: " << e.what() << '\n';
}
```