**Motivation**

Since DataFrame is a statistical library, it often deals with time-series data. So, it needs to keep track of time.

The most efficient way of indexing DataFrame by time is to use an index type of *time\_t* for second precision or *double* or *long long integer* for more precision. DateTime class provides a more elaborate handling of time. Also, it is a general handy DateTime object.

**Code structure**

Both the header (DateTime.h) and source (DateTime.cc) files are part of the DataFrame project. They are in the usual *include/Utils* and *src/Utils* directories.

**Build Instructions**

Follow the DataFrame build instructions.

**Example**

This library can have up to Nano second precision depending on what systems calls are available.

These are some example code:

DateTime now;

DateTime gmt\_now (DT\_TIME\_ZONE::GMT);

DateTime hk\_now (DT\_TIME\_ZONE:: AS\_HONG\_KONG);

cout << "Local Time is: " << now.string\_format (DT\_FORMAT::DT\_TM2) << std::endl;

cout << "GMT Time is: " << gmt\_now.string\_format (DT\_FORMAT::DT\_TM2) << std::endl;

double diff = now.diff\_seconds (gmt\_now);

now = 19721202;

gmt\_now = 19721210;

diff = now.diff\_weekdays (gmt\_now);

now.add\_days(3)

now.add\_weekdays(-2);

|  |
| --- |
|  |
|  |
|  |

For more examples see file *date\_time\_tester.cc*

**Types**

*enum class DT\_FORMAT : unsigned short int {*

*AMR\_DT = 1, // e.g. 09/16/99*

*AMR\_DT\_CTY = 2, // e.g. 09/16/1999*

*EUR\_DT = 3, // e.g. 16/09/99*

*EUR\_DT\_CTY = 4, // e.g. 16/09/1999*

*DT\_TM = 5, // e.g. 09/16/1999 13:51:04*

*SCT\_DT = 6, // e.g. Sep 16, 1999*

*DT\_MMDDYYYY = 7, // e.g. 09161999*

*DT\_YYYYMMDD = 8, // e.g. 19990916*

*DT\_TM2 = 9, // e.g. 09/16/1999 13:51:04.256*

*DT\_DATETIME = 10, // e.g. 20010103 09:31:15.124*

*DT\_PRECISE = 11 // e.g. 1516179600.874123908 = Epoch.Nanoseconds*

*ISO\_DT\_TM = 12, // e.g. 2015-05-05 13:51:04.000234*

*ISO\_DT = 13, // e.g. 2015-05-05*

*};*

These constants are used for formatting date/time into strings.

*enum class DT\_TIME\_ZONE : short int {*

*LOCAL = -2,*

*GMT = 0,*

*AM\_BUENOS\_AIRES = 1,*

*AM\_CHICAGO = 2,*

*AM\_LOS\_ANGELES = 3,*

*AM\_MEXICO\_CITY = 4,*

*AM\_NEW\_YORK = 5,*

*AS\_DUBAI = 6,*

*AS\_HONG\_KONG = 7,*

*AS\_SHANGHAI = 8,*

*AS\_SINGAPORE = 9,*

*AS\_TEHRAN = 10,*

*AS\_TEL\_AVIV = 11,*

*AS\_TOKYO = 12,*

*AU\_MELBOURNE = 13,*

*AU\_SYDNEY = 14,*

*BR\_RIO\_DE\_JANEIRO = 15,*

*EU\_BERLIN = 16,*

*EU\_LONDON = 17,*

*EU\_MOSCOW = 18,*

*EU\_PARIS = 19,*

*EU\_ROME = 20,*

*EU\_VIENNA = 21,*

*EU\_ZURICH = 22,*

*UTC = 23,*

*AS\_SEOUL = 24,*

*AS\_TAIPEI = 25,*

*EU\_STOCKHOLM = 26,*

*NZ = 27,*

*EU\_OSLO = 28,*

*EU\_WARSAW = 29,*

*EU\_BUDAPEST = 30*

*};*

These are the available time zones, used in a few methods and constructors.

*enum class DT\_WEEKDAY : unsigned char {*

*BAD\_DAY = 0,*

*SUN = 1,*

*MON = 2,*

*TUE = 3,*

*WED = 4,*

*THU = 5,*

*FRI = 6,*

*SAT = 7*

*};*

Week days: 1 - 7 (Sunday - Saturday), used by various methods.

*enum class DT\_MONTH : unsigned char {*

*BAD\_MONTH = 0,*

*JAN = 1,*

*FEB = 2,*

*MAR = 3,*

*APR = 4,*

*MAY = 5,*

*JUN = 6,*

*JUL = 7,*

*AUG = 8,*

*SEP = 9,*

*OCT = 10,*

*NOV = 11,*

*DEC = 12*

*};*

Months: 1 - 12 (January - December), used by various methods.

*enum class DT\_DATE\_STYLE : unsigned char {*

*YYYYMMDD = 1,*

*AME\_STYLE = 2,*

*EUR\_STYLE = 3,*

*ISO\_STYLE = 3*

*};*

These constants are used for parsing data

AME\_STYLE: MM/DD/YYYY

EUR\_STYLE: YYYY/MM/DD

ISO\_STYLE: YYYY-MM-DD

*using DateType = unsigned int* // YYYYMMDD

*using DatePartType = unsigned short int //* year, month etc.

*using HourType = unsigned short int* // 0 - 23

*using MinuteType = unsigned short int* // 0 - 59

*using SecondType = unsigned short int* // 0 - 59

*using MillisecondType = short int* // 0 - 999

*using MicrosecondType = int* // 0 - 999,999

*using NanosecondType = int //* 0 - 999,999,999

*using EpochType = time\_t* // Signed epoch

*using LongTimeType = long long int* // Nano seconds since epoch

**Methods**

*explicit DateTime (DT\_TIME\_ZONE tz = DT\_TIME\_ZONE::LOCAL) noexcept;*

A constructor that creates a DateTime initialized to now.

*tz*: Desired time zone from DT\_TIME\_ZONE above.

*explicit DateTime (DateType d,*

*HourType hr = 0,*

*MinuteType mn = 0,*

*SecondType sc = 0,*

*NanosecondType ns = 0,*

*DT\_TIME\_ZONE tz = DT\_TIME\_ZONE::LOCAL) noexcept;*

The constructor that creates a DateTime based on parameters passed.

*d*: Date e.g. 20180112

*hr*: Hour e.g. 13

*mn*: Minute e.g. 45

*sc*: Second e.g. 45

*ns*: Nano-second e.g. 123456789

*tz*: Desired time zone from DT\_TIME\_ZONE above.

*explicit DateTime (const char \*s,*

*DT\_DATE\_STYLE ds = DT\_DATE\_STYLE::YYYYMMDD,*

*DT\_TIME\_ZONE tz = DT\_TIME\_ZONE::LOCAL);*

The constructor that creates a DateTime by parsing a string and based on parameters passed.

Currently, the following formats are supported:

1. YYYYMMDD

AME\_STYLE:

(2) MM/DD/YYYY

(3) MM/DD/YYYY HH

(4) MM/DD/YYYY HH:MM

(5) MM/DD/YYYY HH:MM:SS

(6) MM/DD/YYYY HH:MM:SS.MMM

EUR\_STYLE:

(7) YYYY/MM/DD

(8) YYYY/MM/DD HH

(9) YYYY/MM/DD HH:MM

(10) YYYY/MM/DD HH:MM:SS

(11) YYYY/MM/DD HH:MM:SS.MMM

ISO\_STYLE:

(12) YYYY-MM-DD

(13) YYYY-MM-DD HH

(14) YYYY-MM-DD HH:MM

(15) YYYY-MM-DD HH:MM:SS

(16) YYYY-MM-DD HH:MM:SS.MMM

*s*: The string to be parsed

*ds*: String format from DT\_DATE\_STYLE above

*tz*: Desired time zone from DT\_TIME\_ZONE above.

*void set\_time (EpochType the\_time, NanosecondType nanosec = 0) noexcept;*

A convenient method, if you already have a DateTime instance and want to change the date/time quickly.

*the\_time*: Time as epoch

*nanosec*: Nano seconds

*void set\_timezone (DT\_TIME\_ZONE tz);*

Changes the time zone to desired time zone.

NOTE: This method is not multithread-safe. This method modifies the TZ environment

variable which changes the time zone for the entire program.

*tz*: Desired time zone

*DT\_TIME\_ZONE get\_timezone () const;*

Returns the current time zone.

DateTime &operator = (DateType rhs);

Sets self to right-hand-side.

*rhs*: A date e.g. dt = 20181215;

*DateTime &operator = (const char \*rhs);*

Sets self to right-hand-side.

Currently, the following formats are supported:

1) YYYYMMDD [LOCAL | GMT]

2) YYYYMMDD HH:MM:SS.MMM [LOCAL | GMT]

*rhs*: A date/time string e.g. dt = “20181215”;

*int dt\_compare(const DateTime &rhs) const;*

Compares self with right-hand-side and returns an integer result accordingly.

*rhs*: Another DateTime instance

*DateType date () const noexcept; // e.g. 20020303*

*DatePartType year () const noexcept; // e.g. 1990*

*DT\_MONTH month () const noexcept; // JAN - DEC*

*DatePartType dmonth () const noexcept; // 1 - 31*

*DatePartType dyear () const noexcept; // 1 - 366*

*DT\_WEEKDAY dweek () const noexcept; // SUN - SAT*

*HourType hour () const noexcept; // 0 - 23*

*MinuteType minute () const noexcept; // 0 - 59*

*SecondType sec () const noexcept; // 0 - 59*

*MillisecondType msec () const noexcept; // 0 - 999*

*MicrosecondType microsec () const noexcept; // 0 - 999,999*

*NanosecondType nanosec () const noexcept; // 0 - 999,999,999*

*EpochType time () const noexcept; // Like time()*

*LongTimeType long\_time () const noexcept; // Nano seconds since epoch*

These methods return the corresponding date/time parts.

*DatePartType days\_in\_month () const noexcept; // 28, 29, 30, 31*

It returns the number of days in the month represented in self

*double diff\_seconds (const DateTime &that) const;*

*double diff\_minutes (const DateTime &that) const noexcept;*

*double diff\_hours (const DateTime &that) const noexcept;*

*double diff\_days (const DateTime &that) const noexcept;*

*double diff\_weekdays (const DateTime &that) const noexcept;*

*double diff\_weeks (const DateTime &that) const noexcept;*

These return the diff including the fraction of the unit. This is why they return a double. The diff could be +/- based on "this - that"

*that*: Another instance of DateTime

*void add\_nanoseconds (long nanosecs) noexcept;*

*void add\_seconds (EpochType secs) noexcept;*

*void add\_days (long days) noexcept;*

*void add\_weekdays (long days) noexcept;*

*void add\_months (long months) noexcept;*

*void add\_years (long years) noexcept;*

These methods either advance or pullback the time accordingly. The parameter to these methods could be +/-.

*secs, days*: A positive or negative number representing the units to change time

*template<typename T>*

*void date\_to\_str (DT\_FORMAT format, T &result) const;*

*std::string string\_format (DT\_FORMAT format) const;*

These methods format the date/time into a string based on the format parameter

*T*: Type of string

*result*: a string instance to store the formatted date/time

*format*: String format parameter based on DT\_FORMAT above