Module05-07
C++ Boost：日期与时间

C++ Boost - 日期与时间

- 容器相关
- 字符串和文字处理
- 正则表达式
- 智能指针
- 函数对象相关
- 序列化
- → 日期与时间
- 多线程
- 网络

日期与时间

- 关于日期与时间库
  - boost.date_time 库支持 3 种基本的时间类型：
    - 时间点：在时间连续统 (continuum) 中的一个特定位置。
    - 时间长度：独立于时间连续统上任意点的一段时间长度。
    - 时间间隔：关联于时间连续统上某个特定点的一段时间长度。也称为时间段。

日期与时间 - 格里历

- 日期与时间：
  - 格里历 (Gregorian)
  - Posix 时间 (Posix Time)

日期与时间 - 格里历

- 关于格里历
  - 格里历日期系统提供了基于格里历的日期编程系统。格里历的第一次引入是在 1582 年，它修正了罗马儒略历的一个错误。
  - boost.date_time 实现的日历是一种 " 预想的格里历 " ，即将格里历推广至 1582 年首次采用格里历之前的时间。当前的实现支持从 1400-Jan-01 到 9999-Dec-31 的日期。有很多被引用的 1582 年以前的日期是采用罗马儒略历的，所以如果要求对历史日期进行准确的计算，就必须要小心。
  - 格里历系统的所有类型位于名字空间 boost::gregorian. 本库提供一个方便的头文件 boost/date_time/gregorian/gregorian_types.hpp ，它包含了本库所有类，但不带输入 / 输出。另一个头文件 boost/date_time/gregorian/gregorian.hpp 则包含所有类型以及输入 / 输出代码。
  - 类 boost::gregorian::date 是用户使用的主要时间类型

日期与时间 - 格里历

- 所涉及的内容：
  - 日期： date
  - 日期长度： date duration
  - 日期段： date period
  - 日期迭代器： date iterator
  - 日期生成器和算法： date generators/algorithms
  - 格里历： gregorian calendar

日期与时间 - 格里历 - Date

- Date
  - 类 boost::gregorian::date 是日期编程的主要接口。通常 date 类一旦构造出来就是不可变的，不过它也允许从另一个 date 进行赋值。创建 date 的方法包括：从时钟读取当前日期，使用 日期迭代器，以及 日期算法或生成器。
  - 在内部， boost::gregorian::date 是保存为一个 32 位整数类型的。该类特别被设计为不含虚函数。这样的设计可以进行高效的运算，以及处理大量日期时具有高效的内存使用率。
  - 构造一个 date 时将检查所有的输入，所以不可能构造一个无效的 date. 如 2001-Feb-29 就不可能被构造为一个 date. 多个派生自 std::out_of_range 的异常将被抛出以表示日期输入的哪个方面有错。注意，如果需要，可以用一个无效日期的特殊值来作为 '无效' 或 '空' 的 date.

日期与时间 - 格里历 - Date

- 构造 Date

```
date(greg_year, greg_month, greg_day);
date(const date& d);
date(special_values sv);
date(); // 构造一个not_a_date_time

// 从字符串构造
date from_string(std::string);
date from_undelimited_string(std::string);

// 从clock构造
date day_clock::local_day();
date day_clock::universal_day();
```

日期与时间 - 格里历 - Date

- 构造 Date 示例

```
using namespace boost::gregorian;
date d(2010, Feb, 28);
date d1(neg_infin);
date d2(pos_infin);
date d3(not_a_date_time);
date d4(max_date_time);
date d5(min_date_time);

date d6 = from_string("2010-02-18");
date d7 = from_undelimited_string("20101208");
cout << d6 << '/' << d7 << endl;

cout << day_clock::local_day() << endl;
cout << day_clock::universal_day() << endl;
```

日期与时间 - 格里历 - Date

- Date 访问函数

```
greg_year year() const;
greg_month month() const;
greg_day day() const;
greg_ymd year_month_day() const;
greg_day_of_week day_of_week() const;
greg_day_of_year day_of_year() const;
date end_of_month() const;

bool is_infinity() const;
bool is_neg_infinity() const;
bool is_pos_infinity() const;
bool is_not_a_date() const;
bool is_special() const;

special_value as_special() const;
long modjulian_day() const;
long julian_day() const;
int week_number() const;
```

日期与时间 - 格里历 - Date

- Date 其它操作

```
std::string to_simple_string(date d); // 如: 2010-Apr-12
std::string to_iso_string(date d); // 如: 20100412
std::string to_iso_extended_string(date d);// 如: 2010-04-12

friend ostream& operator<<(ostream& os, const date& d);
friend istream& operator>>(istream& is, date& d);

operator==, operator!=,
operator>, operator<,
operator>=, operator<=

date operator+(date_duration) const;
date operator-(date_duration) const;
date_duration operator-(date) const;

tm to_tm(date);
date date_from_tm(tm datetm);
```

日期与时间 - 格里历 - Date Duration

- 关于 Date Duration（日期长度）
  - 类 boost::gregorian::date_duration 是一个简单的日子计数器，用于 gregorian::date 的运算。日期间隔可正可负。
  - date_duration 类被 typedef 为 boost::gregorian 名字空间中的 days.

日期与时间 - 格里历 - Date Duration

- Date Duration

```
date_duration(long);
date_duration(special_values sv);

long days() const;
bool is_negative() const;
static date_duration unit();
bool is_special() const;

operator<<, operator>> // I/O
operator==, operator!=, operator>,
operator<, operator>=, operator<=

date_duration operator+(date_duration) const;
date_duration operator-(date_duration) const;

// 其它日期间隔
months(int num_of_months);
years(int num_of_years);
weeks(int num_of_weeks);
```

日期与时间 - 格里历 - Date Duration

- Date Duration 示例

```
date d1 = day_clock::local_day();
date d2(2010,Jan, 2);
date_duration dd = d1 - d2;
cout << d1 << " - " << d2 << " = " << dd << " days" << endl;

date_duration dd2(36);
d2 += dd2;
cout << d2 << endl;

months m1(18);
d1 += m1;  // 加上18个月
cout << d1 << endl;
```

日期与时间 - 格里历 - Date Period

- 关于 Date Period
  - 类 boost::gregorian::date_period 提供了对两个日期之间的范围的直接表示。日期段可以通过简化程序的条件判断逻辑来简化一些计算类型。例如，测试某个日期是否在某个没有规律的时间表如周末或假日中，就可以用一组日期段来实现。有多种方法来判断一个 date_period 是否与另一个 date period 交叉，以及生成重叠的日期段。
  - 由相同的开始日期和结束日期所创建的日期段，称为零长度日期段。零长度日期段被认为是无效的（构造一个无效的日期段是完全合法的）。对于这些日期段， last 点总是比 begin 小一个单元。

日期与时间 - 格里历 - Date Period

- Date Period

```
date_period(date, date);
date_period(date, days);
date_period(date_period);

date begin();
date last();
date end();
days length();
bool is_null();
bool contains(date);
bool contains(date_period);
bool intersects(date_period);
date_period intersection(date_period);
date_period is_adjacent(date_period);
date_period is_after(date);
date_period is_before(date);
```

日期与时间 - 格里历 - Date Period

- Date Period

```
    date_period merge(date_period);
    date_period span(date_period);//begin = min(p1.begin,
p2.begin), end = max(p1.end , p2.end)
    date_period shift(days); // begin + days, end + days
    date_period expand(days); // begin -days, end + days

    std::string to_simple_string(date_period dp);
    operator<< operator>>
    operator==, operator!=,
    operator>, operator<
```

日期与时间 - 格里历 - Date Period

- Date Period 示例
  - (DEMO Using date_time example: date period calculation example)

日期与时间 - 格里历 - Date Iterator

- 关于 Date Iterator
  - 日期迭代器为对日期进行迭代提供了一个标准机制。日期迭代器是一种 双向迭代器 并可用于大多日期集合以及其它日期生成任务。
  - 这里的所有迭代器均派生自 boost::gregorian::date_iterator

日期与时间 - 格里历 - Date Iterator

- Date Iterator

```
class date_iterator;  // 所有日期的迭代器的基类（抽象对象）

// 以日为单位迭代，迭代的步距day_count，默认为1日
day_iterator(date start_date, int day_count = 1);

// 以周为单位迭代，迭代的步距week_offset，默认为1周
week_iterator(date start_date, int week_offset = 1);

// 以月为单位迭代，迭代的步距month_offset，默认为1月
month_iterator(date start_date, int month_offset = 1);

// 以年为单位迭代，迭代的步距year_offset，默认为1年
year_iterator(date start_date, int year_offset = 1);
```

日期与时间 - 格里历 - Date Iterator

- Date Iterator 示例
  - (DEMO Using date_time example: print month)

日期与时间 - 格里历 - Date Generators

- 关于 Date Generators
  - 日期算法或生成器是一些生成其它日期或日期时间表的工具。生成器函数以日期的某些部分如月份和日子开始，并通过提供其它部分以生成一个具体的日期。这使得程序员可以表示象 " 二月的第一个星期天 " 这样的概念，并在提供一个或多个年份后创建一组具体的日期。
  - 同时还提供了生成一个日期或计算一段日期长度的独立函数。这些函数以一个 date 对象和一个 weekday 对象为参数。

- Date Generators - 日期生成器类型

```
class year_based_generator { // Base class
    virtual date get_date(greg_year year) = 0;
};

last_day_of_the_week_in_month(greg_weekday, greg_month);
date get_date(greg_year year);
// examples:
last_day_of_the_week_in_month lwdm(Monday,Jan);
date d = lwdm.get_date(2002); //2002-Jan-28

first_day_of_the_week_in_month(greg_weekday, greg_month);
date get_date(greg_year year);
// examples:
first_day_of_the_week_in_month fdm(Monday,Jan);
date d = fdm.get_date(2002);//2002-Jan-07
```

- Date Generators - 日期生成器类型（续 1 ）

```
nth_day_of_the_week_in_month(week_num, greg_weekday,
greg_month);
date get_date(greg_year year);
// examples:
typedef nth_day_of_the_week_in_month nth_dow;
nth_dow ndm(nth_dow::third, Monday,Jan);
date d = ndm.get_date(2002);//2002-Jan-21

partial_date(greg_day, greg_month);
date get_date(greg_year year);
// examples:
partial_date pd(1,Jan);
date d = pd.get_date(2002); //2002-Jan-01

first_day_of_the_week_after(greg_weekday);
date get_date(date d);
// examples:
first_day_of_the_week_after fdaf(Monday);
date d = fdaf.get_date(date(2002,Jan,1)); //2002-Jan-07
```

日期与时间 - 格里历 - Date Generators

- Date Generators - 日期生成器类型（续 2 ）

```
first_day_of_the_week_before(greg_weekday);
date get_date(date d);
// examples:
first_day_of_the_week_before fdbf(Monday);
date d = fdbf.get_date(date(2002,Feb,1)); //2002-Jan-28
```

- Date Generators - 日期生成器算法函数

```
// 计算从给定日期到给定周日的天数。
days days_until_weekday(date, greg_weekday);
// examples:
date d(2004,Jun,1); // 星期二
greg_weekday gw(Friday);
days_until_weekday(d, gw); // 3天

// 计算从给定日期到前一个给定周日的天数。
days days_before_weekday(date, greg_weekday);
// examples:
date d(2004,Jun,1); // 星期二
greg_weekday gw(Friday);
days_before_weekday(d, gw); // 4天
```

- Date Generators - 日期生成器算法函数（续）

```
// 生成一个 date 对象，表示给定日期之后的某个周日的日期。
date next_weekday(date, greg_weekday);
// examples:
date d(2004,Jun,1); // 星期二
greg_weekday gw(Friday);
next_weekday(d, gw); // 2004-Jun-4

// 生成一个 date 对象，表示给定日期之前的某个周日的日期。
date previous_weekday(date, greg_weekday);
// examples:
date d(2004,Jun,1); // 星期二
greg_weekday gw(Friday);
previous_weekday(d, gw); // 2004-May-28
```

日期与时间 - 格里历 - Date Generators

- Date Generators 示例
  - (DEMO Using date_time example: print holidays)

日期与时间 - 格里历 - Gregorian Calendar

- 关于 Gregorian Calendar
  - 类 boost::gregorian::gregorian_calendar 实现了创建格里历日期系统所需的函数。包括将日期的年 - 月 - 日格式转换为天数表示法以及相反的转换。
  - 对于多数用途，这个类只是被 gregorian::date 访问而不是由用户直接使用。不过，也有一些有用的函数可能被象 end_of_month_day 这样的函数使用。

- Gregorian Calendar

```
class gregorian_calendar {
    // 返回星期几(0==星期天，1==星期一，等等)
    // 参见 gregorian::date day_of_week
    static short day_of_week(ymd_type);

    // 将 ymd_type 转换为天数。
    // 该天数是一个从 epoch 起计的绝对数字。
    static date_int_type day_number(ymd_type);

    // 给定年份和月份，确定该月最后一天。
    static short end_of_month_day(year_type, month_type);

    // 将天数转换为 ymd 结构。
    static ymd_type from_day_number(date_int_type);

    // 返回 true 如果指定年份是闰年。
    // gregorian_calendar::is_leap_year(2000) //--> true
    static bool is_leap_year(year_type);
};
```

日期与时间 - Posix 时间

- 日期与时间：
  - 格里历 (Gregorian)
  - Posix 时间 (Posix Time)

日期与时间 - Posix 时间

- 关于 Posix 时间
  - 定义一个无调整的、分辨率为纳秒 / 微秒级的、具有稳定的计算特性的时间系统。如果是纳秒级精度的，则每个 ptime 使用 96 位的底层存储，而微秒级精度则每个 ptime 使用 64 位存储 (详情请见 构建选项)。该时间系统使用格里历来实现时间表示中的日期部分。
  - 包含的头文件 (下面两个头文件之一)：
    - #include "boost/date_time/posix_time/posix_time.hpp" // 包含所有类型和 i/o
    - #include "boost/date_time/posix_time/posix_time_types.hpp" // 只有类型没有 i/o

日期与时间 - Posix 时间

- 涉及的内容
  - 时间： ptime
  - 时间长度： time duration
  - 时间段： time period
  - 时间迭代器： time iterator

日期与时间 - Posix 时间 - ptime

- 关于 ptime
  - 类 boost::posix_time::ptime 是处理时间点的主要接口。通常，ptime 类在构造后就不再改变，不过它也允许进行赋值。
  - 类 ptime 依赖于 gregorian::date，作为时间点的日期部分的接口。
  - 其它创建时间的方法还包括有 时间迭代器。

日期与时间 - Posix 时间 - ptime

- 构造 ptime

```
// ptime 构造函数
ptime();
ptime(date, time_duration);
ptime(special_values sv);
ptime(ptime);

//从字符串构造
ptime time_from_string(std::string);
ptime from_iso_string(std::string);

//从时钟构造
ptime second_clock::local_time();
ptime second_clock::universal_time();
ptime microsec_clock::local_time();
ptime microsec_clock::universal_time();

// 使用转换函数进行构造
ptime from_time_t(time_t t);
ptime from_ftime<ptime> (FILETIME ft);
```

日期与时间 - Posix 时间 - ptime

- 构造 ptime 示例

```
ptime p; // p => not_a_date_time

// 通过日期和时间间隔
ptime t1(date(2002, Jan, 10), time_duration(1, 2, 3));
ptime t2(date(2002, Jan, 10), hours(1) + nanosec(5));

// spec_values
ptime d1(neg_infin);
ptime d2(pos_infin);
ptime d3(not_a_date_time);
ptime d4(max_date_time);
ptime d5(min_date_time);

// 从字符串构造
std::string ts("2002-01-20 23:59:59.000");
ptime t(time_from_string(ts));

// 从不带分隔的字符串构造。
std::string ts("20020131T235959");
ptime t(from_iso_string(ts));
```

日期与时间 - Posix 时间 - ptime

- 访问 ptime

```
date date(); //取出时间的日期部分。
// examples:
date d(2002, Jan, 10);
ptime t(d, hour(1));//t.date() --> 2002-Jan-10;
time_duration time_of_day();// 取出在一天中的时间偏移。
// examples:
date d(2002, Jan, 10);
ptime t(d, hour(1)); // t.time_of_day() --> 01:00:00;

//返回 true 如果 ptime 是正的或负的无限。
bool is_infinity() const;
// examples:
ptime pt(pos_infin);
pt.is_infinity(); // --> true
// 返回 true 如果 ptime 为负无限。
bool is_neg_infinity() const;
// examples:
ptime pt(neg_infin);
pt.is_neg_infinity(); // --> true
```

日期与时间 - Posix 时间 - ptime

- 访问 ptime（续）

```
// 返回 true 如果其值不是一个 ptime
bool is_not_a_date_time() const;
// examples:
ptime pt(not_a_date_time);
pt.is_not_a_date_time(); // --> true

//返回 true 如果 ptime 为某个 special_value
bool is_special() const;
// examples:
ptime pt(pos_infin);
ptime pt2(not_a_date_time);
ptime pt3(date(2005, Mar, 1), hours(10));
pt.is_special(); // --> true
pt2.is_special(); // --> true
pt3.is_special(); // --> false
```

日期与时间 - Posix 时间 - ptime

- 字符串表示 ptime

```
// 转换为 YYYY-mmm-DD HH:MM:SS.fffffffff 字符串,
// 其中 mmm 为3字符月份名。秒的小数部分仅在非零时包含。
// 如: 2002-Jan-01 10:00:01.123456789
std::string to_simple_string(ptime);

//转换为 YYYYMMDDTHHMMSS,fffffffff 其中 T 为日期-时间分隔符。
//如: 20020131T100001,123456789
std::string to_iso_string(ptime);

//转换为 YYYY-MM-DDTHH:MM:SS,fffffffff 其中 T 为日期-时间分隔符。
//如: 2002-01-31T10:00:01,123456789
std::string to_iso_extended_string(ptime);
```

日期与时间 - Posix 时间 - ptime

- ptime 操作符

```
operator<<, operator>>
// example:
ptime pt(not_a_date_time);
stringstream ss("2002-Jan-01 14:23:11");
ss >> pt;
std::cout << pt; // "2002-Jan-01 14:23:11"

operator==, operator!=,
operator>, operator<,
operator>=, operator<=;

// 返回加上日期偏移后的 ptime
ptime operator+(days);
// example:
date d(2002,Jan,1);
ptime t(d,minutes(5));
days dd(1);
ptime t2 = t + dd;
```

日期与时间 - Posix 时间 - ptime

- ptime 操作符（续 1 ）

```
// 返回减去日期偏移后的 ptime
ptime operator-(days);
// example:
date d(2002,Jan,1);
ptime t(d,minutes(5));
days dd(1);
ptime t2 = t - dd;

// 返回加上时间长度后的 ptime
ptime operator+(time_duration);
// example:
date d(2002,Jan,1);
ptime t(d,minutes(5));
ptime t2 = t + hours(1) + minutes(2);
```

日期与时间 - Posix 时间 - ptime

- ptime 操作符（续 2 ）

```
// 返回减去时间长度后的 ptime
ptime operator-(time_duration);
// example:
date d(2002,Jan,1);
ptime t(d,minutes(5));
ptime t2 = t - minutes(2);

// 获得两个时间之间的差距。
time_duration operator-(ptime);
// example:
date d(2002,Jan,1);
ptime t1(d,minutes(5));
ptime t2(d,seconds(5));
time_duration t3 = t2 - t1;//结果为负
```

日期与时间 - Posix 时间 - ptime

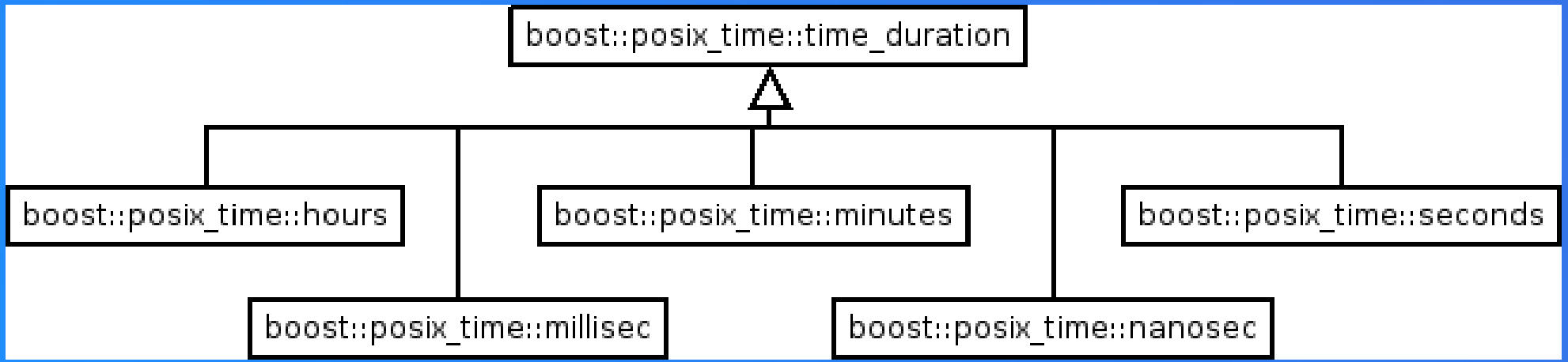
- ptime 与 struct tm 、 FILETIME 、 time_t 的转换

```
tm to_tm(ptime);
date date_from_tm(tm datetm);
tm to_tm(time_duration);

ptime from_time_t(std::time_t);

ptime from_ftime<ptime>(FILETIME);
```

- 关于 Time Duration
  - 类 boost::posix_time::time_duration 是负责表示时间长度的基本类型。时间长度可正可负。通常 time_duration 类提供一个带有小时、分、秒的计数的构造函数，就象在后面的代码片段所示。 time_duration 的分辨率可以在编译期配置。
  - 从基类 time_duration 派生了几个类用于按不同的分辨率进行调整：



日期与时间 - Posix 时间 - Time Duration

- 关于 Time Duration
  - 简单示例：

```
using namespace boost::posix_time;
time_duration td = hours(1) + seconds(10); //01:00:10
td = hours(1) + nanoseconds(5); //01:00:00.000000005

// 注意，是否存在更高分辨率的类(如：纳秒)取决于库的安装。
// 另一个方法是利用 time_duration 的 ticks_per_second()
// 方法来编写可移植的代码，这样可以不管库是如何编译的。
// 以下是计算分辨率无关的计数值的常用方式。
count * (time_duration_ticks_per_second /
 count_ticks_per_second);

// 例如，假设我们想用表示十分之一秒的计数值来构造时间。即每一 tick 为0.1秒。
int number_of_tenths = 5;
//创建一个分辨率无关的计数值 -- 除以10，因为一秒分为10份
int count = number_of_tenths *
 (time_duration::ticks_per_second() / 10);
time_duration td(1, 2, 3, count); //01:02:03.5 //与分辨率设置无关
```

日期与时间 - Posix 时间 - Time Duration

- 构造 Time Duration- 构造函数

```
time_duration(hours, minutes, seconds, fractional_seconds);
// example:
time_duration td(1,2,3,9);
//1 小时 2 分 3 秒 9 纳秒
time_duration td2(1,2,3,123456789);
time_duration td3(1,2,3,1000);
// 对于微秒分辨率(6位)
// td2 => "01:04:06.456789"
// td3 => "01:02:03.001000"
// 对于纳秒分辨率(9位)
// td2 => "01:02:03.123456789"
// td3 => "01:02:03.000001000"

time_duration(special_value sv);
```

日期与时间 - Posix 时间 - Time Duration

- 构造 Time Duration

```
// 基于计数的构造
time_duration hours(long); // 小时数
time_duration minutes(long);// 分钟数
time_duration seconds(long);// 秒数
time_duration milliseconds(long);// 毫秒数
time_duration microseconds(long);// 微秒数
time_duration nanoseconds(long);// 纳秒数

// 从带有分隔符的字符串构造。
// 注: 秒数中超界的小数部分将被截断。
// 如: "1:02:03.123456999" => 1:02:03.123456.
// 该行为受本库编译期的精度影响

// std::string ts("23:59:59.000");
// time_duration td(duration_from_string(ts));
time_duration duration_from_string(std::string);
```

日期与时间 - Posix 时间 - Time Duration

- 访问 Time Duration
  - 提供如 hour() 、 nanosecond() 等函数访问 time_duration

日期与时间 - Posix 时间 - Time Duration

- 以字符串形式表示 Time Duration

```
// 转换为 HH:MM:SS.fffffffff 其中 fff 为秒的分数部分，
// 仅当非零时包含。
// 10:00:01.123456789
std::string to_simple_string(time_duration);

// 转换为 HHMMSS,fffffffff.
// 100001,123456789
std::string to_iso_string(time_duration);
```

日期与时间 - Posix 时间 - Time Duration

- Time Duration 操作符

```
// 流操作符。
// time_duration td(0,0,0);
// stringstream ss("14:23:11.345678");
// ss >> td;
// std::cout << td; // "14:23:11.345678"
operator<<, operator>>;

operator==, operator!=, operator>,
operator<, operator>=, operator<=;

// 加上时间长度。
// time_duration td1(hours(1)+minutes(2));
// time_duration td2(seconds(10));
// time_duration td3 = td1 + td2;
time_duration operator+(time_duration);
```

日期与时间 - Posix 时间 - Time Duration

- Time Duration 操作符（续）

```
// 减去时间长度。
// time_duration td1(hours(1)+nanoseconds(2));
// time_duration td2 = td1 - minutes(1);
time_duration operator-(time_duration);

// 用整数值去除以时间长度。忽略余数。
// hours(3)/2 == time_duration(1,30,0);
// nanosecond(3) / 2 == nanosecond(1);
time_duration operator/(int);

// 用整数值乘以时间长度。
// hours(3)*2 == hours(6);
time_duration operator*(int);
```

日期与时间 - Posix 时间 - Time Duration

- Time Duration 转换成 struct tm

```
tm to_tm(time_duration);

// 将 time_duration 对象转换为 tm 结构的函数。
// 字段 tm_year, tm_mon, tm_mday,
// tm_wday, tm_yday 均设为零。
// 字段 tm_isdst 设为 -1.

// time_duration td(1,2,3);
// tm td_tm = to_tm(td);

/* tm_year => 0
 tm_mon  => 0
 tm_mday => 0
 tm_wday => 0
 tm_yday => 0
 tm_hour => 1
 tm_min  => 2
 tm_sec  => 3
 tm_isddst => -1 */
```

日期与时间 - Posix 时间 - Time Period

- 关于 Time Period
  - 类 boost::posix_time::time_period 提供了对两个时间点间的范围的表示法。时间段可以通过简化程序的条件判断逻辑来简化一些计算类型。
  - 由相同的开始时间点和结束时间点所创建的时间段，称为零长度时间段。零长度时间段被认为是无效的（构造一个无效的时间段是完全合法的）。对于这些时间段， last 点总是比 begin 小一个单元。

日期与时间 - Posix 时间 - Time Period

- 构造 Time Period

```
time_period(ptime, ptime);
// 创建一个[begin, end) 时间段。如果end <= begin 则时间段定义为无效。
// example:
// date d(2002,Jan,01);
// ptime t(d, seconds(10)); //午夜后10秒
// time_period tp(t, hours(3));

time_period(ptime, time_duration);
// 创建一个[begin, begin+len) 时间段,
// 其中end 为begin+len.
// 如果len <= zero 则时间段定义为无效。
// example:
// date d(2002,Jan,01);
// ptime t1(d, seconds(10)); //午夜后10秒
// ptime t2(d, hours(10)); //午夜后10小时
// time_period tp(t1, t2);

time_period(time_period rhs); // 复制构造函数
```

日期与时间 - Posix 时间 - Time Period

- Time Period 操作

```
    time_period shift(time_duration); // begin() + duration,
end() + duration
    time_period expand(time_duration);// begin() - duration,
end() + duration

    ptime begin();
    ptime last();
    ptime end();

    time_duration length();

    bool is_null();
    bool contains(ptime);
    bool contains(time_period);
    bool intersects(time_period);
    time_period intersection(time_period);

    time_period merge(time_period);
    time_period span(time_period);
```

日期与时间 - Posix 时间 - Time Period

- Time Period 字符串表示

```
    std::string to_simple_string(time_period dp);

    // 转换为 [YYYY-mmm-DD hh:mm:ss.fffffffff/YYYY-mmm-DD
hh:mm:ss.fffffffff]
    // 字符串，其中 mmm 为3字符的月份名。
    //
    //[2002-Jan-01 01:25:10.000000001/2002-Jan-31
01:25:10.123456789]
```

日期与时间 - Posix 时间 - Time Period

- Time Period 操作符

```
// 流操作符。
// time_duration td(0,0,0);
// stringstream ss("14:23:11.345678");
// ss >> td;
// std::cout << td; // "14:23:11.345678"
operator<<, operator>>;

operator==, operator!=, operator>,
operator<, operator>=, operator<=
```

日期与时间 - Posix 时间 - Time Period

- Time Period 示例
  - (DEMO Using date_time example: time periods)

日期与时间 - Posix 时间 - Time Iterator

- 关于 Time Iterator
  - 时间迭代器提供了对时间进行迭代的一种机制。时间迭代器类似于 双向迭代器。不过， time_iterators 与标准迭代器的区别在于，它不存在底层的序列，只有计算用的函数。此外， time_iterators 可以与 类 ptime 的实例直接比较。因此不需要一个表示迭代结束点的迭代器，可以直接使用时间点。

日期与时间 - Posix 时间 - Time Iterator

- Time Iterator 示例

```
#include "boost/date_time/posix_time/posix_time.hpp"
#include <iostream>
int main() {
    using namespace boost::gregorian;
    using namespace boost::posix_time;
    date d(2000, Jan, 20); ptime start(d);
    ptime end = start + hours(1);
    time_iterator titr(start, minutes(15)); //每次递增15分钟
    //生成 00:00:00, 00:15:00, 00:30:00, 00:45:00
    while (titr < end) {
        std::cout << to_simple_string(*titr) << std::endl;
        ++titr;
    }
    std::cout << "Now backward" << std::endl;
    //produces 01:00:00, 00:45:00, 00:30:00, 00:15:00
    while (titr > start) {
        std::cout << to_simple_string(*titr) << std::endl;
        --titr;
    }
}
```

日期与时间 - More stuffs...

- 作为一个成熟的日期时间库， Boost.Date_Time 库除了提供 gregorian::date 和 posix_time::ptime 接口之外，还提供了：
  - local_time ： 关注时区、 本地时间的操作
  - date 、 time 的 I/O
  - 时间的序列化