Modern C++11, 14, 17

Part 1 – Fundamental

A null pointer, enum class, if initializer, range for, delegating constructor, string view, attribute
B random, regular expression, filesystem, chrono, timespec, boost posixtime
C boost timer, boost server/client
D smart pointer
E function, binding, lambda, overloading lamba
F type traits, tuple, variant, reference, optional, any
G more about template programming

Part 2 – Thread library

A using `thread` object
B locks to avoid race condition
C synchronization mechanisms
D synchronization models
E speed, divide and conquer, thread local storage

A1. Null pointer

Unlike error-prone `NULL` macro which is a literal 0, `nullptr` is strongly-typed, it can only be assigned to variables of type :

- raw pointer `T* variable = nullptr;`
- function pointer `T2 (*variable)(const T0&, T1) = nullptr;`
- member function pointer `T2 (T::*variable)(const T0&, T1) = nullptr;`

```
void fct(int);
void fct(T*);
fct(0);                     // ambiguous call, compile error
fct(NULL);                  // ambiguous call, compile error
fct(nullptr);               // resolves to void fct(T*)
```

A2. Enum class

There are 3 problems with `enum` :

- `enum` is not scoped which means two enums having same `value` will result in ambiguity
- `enum` size is not customizable which means that we cannot specify the size of `enum`
- `enum` is only partially type safe which means given a function that takes `enum` as argument ...
- we cannot feed it with an integer ...
- however we can process it like an integer inside the function (such as addition and outstreaming)

```
    enum enum0 { value0, value1, value2 };
//  enum enum1 { value0, value1, value2 }; // compile error, unresolved ambiguity, value0/1/2 are declared in enum0 already
    void fct(const enum0& x)
    {
        std::cout << x + 123; // addition and outstreaming as an int is possible
    }

//  fct(0); // compile error, cannot convert int as enum
    fct(value0);
    fct(value1);
    fct(value2);
```

In order to solve these problems, we introduce `enum class`, which is scoped. Its size can be specified by inheritance from base class. It cannot be processed as `int` inside function, unless we `static_cast<int>` it explicitly.

```
    // (1) enum class are scoped, thus no ambiguity below
    enum class enum1                 { value0, value1, value2 };
    enum class enum2 : std::uint16_t { value0, value1, value2 }; // (2) we can specify the underlying size
    enum class enum3 : std::uint32_t { value0, value1, value2 };
    enum class enum4 : std::uint64_t { value0, value1, value2 };
//  enum class enum4 : public std::uint64_t {value0, value1, value2 };  ... compile error, cannot declare public, why?

    template<typename ENUM>
    void check_underlying()
    {
        std::cout << "\nenum is_same : " << std::is_same_v<std::underlying_type_t<ENUM>, std::uint8_t>
                                         << std::is_same_v<std::underlying_type_t<ENUM>, std::uint16_t>
                                         << std::is_same_v<std::underlying_type_t<ENUM>, std::uint32_t>
                                         << std::is_same_v<std::underlying_type_t<ENUM>, std::uint64_t>;
    }

//  void fct(const enum1& x) { std::cout << "\nenum1 value = " << x; }  // (3) compile error, cannot process enum class as int
    void fct(const enum1& x) { std::cout << "\nenum1 value = " << static_cast<int>(x); }
    void fct(const enum2& x) { std::cout << "\nenum2 value = " << static_cast<int>(x); }
    void fct(const enum3& x) { std::cout << "\nenum3 value = " << static_cast<int>(x); }
    void fct(const enum4& x) { std::cout << "\nenum4 value = " << static_cast<int>(x); }


    print_type<enum0>(); // 0010
    print_type<enum1>(); // 1000
    print_type<enum2>(); // 0100
    print_type<enum3>(); // 0010
    print_type<enum4>(); // 0001
    fct(value0);         // print 123
    fct(value1);         // print 124
    fct(value2);         // print 125
    fct(enum1::value0);  // print enum1 0
    fct(enum1::value1);  // print enum1 1
    fct(enum1::value2);  // print enum1 2
```

A3. If initializer

We can add initializer in if condition for sake of simplicity. Useful in `std::string::find`, `std::set::insert` and `std::map::find` :

- ```
  if (auto pos = str.find(pattern); pos!=std::string::npos) str.substr(pos, size)
  else ...
  ```
- ```
  if (auto [iter, flag] = set.insert(key); flag) iter-> ...
  ```
- ```
  if (auto iter = map.find(key); iter!=map.end()) iter->second ...
  else map[key] = ...
  ```

A4. Range-for

Range-for iterates through all elements in a container :

```
std::vector<heavy_type> vec(100);
for(const auto& x : vec) { x.read_only(); }
```

Range-based for-loop is applicable to a container providing that the following are offered :

- `iterator container<T>::begin()`
- `iterator container<T>::end()`
- `void iterator::operator++()`
- `void iterator::operator!=(const iterator& rhs)`
- `type& iterator::operator*()`

A5. Delegating constructor

Constructor is allowed to call another constructor of the same class.

```
struct my_class
{
    my_class(int x) {}                                      // This is called target constructor.
    my_class(const std::string& s) : my_class(stoi(s)) {}   // This is called delegating constructor.
};
```

A6. String view

String view is :

- a thin wrapper representing a `const` view of string
- while offering nearly full string-function, it is either ...
- implemented as two pointers, or ...
- implemented as one pointer plus a size
- constructed from an array or a string as :

```
char arr[5] = {'a','b','c','d','e'};
std::string str = "abcde";

std::string_view sv0(arr, std::size_of(arr));
std::string_view sv0(str);
```

A7. Attribute specifier

`[[maybe_unused]]`	suppress `gcc` compile error due to unused variables
`[[likely]]`	hint to compiler so that it can place the branch with higher prob in next step without jump
`[[unlikely]]`	hint to compiler so that it can place the branch with higher prob in next step without jump
`[[nodiscard]]`	compiler warning if a caller to a function does not handle/store the return value
```

B1. Random number

There are two parts : (1) random number engine, (2) random distribution.

```
std::default_random_engine engine;
std::normal_distribution<double> normal(mean, stddev);
std::poisson_distribution<int> poisson(lambda_t);

double x = 0, xx = 0;
double y = 0, yy = 0;
for(int n=0; n!=N; ++n)
{
     auto a =  normal(engine); x += a; xx += a*a;
     auto b = poisson(engine); y += b; yy += b*b;

}
std::cout << "\nmean = " << x/N << " stdd = " << xx/N – (x/N)*(x/N);
std::cout << "\nmean = " << y/N << " stdd = " << yy/N – (y/N)*(y/N);
```

B2. Regular expression (Regex)

Regular expression is a simple language that defines string pattern for text matching.

1. ECMAScript grammer

	`hello`	single occurrence of template "hello"
	`[aeiou]`	single occurrence of char within a set
	`[a-z]`	single occurrence of char within a range
	`[a-zA-z0-9]`	single occurrence of char within a set of range
	`[a-zA-z0-9]+`	one or more occurrence of char within a set of range (i.e. it doesn't match an empty string)
	`[a-zA-z0-9]*`	zero or more occurrence of char within a set of range (i.e. it does match an empty string)
1-7	`[a-zA-z0-9]?`	zero or one occurrence of char within a set of range
	`[ABCa-z]{3}`	occurrence of exactly 3 times
	`[ABCa-z]{3,}`	occurrence of 3 or more times
	`[ABCa-z]{3,6}`	occurrence of 3-6 times
	`.`	wildcard
	`\`	escape, remove the special meaning of the following character, for example `[a-z\(\)]+`
8-11	`[^A-Z]+`	exclude `A-Z`, matches `a#$%457sx, 43g2@#$sfe ...`
	`a|b+`	logical or, matches `a, b, bb, bbb, bbbb ...`
	`abc(de)?[0-9]*z`	matches `abcz, abcdez, abc135z, abcde2468z, abc01248z`

Example : old MSDOS filename :

```
[a-zA-Z_][a-zA-Z0-9_]*\.[a-zA-Z0-9]{1,3}

// Note : \ becomes \\
// one escape for regex
// one escape for C++
std::string regex("[a-zA-Z_][a-zA-Z0-9_]*\\.[a-zA-Z0-9]{1,3}");
```

2. Usage of regex library

- regex iterator allows iteration through all matched substrings in `std::string`
- construction regex iterator and `operator++()` of invoke pattern search for next match (both operations take time)
- iterator pauses at the next matched substring

```
    #include<regex>
    std::string str(".....");
    std::string pattern("[a-zA-Z_][a-zA-Z0-9_]*\\.[a-zA-Z0-9]{1,3}");
    std::regex  rgx(pattern);                                        // compilation of regex, takes time
//  std::regex  rgx(pattern, std::regex_constants::icase);           // ignore case

    std::sregex_iterator iter(str.begin(), str.end(), rgx);          // construct begin iterator
    std::sregex_iterator iter_end;                                   // construct   end iterator (when not init)
    while(iter!=iter_end) std::cout << *iter;
```

When using C++ regex, beware that :

- construction of regex objects takes time, please limit creation of regex objects,
- most regex errors cannot be detected until runtime, hence exception handling is needed.

Sharing of regex 101

by Alu, Yubo

wild card

`.`	wild card
`\`	escape for special character

quantifier

`?`	zero or one occurence
`*`	zero or more occurences (greedy matching, i.e. find the longest match)
`+`	one or more occurences (greedy matching, i.e. find the longest match)
`*?`	zero or more occurences (lazy matching, i.e. every match is a match)
`+?`	one or more occurences (lazy matching, i.e. every match is a match)
`{3,5}`	specified number of occurences

For example `"ABC" "DEF" "GHIJ"`
results in one match for `".+"`
results in three matches for `".+?"`

meta character

`^`	start of line (goto start of line in `vim` is `0`)
`$`	end of line (goto end of line in `vim` is also `$`)
`[abc]`	optional character
`[abc]+`	optional character
`[a-zA-Z]+`	optional character
`[^a-zA-Z]+`	optional character, `^` means *NOT*

capture and grouping

`(?:xxx)`	grouping without capture
`(xxx)`	capture group
`\1`	recall captured group 1
`\2`	recall captured group 2

alias

`\d`	equivalent to `[0-9]`
`\D`	equivalent to `[^0-9]`
`\w`	equivalent to `[a-zA-Z0-9]`
`\w+`	equivalent to `[a-zA-Z0-9]+` which represents a word delimited by other characters
`\W`	equivalent to `[^a-zA-Z0-9]`
`\b`	equivalent cursor in `vim` insert mode, it means word boundary
`\s`	equivalent to `[\t\r\n\v\f]` which represents delimitors
`\S`	equivalent to `[^\t\r\n\v\f]`

Exercise

Create a regex for date-time `yyyy-mm-dd hh:mm:ss.nano_sec`, used in google test : `\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}.\d{3}'\d{3}'\d{3}`

Exercise

Given a `csv` data file

```
0001,  buy, 1000, 00125000
0005,  buy, 2000, 03381000
0522, sell, 4000, 09468000
```

converts into

```
vec.emplace_back("0001", YLib::buy, 1000, 1.25000);
vec.emplace_back("0005", YLib::buy, 2000, 33.81000);
vec.emplace_back("0522", YLib::sell, 4000, 94.68000);
```

In `vim` we have, `/(\d+), {1,2}(\w)(\w+)/"\1"/`

B3. Filesystem

Standard library offers facilities for :

- checking whether a path exists
- checking if a path is a folder or file
- traverse folder tree (recursively or just one layer)

```
std::filesystem::path s0("D:/DEV/cpp");      // 1. path can be constructed with std::string
std::filesystem::path s1 = s0 / "TraderRun"  // 2. path can be concaternated with operator/ (becomes .../cpp/TraderRun)
std::cout << "\nexists()        = " << std::filesystem::exists(s1)
          << "\nroot_path()     = " << s1.root_path()
          << "\nfilename()      = " << s1.filename()
          << "\nextension()     = " << s1.extension();
          << "\nsystem temp dir = " << std::filesystem::temp_directory_path();

//  for(const auto& x : std::filesystem::directory_iterator(s1)) // non-recursive
    for(const auto& x : std::filesystem::recursive_directory_iterator(s1))
    {
        auto str = x.path().filename();
        if      (std::filesystem::is_directory(x.status()))     std::cout << "\nfolder  --- " << str;
        else if (std::filesystem::is_regular_file(x.status()))  std::cout << "\nfile    --- " << str;
    }
```

B4. Chrono library

Lets start with duration. Duration is an integer together with a unit. We can sum them up, or casted into different unit.

```
std::chrono::seconds      s(1);
std::chrono::milliseconds ms(234);
std::chrono::microseconds us(567);
std::chrono::nanoseconds  ns(890);
auto dur = s + ms + us + ns;
auto d0 = std::chrono::duration_cast<std::chrono::milliseconds>(dur); std::cout << d0.count(); // 1234
auto d1 = std::chrono::duration_cast<std::chrono::microseconds>(dur); std::cout << d1.count(); // 1234567
auto d2 = std::chrono::duration_cast<std::chrono:: nanoseconds>(dur); std::cout << d2.count(); // 1234567890
```

Time point can be get from clock, we can convert it into a duration since epoch 1970 Jan 01 using `time_since_epoch()` :

```
std::chrono::time_point<std::chrono::system_clock> tp = std::chrono::system_clock::now(); // 2020-07-01 12:30:01.234567890

auto  s_since_epoch = std::chrono::duration_cast<std::chrono::     seconds>(tp.time_since_epoch());
auto ms_since_epoch = std::chrono::duration_cast<std::chrono::milliseconds>(tp.time_since_epoch());
auto us_since_epoch = std::chrono::duration_cast<std::chrono::microseconds>(tp.time_since_epoch());
auto ns_since_epoch = std::chrono::duration_cast<std::chrono:: nanoseconds>(tp.time_since_epoch());
std::uint64_t ms = ms_since_epoch.count() - s_since_epoch.count() * 1e3; // ms = 234
std::uint64_t us = us_since_epoch.count() - s_since_epoch.count() * 1e6; // us = 234567
std::uint64_t ns = ns_since_epoch.count() - s_since_epoch.count() * 1e9; // ns = 234567890
```

Difference between two time point gives a duration too :

```
std::chrono::time_point<std::chrono::system_clock> t0 = std::chrono::system_clock::now(); // example 12:30:00.000000000
std::chrono::time_point<std::chrono::system_clock> t1 = std::chrono::system_clock::now(); // example 12:31:01.234567890
auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(t1-t0).count(); // ms = 61234
auto us = std::chrono::duration_cast<std::chrono::microseconds>(t1-t0).count(); // us = 61234567
auto ns = std::chrono::duration_cast<std::chrono:: nanoseconds>(t1-t0).count(); // ns = 61234567890
```

The implementation of time point is hidden, its not trivial to print current time point :

```
template<typename CLOCK>
std::string to_string(const std::chrono::time_point<CLOCK>& tp)
{
    auto  s_since_epoch = std::chrono::duration_cast<std::chrono::    seconds>(tp.time_since_epoch()).count();
    auto ns_since_epoch = std::chrono::duration_cast<std::chrono::nanoseconds>(tp.time_since_epoch()).count();
    std::uint64_t ns = ns_since_epoch - s_since_epoch * 1e9;

    std::time_t tt =  std::chrono::system_clock::to_time_t(tp);
    std::tm     tm = *std::localtime(&tt); // conversion to local time

    std::stringstream ss;
    ss << std::put_time(&tm, "%F %T"); // format = 2021-01-12 12:34:56
    ss << "." << std::setfill('0') << std::setw(9) << ns; // fortunately std::setw is not sticky.
    return ss.str();
}

std::cout << to_string(std::chrono::system_clock::now());
```

Construct a timer using chrono library.

```
struct timer
{
    timer()                     {   t0 = std::chrono::high_resolution_clock::now();
                                    t1 = std::chrono::high_resolution_clock::now();             }
    void click()                {   t0 = t1;
                                    t1 = std::chrono::high_resolution_clock::now();             }
    double elapsed() const      {   std::chrono::duration<double, std::milli> duration = t1-t0;
                                    return duration.count() / 1000.0; // output_unit = sec      }

    std::chrono::high_resolution_clock::time_point t0;
    std::chrono::high_resolution_clock::time_point t1;
};
```

B5. timespec

We can also get time using `clock_getime` which returns `time_spec`, it is a `POD` with two integer members `tv_sec` and `tv_nsec` :

```
time_spec ts0, ts1;
clock_gettime(CLOCK_REALTIME,  ts0); // time since 1970 Jan 01   [for timestamp]
clock_gettime(CLOCK_MONOTONIC, ts1); // time since unknown epoch [for latency measurement]

// can be converted into std::chrono::time_point by ...
auto dur = std::chrono::seconds    (ts0.tv_sec) +
           std::chrono::nanoseconds(ts0.tv_nsec);

std::chrono::time_point<std::chrono::system_clock, std::chrono::nanoseconds> tp(dur);
std::cout << to_string(tp); // see previous page for to_string()
```

B6. boost posix time

Each `ptime` object defines a time point, composed of gregorian `date` and `time_duration` which is time elapsed since midnight.

3 basic classes	date +	time_duration =	ptime
difference of 2 objects	`date_duration`	`time_duration`	`time_duration`
direct initialization	`date d(y,m,d);`	`time_duration dur(h,m,s)`	`ptime t(d,dur);`
init from string	`date d = ...from_str`	`time_duration dur = ...from_str`	`ptime t = ...from_str`
init from clock/other	`date d = ...clock`	`auto dur = hours(1) + minutes(2)`	`ptime t = ...clock`

```
// A1. Construct a date
boost::gregorian::date date0(2016,12,25);                                          // with year, month, day
boost::gregorian::date date1 = boost::gregorian::from_string("2017/1/1");          // with string
boost::gregorian::date date2 = boost::gregorian::day_clock::local_day();           // with clock

// A2. Access a date
if (!date.is_not_a_day()) std::cout << date.year() << date.month() << date.day() << date;     // with year, month, day

// B1. Construct a time duration
boost::posix_time::time_duration dur0(1,2,3,123456789); // hour, mintue, second, fractional_second
boost::posix_time::time_duration dur1 =   boost::posix_time::duration_from_string("1:02:03.123456789");
boost::posix_time::time_duration dur2 =   boost::posix_time::hours(1)   +
                                          boost::posix_time::minutes(2) +
                                          boost::posix_time::seconds(3) +
                                          boost::posix_time::milliseconds(123) +
                                          boost::posix_time::microseconds(456);

// B2. Access a time duration
dur0.hours();               // normalised hours, i.e. boost::posix_time::minutes(200).hours() = 3
dur0.minutes();             // normalised mintues
dur0.seconds();             // normalised seconds
dur0.total_seconds();       // total seconds, i.e. total_seconds() = hours()*3600 + minutes()*60 + seconds()
dur0.total_milliseconds();  // total ms, i.e. total_milliseconds() = total_seconds()*1K + #ms_in_fractional_second
dur0.total_microseconds();  // total us, i.e. total_microseconds() = total_seconds()*1M + #us_in_fractional_second

// C1. Construct a ptime
boost::posix_time::ptime ptime0(boost::gregorian::date(2016,12,25), boost::posix_time::time_duration(1,2,3,123456));
boost::posix_time::ptime ptime1 = boost::posix_time::time_from_string("2016-12-25 01:02:03.123456");
boost::posix_time::ptime ptime2 = boost::posix_time::microsec_clock::local_time();

// C2. Access a ptime
boost::gregorian::date d = ptime.date();
boost::posix_time::time_duration dur = ptime.time_of_day();
```

C1. Boost ASIO timer

- In linux, we use this function for getting time : `clock_gettime`.
- In linux, we do not use boost asio for low latency, it is the highest level API for socket programming :
- highest level `boost::asio`
- middle level `BSD socket`
- lowest level `epoll, kqueue, libevent`

1. IO-task manager and IO-object

IO-task manager `io_service` stores a queue of asynchronized tasks, waiting to be run by calling `io_service::run()`

- synchronous function is blocking, it blocks current thread until the function is done and returns
- asynchronous function is non-blocking, it delegates the job to others and returns immediately, invokes callback when done
- for TCP read, `async_read_some` is prefered : we don't know when it happens and immediate callback on any incoming bytes
- for TCP write, `sync_write` is prefered : we want to send every bytes before moving on to next step

IO object and async function		constructor argument	invocation argument
`deadline_timer`	class	`io_service` + time period	
`tcp::acceptor`	class	`io_service` + port number	
`tcp::socket`	class	`io_service`	
`deadline_timer::async_wait`	function		callback
`tcp::acceptor::async_accept`	function		callback + `tcp::socket`
`tcp::socket::async_read_some`	function		callback + `buffer`
callback of `async_wait`	functor	`deadline_timer`	error placeholder
callback of `async_accept`	functor	`tcp::acceptor` + `tcp::socket`	error placeholder
callback of `async_read_some`	functor	`tcp::socket` + `buffer`	error placeholder + num bytes

2. Waiting and repeated waiting

Lets make a timer …

```
boost::asio::io_service io;                                                  // step 1. declare io-service
boost::asio::deadline_timer timer(io, boost::posix_time::seconds(1));        // step 2. declare io-object
auto cb = [](const auto& err){ std::cout << "done"; };                       // step 3. declare callback
timer.async_wait(cb);                                                        // step 4. invoke io-object async
io.run();                                                                    // step 5. invoke io-service run
```

Lets make timer count repeatedly. The main thread pops a task (async wait) from `io_service`, executes it and callbacks, the callback registers new task (which is also async wait) to `io_service` and quits, thus it keeps popping one task and pushing one new task into `io_service`, this procedure is repeated for a predetermined number of times. Callback functor should have :

- access to the timer so that it can register new waiting task
- access to the counter so that it can stop after certain number of invocation

```
struct callback
{
    callback(boost::asio::deadline_timer &t, int c) : timer(t), count(c) {}

    void fct(const boost::system::error_code& err)
    {
        if (count==0) return; // end the recursion
        --count;
        timer.expires_at(timer.expires_at() + boost::posix_time::seconds(1)); // Dont forget to reinitialize deadline.
        timer.async_wait(boost::bind(&callback::fct, this, boost::asio::placeholders::error));
    }

    boost::asio::deadline_timer &timer;
    int count;
}

// *** The 5 steps *** //
boost::asio::io_service io;                                                      // step 1. declare io-service
boost::asio::deadline_timer timer(io, boost::posix_time::seconds(1));            // step 2. declare io-object
callback cb(timer, 100);                                                         // step 3. declare callback
timer.async_wait(boost::bind(&callback::fct, boost::ref(cb), placeholders::error));  // step 4. invoke io-object async
io.run();                                                                        // step 5. invoke io-service run
```

3. Multithread timer without common resource

Now we instantiate 3 timers, 3 callbacks and run in 3 threads (each run equal number of times) :

```
1   boost::asio::io_service io;
2   boost::asio::deadline_timer timer0(io, boost::posix_time::seconds(1));
    boost::asio::deadline_timer timer1(io, boost::posix_time::seconds(1));
    boost::asio::deadline_timer timer2(io, boost::posix_time::seconds(1));
3   callback cb0(timer, 100);
    callback cb1(timer, 100);
    callback cb2(timer, 100);
4   timer0.async_wait(boost::bind(&callback::fct, boost::ref(cb0), boost::asio::placeholders::error));
    timer1.async_wait(boost::bind(&callback::fct, boost::ref(cb1), boost::asio::placeholders::error));
    timer2.async_wait(boost::bind(&callback::fct, boost::ref(cb2), boost::asio::placeholders::error));
5   boost::thread thread0(boost::bind(&boost::asio::io_service::run, boost::ref(io)));
    boost::thread thread1(boost::bind(&boost::asio::io_service::run, boost::ref(io)));
    boost::thread thread2(boost::bind(&boost::asio::io_service::run, boost::ref(io)));
    thread0.join();
    thread1.join();
    thread2.join();
```

4. Multithread timer with common resource

Consider the following `callback`, `count` is a shared resource, racing condition exists as multithreads access `count` concurrently.

```
struct callback
{
    callback(boost::asio::deadline_timer &t, int& c) : timer(t), count(c) {}

    void fct(const boost::system::error_code& err)
    {
        if (count==0) return;
        --count;
        timer.expires_at(timer.expires_at() + boost::posix_time::seconds(1));
        timer.async_wait(boost::bind(&callback::fct, this, boost::asio::placeholders::error));
    }

    boost::asio::deadline_timer &timer;
    int& count;
}
```

error-placeholder and error itself are two different things.

Suppose two threads entering `callback::fct` when `count` equals to *1*, both survive `if(count==0)` checking, eventually `count` becomes *-1* resulting in an infinity loop. This can be solved by dispatching all `callback` with `boost::asio::strand`. Callbacks dispatched from `strand` are executed concurrently.

```
1   boost::asio::io_service io;
    boost::asio::strand strand(io);
2   boost::asio::deadline_timer timer0(io, boost::posix_time::seconds(1));
    boost::asio::deadline_timer timer1(io, boost::posix_time::seconds(1));
    boost::asio::deadline_timer timer2(io, boost::posix_time::seconds(1));
3   int count = 1000;
    callback cb0(strand, timer, count); // strand for protecting count
    callback cb1(strand, timer, count);
    callback cb2(strand, timer, count);
4   timer0.async_wait(strand.wrap(boost::bind(&callback::fct, boost::ref(cb0), boost::asio::placeholders::error)));
    timer1.async_wait(strand.wrap(boost::bind(&callback::fct, boost::ref(cb1), boost::asio::placeholders::error)));
    timer2.async_wait(strand.wrap(boost::bind(&callback::fct, boost::ref(cb2), boost::asio::placeholders::error)));
5   boost::thread thread0(boost::bind(&boost::asio::io_service::run, boost::ref(io)));
    boost::thread thread1(boost::bind(&boost::asio::io_service::run, boost::ref(io)));
    boost::thread thread2(boost::bind(&boost::asio::io_service::run, boost::ref(io)));
    thread0.join();
    thread1.join();
    thread2.join();
```

We need to modify callback to support strand object :

```
struct callback
{
    callback(boost::asio::strand &s, boost::asio::deadline_timer &t, int c) : strand(s), timer(t), count(c) {}

    void fct(const boost::system::error_code& err)
    {
        if (count==0) return;
        --count;
        timer.expires_at(timer.expires_at() + boost::posix_time::seconds(1));
        timer.async_wait(strand.wrap(boost::bind(&callback::fct, this, boost::asio::placeholders::error)));
    }

    boost::asio::strand& strand;
    boost::asio::deadline_timer &timer;
    int count;
}
```

C2. TCP server / session / client

Let's create TCP server class. It is responsible for making connection through async `accept()`, which invokes callback on completion. The callback constructs a new TCP session and invokes async `accept()` again, thus forming a recursion. The server should maintain a set of sessions connecting to its clients, while each session maintains one socket, through which read and write are done.

Five steps similar to timer

```
    using namespace boost::asio::ip;
    using namespace boost::asio;
    struct tcp_server
    {
1/2     tcp_server(boost::asio::io_service& io, int port) : io_service(io), acceptor(io, tcp::endpoint(tcp::v4(),port))
        {
        }

4       void async_run()
        {
            socket = boost::make_shared<tcp::socket>(io_service);
            acceptor.async_accept(*socket, boost::bind(&tcp_server::callback, this, placeholders::error));
        }

3       void callback(const boost::system::error_code& error)
        {                                                                   // Five steps for server
            socket->set_option(tcp::no_delay(true));                        // 1. no delay
            auto session = boost::make_shared<tcp_session>(socket);         // 2. session declared
            sessions.insert(session);                                       // 3. session inserted
            session->async_run();                                           // 4. run session
            async_run();                                                    // 5. run acceptor
        }
                                                                            // Four main classes' adornament
        boost::asio::io_service& io_service;                                // 1. reference
        boost::asio::acceptor acceptor;                                     // 2. instance
        boost::shared_ptr<tcp::socket> socket;                              // 3. shared_ptr
        std::set<boost::shared_ptr<tcp_session>> sessions;                  // 4. set of shared_ptr
    };

    struct tcp_session
    {
        tcp_session(boost::shared_ptr<tcp::socket> socket) : socket(socket)
        {
        }

        void async_run()
        {
            socket->async_read_some
            (
                boost::asio::buffer(str),
                boost::bind(&tcp_session::callback, this, placeholders::error, placeholders::byte_transferred));
            }
        }

        void callback(const boost::system::error_code& error, size_t byte)
        {
            std::cout << "\nreceive " << str;
            async_run();
        }

        boost::shared_ptr<tcp::socket> socket;
        std::string str;
    };
```

Unlike TCP server which manages a set of connections, TCP client manages its own `tcp::socket` to server only, hence it is simpler.

```
    struct tcp_client
    {
        tcp_client(boost::asio::io_service& io) : io_service(io), socket(io)
        {
        }

        void connect(const std::string& ip, int port)
        {                                                                   // Five steps for client
            tcp::resolver resolver(io_service);                             // 1. declare resolver
            tcp::resolver::query query(ip, boost::lexical_cast<std::string>(port)); // 2. declare query
            tcp::endpoint endpoint = *resolver.resolve(query);              // 3. declare endpoint
            socket.connect(endpoint);                                       // 4. Socket connect
            socket.set_option(tcp::no_delay(true));                         // 5. Socket no delay
        }

        // Offers user an access to socket so that he can do sync/async read/write.

        boost::asio::io_service& io_service;
        tcp::socket socket;
    };
```

D1. Ownership of smart pointers

1 Without proper ownership management, the use of dynamically allocated resource will end up with either :

- forget to release resource, resulting in leakage
- repeated release resource, resulting in undefined behaviour
- raw pointer is exception-unsafe, see this example …

```
void not_exception_safe()
{
    T* p = new T;
    throw my_exception();
    delete p;
}

void exception_safe()
{
    smart_ptr<T> p(new T);
    throw my_exception();
}
```

2 Smart pointers manage object dynamically allocated with `new` operator, they are used like raw pointers :

- dereferenced by `operator*()` and
- members access by `operator->()`.

3 Smart pointers are classified according to the way they manage resource ownership, ownership means the responsibility to release the resources at appropriate time. There are 4 levels of ownerships ordered in increasing complexity and computational load, so we should use the simplest one that caters for our needs. By *principle of least privilege* implement local ownership if transferability is not necessary, implement unique ownership if shared ownership is not necessary.

	movability / copyablility of ownership	example
no ownership		`std::weak_ptr<T>`
local ownership	non-transferable ownership	`std::scoped_ptr<T>`
unique ownership	movable ownership	`std::unique_ptr<T>`
shared ownership	copyable ownership	`std::shared_ptr<T>`

4 Weak pointer is observer of resource and hence is not responsible for resource release. Scoped pointer binds resource allocation to pointer construction and binds resource release to pointer destruction, that is binding resource ownership to pointer lifetime, this is RAII. Unique pointer allows only one pointer owning the resource at the same time, the ownership is movable, resource is released when unique pointer owning resource goes out of scope. Shared pointer permits multiple pointers owning the resource, ownership is copyable, resource is released when the last shared pointer owning resource goes out of scope, it is done by reference counting. It is the ownership itself that is movable or copyable, it is ***NOT*** about the movability or copyability of the resource.

What is Resource Acquisition Is Initialization (RAII)?

It is an idiom of resource management :

- RAII ties resource to an object. Resource is allocated in object construction and is deallocated in object destruction.
- RAII guarantees that resource is available as long as the object is alive.
- RAII is exception safe, as destructor must be called during stack unwinding.

Examples : `std::scope_ptr` and `std::lock_guard`

D2. Implementation of shared pointer (Type erasure)

Reference count is implemented by reference count manager which is also dynamically allocated, it has individual reference counts for shared pointer and weak pointers respectively. Dynamic allocation of the resource is done explicitly using `new` operator which is then housekept by a shared pointer, this very-first shared-pointer is responsible for creation of a manager, ownership can be shared among multiple shared pointers by sharing the same manager (no re-creation of manager involved). Resource will then be released when reference count of shared pointer drops to zero, regardless of reference count of weak pointer.

- construct one manager for managing one dynamically allocated resource
- construct multi managers for managing one resource will result in **duplicated deletion**

```
// This implementation is rejected by Wintermute !!! Please provide deleter, using type-erasure pattern
template<typename T> struct manager
{
    manager(int s, int w, T* p) : count_s(s), count_w(w), ptr(p) {}
    int count_s = 0;
    int count_w = 0;
    T* ptr = nullptr;
};

template<typename T> struct shared_ptr
{
    explicit shared_ptr(T* ptr = nullptr)
    {
        manager_ptr = new manager<T>(1,0,ptr);
    }

    ~shared_ptr()
    {
        decrement();
    }

    shared_ptr(shared_ptr<T>& rhs)
    {
        manager_ptr = rhs.manager_ptr;
        increment();
    }

    shared_ptr<T>& operator=(shared_ptr<T>& rhs)
    {
        decrement();
        manager_ptr = rhs.manager_ptr;
        increment();
        return *this;
    }

    shared_ptr(shared_ptr<T>&& rhs)
    {
        manager_ptr = rhs.manager_ptr;
        rhs.manager_ptr = nullptr;
    }

    shared_ptr<T>& operator=(shared_ptr<T>&& rhs)
    {
        decrement();
        manager_ptr = rhs.manager_ptr;
        rhs.manager_ptr = nullptr;
        return *this;
    }

          operator bool() const { return  (manager_ptr && manager_ptr->ptr); }
    const T& operator*()  const    { return *(manager_ptr->ptr); }
          T& operator*()           { return *(manager_ptr->ptr); }
    const T* operator->() const    { return   manager_ptr->ptr;  }
          T* operator->()          { return   manager_ptr->ptr;  }

    // private implementation
    void increment()
    {
        if (manager_ptr!=nullptr) ++(manager_ptr->count_s);
    }

    void decrement()
    {
        if (manager_ptr!=nullptr && manager_ptr->count_s>0)
        {
            --(manager_ptr->count_s);
            if (manager_ptr->count_s==0)
            {
                delete manager_ptr->ptr;
                delete manager_ptr; manager_ptr = nullptr;
            }
        }
    }

    manager<T>* manager_ptr = nullptr;
};
```

This implementation is tested in :

- Maven second round tech interview 2022
- Wintermute second round tech interview 2022

3 problems are found :

- we can move `T*ptr` to under `shared_ptr` directly so that we don't need double referencing
- we can skip allocating manager when `shared_ptr` is default-constructed
- please provide `deleter` which is implemented by type erasure pattern

What is type erasure pattern?

- please read `interview` code

1. create `deleter_base` analog to `object_base`
2. create `deleter_wrapper` analog to `object_wrapper`
3. instantate `deleter_base*` in `manager` constructor

D3. Instantiation and usage

Instantiation of shared pointer

- direct initialization, which invokes `new` operator twice, once for resource itself and once for manager
- copy initialization, which invokes `new` operator twice, once for resource itself and once for manager
- using factory, which only invokes `new` operator once, with size equals to sum of resource and manager
- do not assign local stack object to shared pointer
- do not assign single heap object to multiple managers, as it results in multiple deletions of the same object

```
// 3 instantiations of shared pointer
std::shared_ptr<T> sp0(new T(arg));                      // direct initialization
std::shared_ptr<T> sp1 = std::shared_ptr<T>(new T(arg));  //   copy initialization
std::shared_ptr<T> sp2 = std::make_shared<T>(arg);        // factory

// 2 incorrect instantiations of shared pointer
std::shared_ptr<T> sp0(&obj);

T* p = new T(arg);
std::shared_ptr<T> sp1(p);
std::shared_ptr<T> sp2(p);
std::shared_ptr<T> sp3(p);
```

Usage of shared pointer

- shared pointer can be tested (returns true if resource exists) as it offers conversion operator to `bool`
- shared pointer can be compared (returns true if two shared pointers point to the same resource)
- shared pointer offers member `get()` to return raw pointer
- shared pointer offers member `reset()`
- shared pointer allows downcast inheritance, which can be implemented with static-cast or dynamic-cast

```
std::shared_ptr<T> sp0(new T(arg0));                     if (!sp0)      std::cout << "empty";
std::shared_ptr<T> sp1(new T(arg1));                     if ( sp0!=sp1) std::cout << "not same";
const T* raw_ptr = sp0.get();
sp0.reset();
sp0.reset(new T(arg2));

std::shared_ptr<BASE> bptr(new DERIVE);
std::shared_ptr<DERIVE> dptr0 = std:: static_pointer_cast<DERIVE>(bptr); // crash if invalid, faster
std::shared_ptr<DERIVE> dptr1 = std::dynamic_pointer_cast<DERIVE>(bptr); // throw if invalid, slower
```

Here is the implementation of the casting functions :

```
template<class T, class U> std::shared_ptr<T> const_pointer_cast(const std::shared_ptr<U>& sp) noexcept
{
    auto p = const_cast<typename std::shared_ptr<T>::element_type*>(sp.get());
    return std::shared_ptr<T>(sp, p);
}

template<class T, class U> std::shared_ptr<T> static_pointer_cast(const std::shared_ptr<U>& sp) noexcept
{
    auto p = static_cast<typename std::shared_ptr<T>::element_type*>(sp.get());
    return std::shared_ptr<T>(sp,p);
}

template<class T, class U> std::shared_ptr<T> dynamic_pointer_cast(const std::shared_ptr<U>& sp) noexcept
{
    if (auto p = dynamic_cast<typename std::shared_ptr<T>::element_type*>(sp.get()))
         return std::shared_ptr<T>(sp,p);
    else return std::shared_ptr<T>();
}

template<class T, class U> std::shared_ptr<T> reinterpret_pointer_cast(const std::shared_ptr<U>& sp) noexcept
{
    auto p = reinterpret_cast<typename std::shared_ptr<T>::element_type*>(sp.get());
    return std::shared_ptr<T>(sp,p);
}
```

Instantiation of weak pointer

- direct initialization
- copy initialization

```
// 2 instantiations of weak pointer
std::shared_ptr<T> sp(new T);
std::weak_ptr<T> wp0(sp);                                // direct initialization
std::weak_ptr<T> wp1=sp;                                 //   copy initialization
```

Usage of weak pointer

Functionality of weak pointer is quite limited, it does *NOT* keep resource alive.

- weak pointer offers member `expired()` to query existence of resource
- weak pointer offers member `lock()` to construct shared pointer, incrementing reference count, safeguarding resource

```
if (!wp.expired())
{
     std::shared_ptr<T> sp = wp.lock();
     sp->fct();
}
```

Can we declare shared pointer pointing to stack memory object?

Yes, but you should ensure (1) do not access that shared pointer when resource is popped from stack and (2) offer a good deleter.

```
// Suppose someone offer such function, I want to call it with my stack object ...
void fct(std::shared_ptr<input>& x);

// I can do it this way ... please read latter section for syntax of customized deleter
input my_stack_object;
auto sp = std::shared_ptr<input>(my_stack_object, [](input*){ /* no delete */ });
fct(sp);
```

D4. Problem – cycle

Reference count works well when multiple shared pointers pointing the same resource.

```
struct T
{
    T() { std::cout << "\nconstruct";     }
   ~T() { std::cout << "\ndestruct";      }
};

void test_no_leakage()
{
     std::shared_ptr<T> p0(new T);                                    // print construct (allocate one resource)
     {
          std::shared_ptr<T> p1 = p0;
          {
               std::shared_ptr<T> p2 = p1;
               p0 = p2;                                               // oops, forming a cycle
               std::cout << "\np0 count = " << p0.use_count();        // 3
               std::cout << "\np1 count = " << p1.use_count();        // 3
               std::cout << "\np2 count = " << p2.use_count();        // 3
          }
          std::cout << "\np0 count = " << p0.use_count();             // 2
          std::cout << "\np1 count = " << p1.use_count();             // 2
     }
     std::cout << "\np0 count = " << p0.use_count();                  // 1
}                                                                     // print destruct (yeah, it is safe)
```

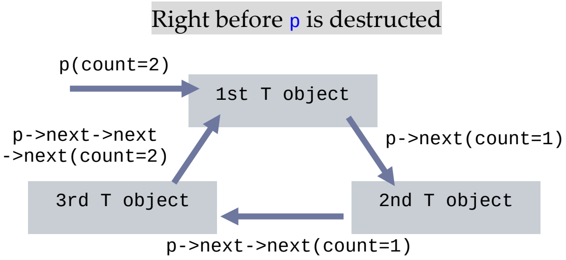
However, problem happens when there are multiple resources, each contains, by composition, a shared pointer pointing to another resource, thus forming a cycle which is different from the one above. Reference counts are still updated correctly, yet when the only local variable `p` goes out of it scope no resource is released as reference count is non zero, while leaving the cycle detached from any local variable, every shared pointer in the cycle keeps each other alive forever, corresponding resources will never be released even when no one else points to them from outside the cycle, in other words, leakage of 3 instances happens.

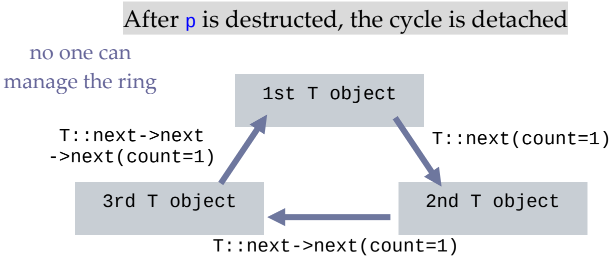
```
struct T // Let’s build a list with shared pointer instead of raw pointer.
{
    T() { std::cout << "\nconstruct";     }
   ~T() { std::cout << "\ndestruct";      }
    std::shared_ptr<T> next;
    std::weak_ptr<T> next_weak;
};

void test_leakage()
{
    std::shared_ptr<T> p(new T);                                      // print construct
    p->next = std::shared_ptr<T>(new T);                              // print construct
    p->next->next = std::shared_ptr<T>(new T);                        // print construct

     std::cout << "\np count = " << p.use_count();                    // 1
     std::cout << "\np+ count = " << p->next.use_count();             // 1
     std::cout << "\np++ count = " << p->next->next.use_count();      // 1
     p->next->next->next = p;
     std::cout << "\np count = " << p.use_count();                    // 2, which is correct
     std::cout << "\np+ count = " << p->next.use_count();             // 1
     std::cout << "\np++ count = " << p->next->next.use_count();      // 1
     std::cout << "\np+++ count = " << p->next->next->next.use_count(); // 2 , which is correct
}                                                                     // no destructor called
```

The diagram illustrates that reference count works fine. Rectangle denotes resource, while arrow denotes shared pointer. When `p` is destructed, the reference count for the first dynamically allocated object is reduced to one thus its resource is not released, after that all reference counts in the ring equal to one, keeping resource of each other alive forever.





Cycle is common in list and tree. The problem is solved by replacing the red line with weak pointer, which breaks the cycle :

```
p->next->next->next_weak = p;
```

D5. Problem – Share from this

If an object is dynamically allocated and kept by shared pointer, and if one of its member function wants to pass the shared pointer to other global functions, can we do the following?

```
void T::fct()
{
    std::shared_ptr<T> sp0(this);
    global_fct(sp0);
}

void test()
{
    std::shared_ptr<T> sp(new T);
    sp->fct();          // 1st deletion inside T::fct(), when sp0 goes out of scope (ref count falls to zero)
}                       // 2nd deletion inside test(),   when sp  goes out of scope
```

No it does not work, because two managers (`sp` and `sp0`) are managing the same resource, which leads to double deletion.

<u>*Solution*</u>

This problem can solved by weak pointer as shown in *RHS* above.

```
void T::fct()
{
    std::weak_ptr<T> wp(this);
    global_fct(wp.lock());
}
```

Luckily we do not need to implement weak-pointer-solution ourselves, as we can use the template class `std::enable_shared_from_this`, which is composed of a weak pointer and function `shared_from_this()` the latter constructs one shared pointer from the weak pointer. Declare `T` using curiously recurring template pattern *(CRTP)* :

```
template<typename T> struct enable_shared_from_this
{
    enable_shared_from_this() : wp(this) {                  }
    shared_ptr<T> shared_from_this()     { return wp.lock(); }
    std::weak_ptr<T> wp;
};

class T : public std::enable_shared_from_this<T>
{
    void fct()
    {
        shared_ptr<T> sp0 = shared_from_this();        // Increment reference count
        fct(sp0);
    }                                                  // Destruction of sp0 decreases reference count.
};

void test()
{
    shared_ptr<T> sp(new T);
    sp->fct();
//  T obj; obj.fct();                                  // Don't do that, it crashes.
}                                                      // Destruction of sp decreases reference count, hence releases T.
```

D6. Object pool and custom deleter

Suppose we have a class owning a vector of `resource`, we should pick one out of the following implementation :

```
std::vector <resource> MY_CLASS::impl; // No, involve slow copying.
std::vector<*resource> MY_CLASS::impl; // No, we don't know when will owners of resources delete them, they may not be alive.
std::vector<std::unique_ptr<resource>> MY_CLASS::impl;
```

The final method has to implement `void MY_CLASS::add(std::unique_ptr<resource>& sp) { impl.push_back(sp); }` and users have to :

```
std::vector<std::unique_ptr<resource>> impl;
impl.push_back(std::make_unique<resource>(a,b,c);

// How about stack objects? Is it possible to create unique ptr to stack obj? A deleter that does nothing?
resource obj(a,b,c);
impl.push_back(std::unique_ptr<resource, [](resource*){ /* do_nothing */ }>(obj)); // please check if this is correct
```

Can we declare smart pointer pointing to stack object? Yes, if we can provide our own deleter. Lets try for unique pointer.

```
struct resource
{
    resource(std::uint32_t a_=1, std::uint32_t b_=2, std::uint32_t c_=3) : a(a_),b(b_),c(c_) {}
    std::uint32_t a;
    std::uint32_t b;
    std::uint32_t c;
};

template<std::uint32_t N> class resource_pool // Todo : make it a template, so that resource can be any type
{
public:
    resource_pool() { for(std::uint32_t n=0; n!=N; ++n) flags[n] = true; }

    struct resource_deleter
    {
        resource_deleter(resource_pool<N>& pool_, std::uint32_t index_) : pool(pool_), index(index_) {}
        void operator()(resource* ptr) { pool.flags[index] = true; }
        resource_pool<N>& pool;
        std::uint32_t index;
    };

public:
    static const std::uint32_t size = N;
    friend class resource_deleter;

    // Normally we cannot assign smart pointer to stack object, unless we provide appropriate deleter in declaring output.
    template<typename... ARGS> auto make_unique(ARGS&&... args)
    {
        for(std::uint32_t n=0; n!=N; ++n)              Please note that two template parameters : T & T_deleter
        {                                              together defines one std::unique_ptr<T, T_deleter>.
            if (flags[n])                              typeof(std::unique_ptr<T,T_del0>) != typeof(std::unique_ptr<T, T_del1>)
            {
                flags[n] = false;

                resource_deleter deleter(*this, n);                                // we cannot call std::make_unique()
                std::unique_ptr<resource, resource_deleter> output(nullptr, deleter);  // when we are using custom deleter.
                output.reset( new (&impl[n]) resource{std::forward<ARGS>(args)...} );
                return output;
            }
        }
        resource_deleter deleter(*this, 0);
        std::unique_ptr<resource, resource_deleter> output(nullptr, deleter);
        return output;
    }

private:
    resource impl[N]; // TOdo : change "resource array" to "char array"
    bool flags[N];    // true for available
};

// Testing
resource_pool<5> pool;
auto x0 = pool.make_unique(11,12,13);
auto x1 = pool.make_unique(21,22,23);     // 2 resources are used
{
    auto x2 = pool.make_unique(31,32,33);
    auto x3 = pool.make_unique(41,42,43); // 4 resources are used
}
auto x4(std::move(x1));                   // 2 resources are used
auto x5 = pool.make_unique(51,52,53);
auto x6 = pool.make_unique(61,62,63);
auto x7 = pool.make_unique(71,72,73);     // 5 resources are used
auto x8 = pool.make_unique(81,82,83);     // no more resource in pool, return nullptr
```

- There is no deleter for `std::shared_ptr<T>`, as it is managed by a control-block inside `std::shared_ptr<T>`,.
- There are ***two*** template parameters for `std::unique_ptr<T,DELETER>`, they together define one unique pointer type.

D7. Allocator

Like the custom deleter in smart pointer, we can have custom allocator for STL container.

- deleter in smart pointer defines a functor for deletion
- allocator in container defines one function for allocate and one for deallocate

```
void deleter::operator()(T* ptr);
T*   allocator::  allocate(size_t num_objects, const void* hint = 0);
void allocator::deallocate(void* ptr, size_t num_objects);
```

Here is an example tested in Ubuntu.

```
template<typename T> class my_allocator
{
public: // STL requires the following typedef
    typedef size_t size_type;        typedef ptrdiff_t difference_type;
    typedef T* pointer;              typedef const T*  const_pointer;
    typedef T& reference;            typedef const T&  const_reference;
    typedef T  value_type;

public:
    T* allocate(size_t n, const void* hints=0) // n = num of T-instances (not bytes)
    {
        T* p = new T[n];
        std::cout << "\nallocate " << std::dec << n << " objects at " << std::hex << (std::uint64_t)p;
        return p;
    }
    void deallocate(void* p, size_t n) // n = num of T-instances (not bytes)
    {
        delete[] reinterpret_cast<T*>(p);
        std::cout << "\ndeallocate " << std::dec << n << " objects at " << std::hex << (std::uint64_t)p;
    }

private:
    // Typical implementation is ...
//  std::array<  T,1024> cells;
//  std::array<int,1024> queue_of_unused_cells; // index to unused cells
};

std::vector<int, my_allocator<int>> vec;       // For vector, it prints 1,2,4,8,16,32,64,128
std::list  <int, my_allocator<int>> list;      // For list,   it prints 1,1,1,1,1,1, ...
for(int n=0; n!=100; ++n) vec.push_back(n);    for(const auto& x:vec)  std::cout << std::dec << x << " ";
for(int n=0; n!=10; ++n) list.push_back(n);    for(const auto& x:list) std::cout << std::dec << x << " ";
```

Pointing to stack memory

```
A a(3,4,5);
auto sp0 = std::make_shared<A>(1,2,3);          sp0->fct();
std::shared_ptr<A> sp1(new A(2,3,4));           sp1->fct();
std::shared_ptr<A> sp2(&a, [](A*){});           sp2->fct();
```

D8. Extension to array

Smart pointer is a safe way to manage raw pointer. Can it be extended to raw array? The answer is Yes. We can consider :

- `std::unique_ptr<T>` as a smart manager for raw pointer `T* ptr = new T{a,b,c};`
- `std::unique_ptr<T[]>` as a smart manager for raw array `T* ptr = new T[4]{{a0,b0,c0},{a1,b1,c1},{a2,b2,c2},{a3,b3,c3}};`
- there is `operator*` for `std::unique_ptr<T>` but not for `std::unique_ptr<T[]>`
- there is `operator[]` for `std::unique_ptr<T[]>` but not for `std::unique_ptr<T>`
- both of them are `T*` under the hood, thus we have the following type checking results ...
- an example of using `std::unique_ptr<T[]>` can be found in `YLib` *custom string* as a work-around to solve bug in *reckless-logger*
- in short, `std::unique_ptr<T[]>` is ***NOT*** unique pointer of `T[]`, instead it is unique array of `T`

```
struct T { std::uint32_t x; std::uint32_t y; std::uint32_t z; };
std::cout << std::is_same<T*,  T[]>::value;                                           // false
std::cout << std::is_same<T*,  decltype(std::declval<std::unique_ptr<T  >>().get())>::value;  //  true
std::cout << std::is_same<T*,  decltype(std::declval<std::unique_ptr<T[]>>().get())>::value;  //  true
std::cout << std::is_same<T**, decltype(std::declval<std::unique_ptr<T[]>>().get())>::value;  // false
std::cout << std::is_same<T[], decltype(std::declval<std::unique_ptr<T[]>>().get())>::value;  // false
std::cout << std::is_same<T&,  decltype(*(std::declval<std::unique_ptr<T  >>()   ))>::value;  //  true
std::cout << std::is_same<T&,  decltype( (std::declval<std::unique_ptr<T[]>>()[0]))>::value;  //  true

std::unique_ptr<T>   t(new T(11,12,13));
std::unique_ptr<T[]> ts(new T[4]{{21,22,23}, {31,32,33}, {41,42,43}, {51,52,53}});

std::cout << *t;
for(int n=0;n!=4;++n) std::cout << ts[n];
```

E1. Function and member function

Utility for function call :

- `std::function`
- `std::mem_fn`
- `std::invoke`
- `std::apply`

In C++11, `std::function` offers a standardised wrapper for :

- global function
- member function
- functor
- lambda

so that they can be consumed which can be consumed through :

- invocation inplace
- invocation in a for-loop, such as a container of tasks `std::function`
- invocation in algorithms, such as `std::for_each` and `std::transform`
- invocation in threads, such as `std::thread` and `std::async`

In C++03, there is no standard way to do so. Basically we need 4 steps :

```
// [Step 1] Define function
X fct(const Y&, const Z&);
struct T
{
    X mem(const Y&, const Z&);
    X operator()(const Y&, const Z&);
};

// [Step 2] Declare function pointer
T object; Y y; Z z;
X (   *fp0)(const Y&, const Z&);
X (T::*fp1)(const Y&, const Z&);
```

```
// [Step 3] Initialize function pointer

fp0 = &fct;    // & is necessary
fp1 = &T::mem; // & is necessary
```

```
// [Step 4] Invoke function pointer
X x0 =          fp0 (y,z);
X x1 = (object.*fp1)(y,z);
X x2 =  object      (y,z);
X x3 = [](const Y&, const Z&){ ... return X{}; }(y,z);
```

In C++11, we have a standardized approach :

```
// [Step 2,3] Declare and initialize std::function
std::function<X(const Y&, const Z&) f0 = &fct;
std::function<X(const Y&, const Z&) f1 = std::bind(&T::mem, std::ref(object), _1, _2);
std::function<X(const Y&, const Z&) f2 = std::bind(         std::ref(object), _1, _2);
std::function<X(const Y&, const Z&) f3 = [](const Y&, const Z&){ ... return X{}; };

// [Step 4] Invoke std::function

X x0 = fp0(y,z);
X x1 = fp1(y,z);
X x2 = fp2(y,z);
X x3 = fp3(y,z);
```

```
std::vector<std::function<X(const Y&, const Z&)>> fps;
fps.push_back(&fct);
fps.push_back(std::bind(&T::mem, std::ref(object), _1, _2));
fps.push_back(object);
fps.push_back([](const Y&, const Z&){ ... return X{}; });
for(auto& f:fps) f(y,z);
```

Furthermore, we have `std::mem_fn`

```
auto mf = std::mem_fn(&T::mem);
mf(object,y,z);
```

Furthermore, we have `std::invoke` and `std::apply`, they are slightly different (please verify in `gcc`) :

```
std::invoke(fct,y,z);
std::invoke(f0, y,z);
std::invoke(f1, y,z);
std::invoke(f2, y,z);
std::invoke(f3, y,z);
std::apply(fct,std::make_tuple(y,z));
std::apply(f0, std::make_tuple(y,z));
std::apply(f1, std::make_tuple(y,z));
std::apply(f2, std::make_tuple(y,z));
std::apply(f3, std::make_tuple(y,z));
```

E2. Binding and reference wrapper

Template function `std::bind` is a tools that aids the construction of `std::function`. It offers features :

- reduce arity by binding to *constants or variables*
- reorder remaining arguments by `std::placeholders::_1`
- return anonymous `std::function` objects

Binding can be done in 5 approaches :

```
object.name = "A";
auto f0 = std::bind(&T::mem,           object,  _1,_2);      // 1. pass object by value
auto f1 = std::bind(&T::mem,          &object,  _1,_2);      // 2. pass object by pointer
auto f2 = std::bind(&T::mem, std:: ref(object), _1,_2);      // 3. pass object by reference
auto f3 = std::bind(&T::mem, std::cref(object), _1,_2);      // 4. pass object by const reference
auto f4 = std::bind(&T::mem, std::move(object), _1,_2);      // 5. pass object by rvalue reference
```

Fail to bind `x` to `y` means, given a function signature `fct(x)`, when caller invokes `fct(y)`, compiler cannot resolve the function, where `std::ref` is a factory of `std::reference_wrapper`. The underlying value of reference wrapper can be accessed through `get()`.

```
template<typename T>
class reference_wrapper
{
    template<typename U, typename ...>
    // SFINAE to ensure : (1) std::remove_cvref_t<U> == T
                          (2) std::remove_cvref_t<U> != std::reference_wrapper<T>
    constexpr explicit reference_wrapper(U&& u) : std::addressof(std::forward<U>(u))
    {
    }

    Constexpr T& get() const noexcept { return *_ptr; }
    T* _ptr;
};

void fct(std::reference_wrapper<gaussian>& ref_gauss)
{
    ref_gauss.get()...
}
```

E3. Lambda

Lambda defines anonymous function with the following syntax :

```
[]           (input_argument_list)              { function_definition }
[capture_list](input_argument_list)             { function_definition }
[capture_list](input_argument_list)-> return_type { function_definition }
```

1. Input argument list

Input argument list is like `std::placeholders` in `std::bind`, it is declared like ordinary function.

```
// case 1 : invocation inplace
std::cout << [](int n, int m) ->std::string { ... return "hello"; }(1,2);

// case 2/3 : invocation by for-each algorithm
std::vector<T> vec;
std::for_each(vec.begin(), vec.end(), [](const T&    t) { std::cout << t; });
std::for_each(vec.begin(), vec.end(), [](const auto& t) { std::cout << t; });

// case 4 : invocation by thread
std::thread t([](){ auto npv = price(deal); return npv; });
```

2. Return type

Some examples above do return, yet we dont need to specify the return type provided that it can be deduced from the single return statement. However, when there are multiple return statements, or when the return type cannot be deduced, we need to specify the return type through trailing return type syntax *TRTS*.

```
std::cout << [](int x, int y)->double // TRTS can be remove if return type can be uniquely deduced.
{
    if      (x>+100) return (double)x+y;
    else if (x<-100) return (double)x-y;
    else             return (double)x/y;
}   (1,2);
```

Remark :

- lambda output is either defined by *TRTS* or *auto deduction* if return type can be uniquely deduced
- *TRTS* is used in `decltype` return type of an ordinary function or in lambda function

3. Capture list

Capture list specifies set of variables to be accessible inside lambda function body, without explicit passing as argument, by placing their names inside square brackets at the front, defaulted as capture-by-value, adorned with `&` for capture-by-reference.

```
int sum = 0;
std::for_each(vec.begin(), vec.end(), [&sum](int x){ sum += x; });

[ obj0]  (const X& x)->Y { ... } // 1. capture all obj0 by value
[&obj1]  (const X& x)->Y { ... } // 2. capture all obj1 by reference
[=, obj0](const X& x)->Y { ... } // 3. capture all local variables by value     and obj0 by value
[&,&obj1](const X& x)->Y { ... } // 4. capture all local variables by reference and obj1 by reference
[this]   (const X& x)->Y { ... } // 5. Capture all data members in the same object
```

Finally, when we implement lambda inside a member function, data members are not regarded as local block scope variables, they belong to class scope, thus we cannot put data members into the capture list (it results in compile error). However, `this` pointer is a hidden variable in the local block scope, thus we should capture `this` pointer when lambda needs to access data members.

```
struct statistics
{
    void find_sum()
    {
        std::for_each(vec.begin(), vec.end(), [this](const auto& x){ sum += x; });
    }

    std::vector<int> vec;
    int sum = 0;
};
```

4. Under the hood
Under-the-hood implementation of lambda is instantiation of an unnamed functor, constructed with all variables in the capture list, having function-call-operator being invoked with input argument list and returns the *TRTS* specified type.

```
// Using lambda
U u, V v;
auto fct = [u, &v](const X& arg)->Y { ... return Y{}; };
X x; Y y = fct(x);

// Under the hood
struct compiler_generated_lamda
{
    compiler_generated_lamda(U u, V& v) : m0(u), m1(v) {}
    Y operator()(const X& x) { ... return Y{}; }
    U m0; V& m1;
};

U u, V v;
auto fct = compiler_generated_lamda(u, v);
X x; Y y = fct(x);
```

E4. Overloading lamba

Overloading lambda is a lambda function offering multiple overloads, which are resolved according to input parameters. However it is not automatically supported in *STL C++*, we need to add two lines, one line is variadic template class, another line is called class template argument deduction CTAD, which is available in C++17 only :

```
// Two lines using three C++20 features (we will explain in section G)
template<typename...Ts> struct overloader : public Ts... { using Ts::operator()...; }; // line 1 variadic class
template<typename...Ts> overloader(Ts...) -> overloader<Ts...>;                         // line 2 variadic CTAD (not fct nor class)

// Using overloading lambda
U u, V v;
auto fct = overloader
{
    [u, &v](const X0& arg)->Y { v = v0; ... return Y{}; },
    [u, &v](const X1& arg)->Y { v = v1; ... return Y{}; },
    [u, &v](const X2& arg)->Y { v = v2; ... return Y{}; }
};
X0 x0; auto y0 = fct(x0); std::cout << v; // should print v0
X1 x1; auto y1 = fct(x1); std::cout << v; // should print v1
X2 x2; auto y2 = fct(x2); std::cout << v; // should print v2

// Under the hood
struct compiler_generated_lamda
{
    compiler_generated_lamda(U u, V& v) : m0(u), m1(v) {}
    Y operator()(const X0& x) { m1 = v0; ... return Y{}; }
    Y operator()(const X1& x) { m1 = v1; ... return Y{}; }
    Y operator()(const X2& x) { m1 = v2; ... return Y{}; }
    U m0; V& m1;
};

U u, V v;
auto fct = compiler_generated_lamda(u, v);
X0 x0; auto y0 = fct(x0); std::cout << v; // should print v0
X1 x1; auto y1 = fct(x1); std::cout << v; // should print v1
X2 x2; auto y2 = fct(x2); std::cout << v; // should print v2
```

Comparison among `nullptr` `std::function` and `auto` deduction

nullptr	std::function	auto deduction
ptr to object		
ptr to global function	global function	output from global function
ptr to member function	member function	output from member function
	functor	
		output from template function
	lambda function	output from lambda function

Comparison between *binding* and *capturing in lambda*

• bind by value	`bind(&T::f, x, ...`	capture by value	`[ x ] (auto y, auto z){ ... }`
• bind by pointer	`bind(&T::f, &x, ...`	capture by reference	`[&x ] (auto y, auto z){ ... }`
• bind by reference	`bind(&T::f, std:: ref(x) ...`	capture all by value	`[ = ] (auto y, auto z){ ... }`
• bind by const ref	`bind(&T::f, std::cref(x) ...`	capture all by reference	`[ & ] (auto y, auto z){ ... }`
• bind by rvalue ref	`bind(&T::f, std::move(x) ...`	capture this	`[this](auto y, auto z){ ... }`

F1. Type traits

Here are some common type traits for template programming.

```
template<typename T, T N> struct integral_constant
{
//   typedef T value_type;     // old syntax
     using     value_type = T; // new syntax
     static const T value = N;
};

typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

Besides `STL` offers shortcut to value and type for most of the above traits. Given a traits `xxx` :

- shortcut to value is implemented by variable template
- shortcut to type is implemented by alias template

```
// shortcut to value
template<typename T, typename U>
inline constexpr bool traits_xxx_v = traits_xxx<T,U>::value;

// shortcut to type
template<typename T, typename U>
using traits_xxx_t = typename traits_xxx<T,U>::type;
```

Here are some *is-type-traits* which primarily inherit from `false_type` and is specialized to inherit from `true_type`.

```
template<typename T, typename U> struct is_same         : std::false_type {}; // primary definition
template<typename T>             struct is_same<T,T>    : std:: true_type {};  // specialization
template<typename T> struct is_integral                 : std::false_type {};
template<>           struct is_integral<std::uint8_t>   : std:: true_type {};
template<>           struct is_integral<std::uint16_t>  : std:: true_type {};
template<>           struct is_integral<std::uint32_t>  : std:: true_type {};
template<>           struct is_integral<std::uint64_t>  : std:: true_type {};
template<typename T> struct is_pointer_helper           : std::false_type {};
template<typename T> struct is_pointer_helper<T*>       : std:: true_type {};
template<typename T> struct is_pointer                  : is_pointer_helper<typename std::remove_cv<T>::type> {};
template<typename T> struct is_reference                : std::false_type {};
template<typename T> struct is_reference<T&>            : std:: true_type {};
template<typename T> struct is_reference<T&&>           : std:: true_type {};
```

Here are some *remove-type-traits* which have same `typedef` in both primary definition and specialization.

```
template<typename T> struct remove_const                        { typedef T type; }; // or new syntax ...
template<typename T> struct remove_const<const T>               { typedef T type; }; // using type = T;
template<typename T> struct remove_volatile                     { typedef T type; };
template<typename T> struct remove_volatile<volatile T>         { typedef T type; };
template<typename T> struct remove_cv                           { typedef T type; };
template<typename T> struct remove_cv<const T>                  { typedef T type; };
template<typename T> struct remove_cv<volatile T>               { typedef T type; };
template<typename T> struct remove_cv<const volatile T>         { typedef T type; };
template<typename T> struct remove_pointer                      { typedef T type; };
template<typename T> struct remove_pointer<T*>                  { typedef T type; };
template<typename T> struct remove_pointer<T* const>            { typedef T type; };
template<typename T> struct remove_pointer<T* volatile>         { typedef T type; };
template<typename T> struct remove_pointer<T* const volatile>   { typedef T type; };
template<typename T> struct remove_reference                    { typedef T type; };
template<typename T> struct remove_reference<T&>                { typedef T type; };
template<typename T> struct remove_reference<T&&>               { typedef T type; };
template<typename T> struct decay
{
     typedef typename std::remove_reference<T>::type U;
     typedef typename std::conditional<    std::is_array<U>::value, typename std::remove_extent<U>::type*, // case 1
             typename std::conditional< std::is_function<U>::value, typename std::add_pointer<U>::type,    // case 2
                                                                     typename std::remove_cv<U>::type       // case 3
             >::type
             >::type type;
};
// For most cases, it is case 3, std::decay<T>::type is equivalent to std::remove_cv<std::remove_reference<T>>::type.
// When T is an array, it is case 1, with special treatment.
// When T is a function, it is case 2, with special treatment.
```

Here is a *is-template-traits* which checks whether a type is an instantiation of a template. Here is my naive implementation :

```
template<typename CT> struct is_vector                 { using type = std::false_type; };
template<typename  T> struct is_vector<std::vector<T>> { using type = std:: true_type; };

std::cout << is_vector<      std::vector<std::uint32_t> >::value; // return std::true_type
std::cout << is_vector<const std::vector<std::uint32_t> >::value; // return std::false_type
std::cout << is_vector<const std::vector<std::uint32_t>&>::value; // return std::false_type
```

In order to solve above problem, we need to add more specializations to `is_vector` traits regarding to `const`, `volatile` and reference, there are 8 combinations. To clumsy code, we prefer the following implementation which builds on overloading of `impl` :

```
template<typename CT> struct is_vector
{
private:
    using CT_REMOVED = std::remove_cvref_t<CT>;
    template<typename T>
    static auto impl(std::vector<T>*) -> std::true_type;         // with SFINAE, we can deduce T from CT
    static std::false_type impl(void*);                          // no definition needed

public:
    using type = decltype(impl(std::declval<CT_REMOVED*>()));
};

std::cout << is_vector<      std::vector<std::uint32_t> >::value; // return std::true_type
std::cout << is_vector<const std::vector<std::uint32_t> >::value; // return std::true_type
std::cout << is_vector<const std::vector<std::uint32_t>&>::value; // return std::true_type
```

Here are other conditional template traits :

```
template<bool B, typename T, typename F> struct conditional              { typedef T type; };
template        <typename T, typename F> struct conditional<false, T, F> { typedef F type; };
template<bool B, typename T = void>      struct enable_if                {                 };
template        <typename T>             struct enable_if<true, T>       { typedef T type; };
```

Main difference between `conditional` and `enable_if` is that the former always has a `typedef` while the latter doesnt, trying to access the `typedef` for `enable_if<false,T>` will result in compilation error, removing certain template overloads from overload resolution process. We will see how `enable_if` is used in SFINAE and why default template parameter `=void` is added in later section.

```
enable_if<is_integral<T>::value, U>::type object;
// compile ok  when T is integral type, then object is type U
// compile err when T is non integral
```

A little bit more complicated traits involving two template parameters. The following implementation of `is_base_of` is simplier than that suggested in `cppreference`, yet it works, tested in `gcc`.

```
template <typename B> std::false_type is_ptr_convertible_to(const void*);
template <typename B> std::true_type  is_ptr_convertible_to(const B*);
template <typename B, typename D>
struct is_base_of : public decltype(is_ptr_convertible_to<B>(static_cast<D*>(nullptr)))
{
};

// Testing example
class A {};
class B : public A {};
class C {};

std::cout << "\n" << std::boolalpha << is_base_of<A,B>::value; // true
std::cout << "\n" << std::boolalpha << is_base_of<B,A>::value; // false
std::cout << "\n" << std::boolalpha << is_base_of<B,C>::value; // false
std::cout << "\n" << std::boolalpha << is_base_of<C,C>::value; // true
```

F2. Pair-tuple / structured binding

Pair and tuple

- helper template class : `std::tuple_size, std::tuple_element`
- helper template function `std::get` for member access, `std::make_tuple` as factory
- helper template function : `td::tuple_cat` and `std::apply`

```
template<typename PAIR, typename TUPLE> auto merge_tuple(const PAIR& pair, const TUPLE& tuple)
{
    return std::make_tuple(pair.first, pair.second, std::get<0>(tuple), std::get<1>(tuple), std::get<2>(tuple));
}
template<typename PAIR, typename TUPLE> void algo(const PAIR& pair, const TUPLE& tuple)
{
    typename PAIR::first_type x0; = pair.first;
    typename PAIR::second_type x1; = pair.second;
    if constexpr (std::tuple_size<TUPLE>::value == 3)
    {
        typename std::tuple_element<0, TUPLE>::type y0;
        typename std::tuple_element<2, TUPLE>::type y2;
        std::tie(x0,x1,y0,std::ignore,y2) = merge_tuple(pair, tuple);
    }
}
```

What is `std::apply`? How to implement `std::apply`?

```
template<typename F, typename TUPLE, std::size_t...NS>
void apply_impl(F fct, const TUPLE& tuple, const std::index_sequence<NS...>& seq)
{
    fct(std::get<NS>(tuple)...);
}
template<typename F, typename... ARGS> void apply(F fct, const std::tuple<ARGS...>& tuple)
{
    apply_impl(fct, tuple, std::make_index_sequence<sizeof...(ARGS)>{});
}

// Testing program then becomes ... where T0,T1,... are concrete types
auto t0 = std::tuple<T0,T1,T2,T3,T4>{};
auto t1 = std::tuple<T0,T1,T2,T3,T4,T5,T6>{};
auto f0 = [](const T0& x0, const T1& x1, const T2& x2, const T3& x3, const T4& x4){...};
auto f1 = [](const T0& x0, const T1& x1, const T2& x2, const T3& x3, const T4& x4, const T5& x5, const T6& x6){...};
// Tradition way ... (1) cumbersome (2) need to change on adding elements to tuple
f0(std::get<0>(t0), std::get<1>(t0), std::get<2>(t0), std::get<3>(t0), std::get<4>(t0));
f1(std::get<0>(t1), std::get<1>(t1), std::get<2>(t1), std::get<3>(t1), std::get<4>(t1), std::get<5>(t1), std::get<6>(t1));
// Index sequence way ... (1) neat (2) no need to change on adding elements to tuple
apply(f0,t0);
apply(f1,t1);
```

Structured binding

Structured binding `auto [x,y,z]` offers a neat syntax for collecting returned product types. 5 common usages :

- pairs / tuples
- aggregates
- arrays
- insert to `std::map` or
- range for iteration through `std::map`

Binding local variable using `std::tie` with `std::ignore`

```
struct T           { int i_; double d_; std::string s_; };
auto factory_T0() { return std::make_pair(123, 3.1415);                      }
auto factory_T1() { return std::make_tuple(123, 3.1415, "This is pi.");     }
auto factory_T2() { return T{123, 3.1415, "This is pi."};                   }
T array[3] = {t0,t1,t2};

// instantiation
int i; double d; std::string s;
std::tie(i, d, s)  = factory_T1();
auto  [x1, y1, z1] = factory_T1();
auto& [x2, y2, z2] = factory_T2();
const auto& [x3, y3, z3] = array;
// insert to map
if (auto [iter, flag] = map.insert(std::make_pair(key, value)); flag) // iter = std::map<K,V>::iterator, flag = bool
{
    std::cout << "key = " << iter->first << "value = " << iter->second;
}
// iterate through map
for (const auto& [k,v] : map)
{
    std::cout << "key = " <<k << "value = " << v;
}
```

F3. Variant and visitor

In type theory, there are two algebraic types :

- product type such as tuple and aggregate (its feasible set equals to the product of feasible sets for all members)
- sum type such as union and variant (its feasible set equals to the sum of feasible sets for all members)
- feasible set of `struct { bool f; char c };` `(true,'a'), (true,'b'), (true,'c'), ... (false,'a'), (false,'b'), (false,'c'), ...`
- feasible set of `union { bool f; char c };` `true, false, 'a', 'b', 'c', ...`

Variant is a variadic template for sum type :

- type-safe version of union
- only one member is active
- all members share the same memory

Variant can be instantiated by :

- default initialization `std::variant<int, double, std::string> v0; // default as index 0`
- direct initialization `std::variant<int, double, std::string> v1(3.1415);`
- copy initialization `std::variant<int, double, std::string> v2 = "abcdef";`

Variant can be consumed by :

- check index and `std::get<index>` getting inactive member will throw exception
- check alternative and `std::get<type>` getting inactive member will throw exception
- apply `std::get_if<type>` on pointer check the return pointer before referencing
- apply visitor pattern `std::visit`
- apply visitor pattern and overloading lambda

```
void print0(const std::variant<int, double, std::string>& v)
{
    if      (v.index() == 0) std::cout << "\nint    = " << std::get<0>(v);
    else if (v.index() == 1) std::cout << "\ndouble = " << std::get<1>(v);
    else if (v.index() == 2) std::cout << "\nstring = " << std::get<2>(v);
}

void print1(const std::variant<int, double, std::string>& v)
{
    if      (std::holds_alternative<int>(v))         std::cout << "\nint    = " << std::get<int>(v);
    else if (std::holds_alternative<double>(v))      std::cout << "\ndouble = " << std::get<double>(v);
    else if (std::holds_alternative<std::string>(v)) std::cout << "\nstring = " << std::get<std::string>(v);
}

void print2(const std::variant<int, double, std::string>& v)
{
    if      (const auto& iter = std::get_if<int>(&v); iter)         std::cout << "\nint    = " << *iter;
    else if (const auto& iter = std::get_if<double>(&v); iter)      std::cout << "\ndouble = " << *iter;
    else if (const auto& iter = std::get_if<std::string>(&v); iter) std::cout << "\nstring = " << *iter;
}

struct visitor
{
    void operator()(const int& i)         const { std::cout << "\nint    = " << i; }
    void operator()(const double& d)      const { std::cout << "\ndouble = " << d; }
    void operator()(const std::string& s) const { std::cout << "\nstring = " << s; }
};

std::variant<int, double, std::string> v0;              print0(v0); print1(v0); print2(v0); std::visit(visitor{}, v0);
std::variant<int, double, std::string> v1(3.14);        print0(v1); print1(v1); print2(v1); std::visit(visitor{}, v1);
std::variant<int, double, std::string> v2 = "abcdef";   print0(v2); print1(v2); print2(v2); std::visit(visitor{}, v2);

std::visit(overloader
{
    [](const int& i)         { std::cout << "\nint    = " << i; },
    [](const double& d)      { std::cout << "\ndouble = " << d; },
    [](const std::string& s) { std::cout << "\nstring = " << s; }
}, v0);
```

F4. Reference wrapper

Please read "Pointers, References and Optional References" in Fluent C++

	nullity	reassignment	disadvantages
reference	can't be null	can't be reassigned	no default construct, noncopyable, nonmovable
`reference_wrapper<T>`	can't be null	can be reassigned	can't represent null
`optional<reference_wrapper<T>>`	can be null	can be reassigned	need null checking before use
pointer	can be null	can be reassigned	need null checking before use

Reference is a stuborn alias of object. With its non-copyability and non-movability :

- declaring reference as class member like `std::tuple<T&...>` makes the class non-copyable/movable (copy constructor deleted)
- declaring reference inside `std::vector<T&>` and `std::optional<T&>` will not compile

Thus a copyable and movable reference wrapper `std::reference_wrapper<T>` is designed to emulates a reference :

- it is internally implemented as `T* _ptr`
- it cannot be null, it cannot reference to temporary object
- it cannot be default constructed
- it can be copied or moved
- it is created by factory : `std::reference_wrapper<T>` by `std::ref()` and `std::reference_wrapper<const T>` by `std::cref()`
- it is accessed by member : `std::reference_wrapper<T>::get()`

In particular :

- we can use `std::optional<std::reference_wrapper<const V>>` as return value for map's find key function
- we can declare `void fct(std::reference_wrapper<T> x)` passing wrapper as argument, no `&` is needed
- we can declare `class T{ std::reference_wrapper<int> x }` while keeping `T` copyable and movable

Please note the differences among `std::pair` / `std::tuple`, `std::vector` and `std::optional` :

- `std::pair<T&,U&>` / `std::tuple<T&,U&,V&>` does compile, but it is not copyable
- `std::vector<T&>` / `std::optional<T&>` do **not** compile
- factory `std::make_pair(std::ref(t),std::ref(u))` creates `std::pair<T&,U&>`, **not** creates `std::pair<std::reference_wrapper<T>,...>`
- factory `std::make_optional(std::ref(t))` creates `std::optional<std::reference_wrapper<T>>`, **not** creates `std::optional<T&>`

```
     struct T { int mem0; int mem1; int mem2; };
     struct X { T& m;                         };
     struct Y { std::reference_wrapper<T> m;  };

//   void fct(std::reference_wrapper<const T>& ref) // compile error, as we cannot bind lvalue& to rvalue (std::ref creates rvalue)
     void fct(std::reference_wrapper<const T>  ref) { std::cout << ref.get(); }

     // *** Subtest 1 *** //
     T obj(1,2,3);
     auto r0 = std::ref(obj);
     auto r1 = std::cref(obj);
     std::cout << std::is_same_v<decltype(r0), std::reference_wrapper<T>>;              // true
     std::cout << std::is_same_v<decltype(r1), std::reference_wrapper<const T>>;        // true
     std::cout << std::hex << &(obj.mem0) << &(r0.get().mem0) << &(r1.get().mem0);      // same address
     std::cout << std::hex << &(obj.mem1) << &(r0.get().mem1) << &(r1.get().mem0);      // same address

     // *** Subtest 2 *** //
     std::pair<T&,T&> p0{obj,obj};                                                      // compile OK
     auto p1 = std::make_pair(std::ref(obj), std::ref(obj));
     p1.first.mem0 = 111;
     p1.first.mem1 = 222;
     p1.first.mem2 = 333;
     std::cout << std::is_same_v<decltype(p1), std::pair<std::reference_wrapper<T>, std::reference_wrapper<T>>>;  // false
     std::cout << std::is_same_v<decltype(p1), std::pair<T&,T&>>;                                                 // true
     std::cout << p1.second;

    // *** Subtest 3 *** //
//   std::vector<T&> v;                                                                 // cannot compile
     std::vector<std::reference_wrapper<T>> v{obj, obj, obj};
     v[0].get().mem0 = 444;
     v[1].get().mem1 = 555;
     v[2].get().mem2 = 666;
     for(const auto& x:v) std::cout << x.get();

    // *** Subtest 4 *** //
//   std::optional<T&> opt(obj);                                                        // cannot compile
     auto opt = std::make_optional(std::ref(obj));
     std::cout << std::is_same_v<decltype(opt), std::optional<std::reference_wrapper<T>>>; // true
     std::cout << std::is_same_v<decltype(opt), std::optional<T&>>;                     // false
     if (opt) std::cout << opt->get();

     // *** Subtest 5 *** //
     fct(std::cref(obj)); // std::cref creates a temporary std::reference_wrapper<const T>

     // *** Subtest 6 *** //
     X x(obj), x2(obj);   // both are ok ...
     Y y(obj), y2(obj);   // both are ok ...
     std::cout << x.m;
     std::cout << y.m ;   // same value
//   x2 = x; // compile error
     y2 = y;
```

F5. Optional

Optional is a template wrapper for any class `T`, so that :

- it allows `T` to bear an un-initialized value `std::nullopt`
- it is accessed like a pointer, with `std::nullopt` analogous to `nullptr`
- it is used as output of `find` function in containers, such as `std::optional<V> my_map<K,V>::get(const K&) const`

Optional can be instantiated by :

- default initialization `std::optional<T> opt; // initialized as std::nullopt`
- direct initialization `std::optional<T> opt(T{arg0,arg1,arg2}); // or equivalently ...` `std::optional<T> opt( {arg0,arg1,arg2});`
- copy initialization `std::optional<T> opt = T{arg0,arg1,arg2};`
- factory `std::optional<T> opt = std::make_optional(T{arg0,arg1,arg2}); // or equivalently ...` `std::optional<T> opt = std::make_optional( {arg0,arg1,arg2});`

Optional can be consumed by :

- check existence by `if (opt) ...`
- dereferenced by `(*opt).fct();`
- redirected by `opt->fct();`

Application in `map`

```
std::optional<V> get(const K& key) const
{
    std::optional<V> output = std::make_optional(impl[key]);
    ...
    return output; // using named-return-value-optimization
}
```

Example tested in MSVS

- optional of simple object
- optional of movable object
- optional of reference wrapper
- optional of reference cannot compile, as `T&` must be initialized, while `std::reference_wrapper<T>` delay-initialized

```
struct T { int i; double d; std::string s; };                     // assume T to be movable

void print(std::optional<T>& opt)
{
    if (!opt) std::cout << "\nEmpty T";
    else      std::cout << "\nInit T = " << opt    ->i << " " << opt->d << " " << opt->s;
}
void print(std::optional<std::reference_wrapper<T>>& opt)
{
    if (!opt) std::cout << "\nEmpty T_ref";
    else      std::cout << "\nInit T_ref = " << opt->get().i << " " << opt->get().d << " " << opt->get().s;
}

// Optional of simple object
std::optional<T> opt_0;                                           print(opt_0); // as expected
std::optional<T> opt_1(T{100, 3.14, "abc"});                      print(opt_1); // ...
std::optional<T> opt_2 = T{200, 1.23, "def"};                     print(opt_2); // ...
auto opt_3 = std::make_optional<T>({300, 9.99, "----"});          print(opt_3); // ...

// Optional of movable object
T obj{12345, 0.12345, "temp-obj"};
std::optional<T> opt_4(obj);                                      // invoke copy semantics
std::optional<T> opt_5(std::move(obj));                           // invoke move semantics
opt_4 = obj;                                                      // invoke copy semantics
opt_5 = std::move(obj);                                           // invoke move semantics

// Optional of reference
//  std::optional<T&> opt(obj);                                   // compile error, cannot declare optional of T&
//  std::optional<std::reference_wrapper<T>> opt(obj);            // compile error, T& != std::reference_wrapper<T>
std::optional<std::reference_wrapper<T>> opt7;
std::optional<std::reference_wrapper<T>> opt8(std::ref(obj));
std::optional<std::reference_wrapper<T>> opt9 = std::ref(obj);
auto optA = std::make_optional<std::reference_wrapper<T>>(std::ref(obj));

optA->get().i = 54321;
optA->get().d = 0.54321;
optA->get().s = "temp-obj modified";
print(opt7);                                                      // modified as expected
print(opt8);                                                      // modified as expected
print(opt9);                                                      // modified as expected
```

F6. Any

If `std::variant` can be considered as an analogy to union, then `std::any` can be considered as an analogy to `void*`.

- `std::variant` is a variadic template, while `std::any` does not specify its types in compile time
- `std::variant` is compile time allocated, while `std::any` may involve dynamic allocation
- `std::variant` is default initialized as index 0, while `std::any` is default initialized as `no-value`

Any can be instantiated by :

- default initialization `std::any a;`
- direct initialization `std::any a(123);`
- copy initialization `std::any a = std::string("abc");`
- factory `auto a = std::make_any<T>(arg0,arg1,arg2);`

Any can be consumed by :

- check existence by `if (a.has_value()) ...`
- get by `T t = std::any_cast<T>(a);`
- reset by `a.reset();`

Remark

Please note :

- Both `std::make_tuple` and `std::tie` are aggregation to generate tuple. The former can store value, the latter cannot.
- Both `std::tuple` and `std::variant` are visited by :

```
std::apply(functor, tuple);
std::visit(overloader, variant);
```

- Both `std::make_pair` and `std::tuple` factories can be replaced by CTAD plus deduction guide.

More about Template programming

G1 Variadic template
- 10 expansion loci
- 9 implementation examples

G2 Explanation of overloading lambda
- 3 new features in C++17
- 4 steps to overloading lambda

G3 SFINAE
- define customised type traits
- define overloads for template function and
  define overloads for template class member function
- the perfect forwarding example

G4 SFINAE - building complicated type traits
- adding template parameter for `TEST<T>`
- the container example and use of `std::void_t`
- the iterator example and use of `std::declval` and `decltype`

G5 Index sequence
- how to use `std::index_sequence` and its factory `std::make_index_sequence`
- how to implement index sequence and its factory (various factories)
- useful application

G6 Generalized constant expression

•[3]	use of `if constexpr`	since C++17
•[6]	use of `constexpr` for global function	since C++11
•[6]	use of `constexpr` for class	since C++11
•[1]	use of `consteval`	since C++20
•[1]	use of `constinit`	since C++20
-	`constexpr` and `constexpr` are two different features	

<u>Constraining template parameters</u>

When we define template function or class with on template parameter `T`, how can be apply constraints on `T`? There are 3 ways :

•	using *SFINAE* for C++14 or previous version	see G3 and G4
•	using `if constexpr` for C++17	see G5
•	using `concept` for C++20	see C++20 doc

We will go through some examples using *SFINAE* in this section :

•	limit `T` to simple target types	see section G3
•	limit `T` to types offering `T::value_type`	see section G4 which needs `std::void_t`, `std::declval` and `decltype`
•	limit `T` to types offering `T::operator++()`	see section G4 which needs `std::void_t`, `std::declval` and `decltype`

<u>Template programming for runtime and compile-time</u>

Lets take variadic template example `std::tuple`

•	runtime	`std::get<3>(tuple)`
•	compile time	`std::tuple_size<std::tuple<A,B,C,D>>::value`

G1. Variadic template – parameter pack (C++11) and fold expression (C++17)

There are different ways to expand a parameter pack, please refer to cpp-reference. It is known as **expansion loci**. Let's assume that here are 3 parameters in the pack for the ease of illustration. In general ellipsis operator expands the nearest part of statement on its *LHS*. The syntax is slightly different when using `sizeof...` only. **Fold expression** is introduced in C++17, it is defined as a parameter pack operated on binary operator. Fold expression is usually used together with comma operator in `std::apply`, please refer to PSQL interface (my design document).

```
template<typename... Ts> void fct(Ts... args)
{
    // case 1 : args...
    g(x, y, f(args ...));                          => g(x, y, f(arg0, arg1, arg2));
    g(x, y, f(args)...);                           => g(x, y, f(arg0), f(arg1), f(arg2));
    g(x, y, ++args ...);                           => g(x, y, ++arg0,  ++arg1,  ++arg2);
    g(x, y,  &args ...);                           => g(x, y,  &arg0,   &arg1,   &arg2);

    // case 2 : nested args...
    g(x, y, f(args...) + args...);                 => g(x, y, f(args...) + arg0,
                                                              f(args...) + arg1,
                                                              f(args...) + arg2);
                                                   => g(x, y, f(arg0, arg1, arg2) + arg0,
                                                              f(arg0, arg1, arg2) + arg1,
                                                              f(arg0, arg1, arg2) + arg2);

    // case 3 : Ts(args)...
    g(x, y, const_cast<const Ts*>(&args)...);      => g(x, y, const_cast<T0*>(&arg0),
                                                              const_cast<T1*>(&arg1),
                                                              const_cast<T2*>(&arg2));
    g(x, y, std::forward<Ts>(args)...);            => g(x, y, std::forward<T0>(arg0),
                                                              std::forward<T1>(arg1),
                                                              std::forward<T2>(arg2));

    // case 4 : args... in right fold expression
    (f(args) OP ...);                              =>        f(arg0) OP (f(arg1) OP  f(arg2))
    (f(args) OP ... OP x);                         =>        f(arg0) OP (f(arg1) OP (f(arg2) OP x))
         (... OP f(args));                         =>       (f(arg0)  OP f(arg1)) OP f(arg2)
    (x OP ... OP f(args));                         => ((x OP f(arg0)) OP f(arg1)) OP f(arg2)

    // case 1" : Ts... in template parameter
    std::tuple<Ts...,X,Y> temp;                    => std::tuple<T0,T1,T2,X,Y> temp;
    std::tuple<typename Ts::type...,X,Y> temp;     => std::tuple<T0::type,T1::type,T2::type,X,Y> temp;

    // case 2" : Ts... in inheritance
    class A : public Ts...                         => class A : public T0, public T1, public T2
    {                                                 {
         A(const Ts&... args)                             A(const T0& arg0, const T1& arg1, const T2& arg2)
         : Ts(args)...                                    : T0(arg0), T1(arg1), T2(arg2)
         {}                                               {}
    };                                                };

    // case 3" : Ts... in using (usually work with case 3")
    using Ts::fct...;                              => using T0::fct, using T1::fct, using T2::fct;

    // case 4" : sizeof...(Ts)
    int array[sizeof...(Ts)+2] = {x, y, args...};  => int array[3+2] = {x, y, arg0, arg1, arg2};
}
```

Here are some examples for variadic template.

```
// Example 1. Forwarding with recursion (2 approaches)
```

```
// one-arg boundary case
template<typename T>
void alg(T&& arg)
{
    imp(std::forward<T>(arg));
}

// general case
template<typename T, typename... Ts>
void alg(T& arg, Ts&&... args)
{
    alg(std::forward<T>(arg));
    alg(std::forward<T>(args)...);
}
```

OR

```
// empty boundary case
// non-template
void alg()
{
    // do nothing
}

// general case
template<typename T, typename... Ts>
void alg(T&& arg, Ts&&... args)
{
    imp(std::forward<T>(arg));
    alg(std::forward<T>(args)...);
}
```

```
// Example 2. Forwarding without recursion
template<typename T> class vector
{
    template<typename... Ts> void emplace_back(Ts&&... args)
    {
        new (impl[next_write]) T(std::forward<Ts>(args)...); // please verify the placement new syntax
        ++next_write;
    }

    T impl[128];
    int next_write;
}


// Example 3. Fold expression for logger
template<typename OS, typename... Ts> void printer(OS& os, Ts&&... args)
{
    (os << ... << std::forward<Ts>(args));
}

printer(std::cout, 3.1415, "This is an apple", T(1,2,3));


// Example 4. Fold expression for vector insertion (using comma operator)
template<typename C, typename... Ts> void batch_push_back(C& container, Ts&&... args)
{
    (..., container.push_back(std::forward<Ts>(args)));
//  Expand the above, we have :
//  (((container.push_back(std::forward<T0>(arg0)), container.push_back(std::forward<T1>(arg1))),
//                                                  container.push_back(std::forward<T2>(arg2))),
//                                                  container.push_back(std::forward<T3>(arg3)));
}

alg::variant_vector vv;
batch_push_back(vv, 123, 3.1415, "ABCDEF", T(1,2,3));


// Example 5. Factory make_tuple
template<typename... Ts> auto make_tuple(Ts&&... args)
{
    // There are two parameter pack expansions :
    // one for Ts,   see case 2" in previous section
    // one for args, see case 3  in previous section
    // but they are not nested.
    return std::tuple<std::decay<Ts>...>(std::forward<Ts>(args)...);
}


// Example 6. Factory tie
template< typename... Ts> auto tie(Ts&... args)
{
    // Main difference between make_tuple and tie is that
    // the former takes universal reference input
    // the latter takes lvalue reference input
    return std::tuple<Ts...>(args...);
}
```

G2. Explanation of overloading lambda

Read "Using that overloaded trick : Overloading Lambdas in C++17" by Tamir Bahar. Overloading lambda is implemented by :

```
template<typename... Ts> struct overloader : public Ts... { using Ts::operator()...; };
template<typename... Ts> overloader(Ts...) -> overloader<Ts...>;
```

1. Make use of three C++17 new features

- variadic `using Ts::operator()...`
- aggregate initialization for derived class
- auto deduction for template class using class-template-argument-deduction CTAD, and deduction guide

1a. about `using`

`using` is an alias, there are two special usages relevant to template :

```
// (1) template alias
template<typename T> using iter = std::vector<T>::iterator;
iter<int> i = vec.begin();

// (2) alias inside variadic template class derived from multiple base
template<typename... BASES> derived : public BASES...
{
    using BASES::fct...;
    void fct(const T_DERIVED& arg);
};
```

In the second case, `using BASES::fct...` means bringing function `BASES::fct` in each base class to the scope of derived class. If each of them has a different prototype, like :

```
BASE0::fct(const T_BASE0& arg);
BASE1::fct(const T_BASE1& arg);
BASE2::fct(const T_BASE2& arg);
derived::fct(const T_DERIVED& arg);
```

then the compiler will try to resolve among them whenever `fct(x)` is invoked, according to the type of `x`.

1c. about CTAD and deduction guide

Before C++17 template type deduction works only for template function, but not for template class. Since C++17 we can do the same for template class with CTAD feature. Suppose a template class object is declared with `my_class obj(x,y,z)`, compiler will resolve for the appropriate constructor, as well as type deduction of `U/V/T`. However CTAS may fail (i.e. unresolve constructor or fail to deduce all template parameters), in that case we need to provide user-defined deduction guide(s) to help compiler.

```
template<typename U, typename V, typename T> struct my_class
{
    my_class(const U& u) // CTAD fails, need deduction guide
    {
        p = std::make_pair(u, 100); for(int n=0; n!=5; ++n) vec.push_back('x');
    }
    my_class(const U& u, const V& v) // CTAD fails, need deduction guide
    {
        p = std::make_pair(u,v); for(int n=0; n!=10; ++n) vec.push_back(v+n);
    }
    my_class(const U& u, const V& v, const T& t0, const T& t1) // CTAD works, as long as all U/V/T can be deduced
    {
        p = std::make_pair(u,v); for(auto t=t0; t!=t1; ++t) vec.push_back(t);
    }

    std::pair<U,V> p;
    std::vector<T> vec;
};

// Deduction guides
template<typename U>             my_class(const U& u)             -> my_class<U,int,char>;
template<typename U, typename V> my_class(const U& u, const V& v) -> my_class<U,V,V>;

// Testing
my_class x0{std::string("Test obj0")};
my_class x1{std::string("Test obj1"), 0.123};
my_class x2{std::string("Test obj2"), 0.123, 50, 60};
std::cout << std::is_same_v<decltype(x0), my_class<std::string, int, char>>;
std::cout << std::is_same_v<decltype(x1), my_class<std::string, double, double>>;
std::cout << std::is_same_v<decltype(x2), my_class<std::string, double, int>>;
```

2. Break it down into 4 steps

First of all, lets define a functor as follows :

```
struct v0 { void operator()(const int& i)         const { std::cout << "\nint    = " << i; } };
struct v1 { void operator()(const double& d)      const { std::cout << "\ndouble = " << d; } };
struct v2 { void operator()(const std::string& s) const { std::cout << "\nstring = " << s; } };
struct visitor : public v0,v1,v2
{
    using v0::operator(); // explicit using declaration is needed, otherwise compile error when trying to invoke operator()
    using v1::operator();
    using v2::operator();
};

visitor x;
x(123); x(3.14); x(std::string("abc"));
```

Secondly, we make `v0/v1/v2` into lambda and make `visitor` a variadic template. This step makes use of two new features in C++17.

```
auto v0 = [](const int& i)         { std::cout << "\nint    = " << i; };
auto v1 = [](const double& d)      { std::cout << "\ndouble = " << d; };
auto v2 = [](const std::string& s) { std::cout << "\nstring = " << s; };

template<typename... Ts> struct visitor : public Ts...
{
    using Ts::operator()...; // explicit using declaration for parameter pack [New feature 1]
};

visitor<decltype(v0), decltype(v1), decltype(v2)> x{ v0,v1,v2 }; // aggregate initialization for derived class [New feature 2]
x(123); x(3.14); x(std::string("abc"));
```

Thirdly, try to remove the ugly `decltype` by providing factory function.

- `visitor` is a template class, we need to specify template parameter for instantiation
- `make_visitor` is a template function, auto deduction for template parameter saves us from ugly `decltype`
- `make_visitor` is similar to the implementation of `std::make_tuple` or `std::tie`

```
template<typename... Ts> struct visitor : public Ts... { using Ts::operator()...; };
template<typename... Ts> auto make_visitor(Ts... args) { return visitor<Ts...>{args...}; }
// Why not taking (Ts&... args) or (Ts&&... args) as input? Because args are lambda.

auto x = make_visitor
(
    [](const int& i)         { std::cout << "\nint    = " << i; },
    [](const double& d)      { std::cout << "\ndouble = " << d; },
    [](const std::string& s) { std::cout << "\nstring = " << s; }
);
x(123); x(3.14); x(std::string("abc"));
```

Finally, in C++17 *class template argument deduction CTAD* is introduced. It is a template guide, not a template function nor a template class, which helps compiler to deduce template parameter type without factory. Without *CTAD* compiler can only deduce template parameter for template function (but not for template class). Now replace `make_visitor` by a guide to `visitor` :

```
template<typename... Ts> struct visitor : public Ts... { using Ts::operator()...; };
template<typename... Ts> visitor(Ts...) -> visitor<Ts...>; // CTAD [New feature 3]
```

It allows type deducton of class template on construction of `visitor(A&,B&,C&)` object as type `visitor<A,B,C>`.

G3. SFINAE

1. Define type traits

Type traits is template class of `T` which inherits from `true_type` when `T` fulfills certain criteria, and inherits from `false_type` otherwise. Basic idea is that when `T` satisfies both primary definition and one or more specializations, then there is a higher priority for picking the specialization. Here is a type traits checking whether `T` belongs to target set.

```
template<typename T> struct is_target_type    : std::false_type {}; // primary definition
template<>           struct is_target_type<A> : std:: true_type {}; // specialization A
template<>           struct is_target_type<B> : std:: true_type {}; // specialization B
template<>           struct is_target_type<C> : std:: true_type {}; // specialization C

std::cout << is_target_type<A>::value;            // print  true
std::cout << is_target_type<int>::value;          // print false
std::cout << is_target_type<std::string>::value;  // print false
```

2a.Define overloads for template functions

Let's consider a template function with two implementations. How can we force the compiler to resolve to the first one for all target types `T` during compile time and resolve to the second one otherwise?

```
template<typename T> void fct(const T& x) {         // implementation for     target_type T     }
template<typename T> void fct(const T& x) {         // implementation for non_target_type T     }
```

The idea is to make a substitution of `T = target_type` fails for the second implementation, with SFINAE, a substitution failure is not a compilation error, it only removes failed-overload (which is the second implementation in this case) from the resolution-list, and as a result, it can only resolve to the first implementation for `T = target_type`. Vice versa for `T = non_target_type`.

```
// Method 1 : std::enable_if as return type
template<typename T> typename std::enable_if< is_target_type<T>::value, void>::type fct(const T& x){...}
template<typename T> typename std::enable_if<!is_target_type<T>::value, void>::type fct(const T& x){...}

// Method 2 : std::enable_if as extra template parameter
template<typename T, typename std::enable_if< is_target_type<T>::value, int>::type DUMMY = 0> void fct(const T& x){...}
template<typename T, typename std::enable_if<!is_target_type<T>::value, int>::type DUMMY = 0> void fct(const T& x){...}
```

In method 2, introduce an extra non-type template parameter to the template function :

- its type does not matter (as long as it can be easily assigned with a default value) `int` is picked for convenience
- its name does not matter, `DUMMY` is picked as the name, we can also unname it
- its value does not matter, `=0` is picked as the default value
- however a default value `=0` is a must, because caller never fill it during invocation

```
fct(A{a,b,c});                           // resolves to 1st implementation
fct(B{d,e,f});                           // resolves to 1st implementation
fct(123);                                // resolves to 2nd implementation
fct("abcdef");                           // resolves to 2nd implementation
fct(std::vector<int>{1,2,3,4,5,6,7});    // resolves to 2nd implementation
```

2b.Define overloads for template class member functions

For template class with template parameter `T` *SFINAE* does not work. As `T` is not deduced, substitution failure will result in error.

```
template<typename T> class algo // DOES NOT WORK
{
    typename std::enable_if< is_target_type<T>::value, void>::type fct(){...}
    typename std::enable_if<!is_target_type<T>::value, void>::type fct(){...}
};

// compilation error for both following calls
algo<A> obj0;     // when T=A,   !is_target_type<A>  ::value is false, and there is no return type for 2nd overload
algo<int> obj1;   // when T=int,  is_target_type<int>::value is false, and there is no return type for 1st overload
```

Therefore we should not put non-deduced type `T` into `std::enable_if<T>`, instead we introduce dummy type `U` which is default to be `T`, and we need to put `U` inside `std::enable_if<U>`, then substitution failure of `U` is not considered as error.

```
// Method 1 : std::enable_if as return type
template<typename T> class algo
{
    template<typename U=T> typename std::enable_if< is_target_type<U>::value, void>::type fct(){...}
    template<typename U=T> typename std::enable_if<!is_target_type<U>::value, void>::type fct(){...}
};

// Method 2 : std::enable_if as extra template parameter
template<typename T> class algo
{
    template<typename U=T, typename std::enable_if< is_target_type<U>::value, int>::type DUMMY = 0> void fct(){...}
    template<typename U=T, typename std::enable_if<!is_target_type<U>::value, int>::type DUMMY = 0> void fct(){...}
};
```

3. The perfect forwarding example

This technique can be used in perfect forwarding.

```
template<typename T, typename std::enable_if<std::is_lvalue_reference<T>::value, int>::type = 0> implement(T&& x)
{
    // lvalue implementation
}
template<typename T, typename std::enable_if<!std::is_lvalue_reference<T>::value, int>::type = 0> implement(T&& x)
{
    // non-lvalue implementation
}
template<typename T> void perfect_forwarding(T&& x)
{
    implement(std::forward<T>(x));
}
```

G4. SFINAE - Building complicated type traits

1. Adding extra parameter

To implement complicated type traits, we extend the traits into a template with two template parameters `T` and `U` :

- `T` is the type under test
- `U` is the test for target type, we have `U=TEST<T>` in specialization

→ when `TEST<T>` does compile, compiler resolves to the specialization as it has a higher priority

→ when `TEST<T>` doesn't compile, substitution fails for the specialization, compiler resolves to the generic definition

```
template<typename T, typename U> struct my_traits            : std::false_type {}; // generic definition
template<typename T>             struct my_traits<T, TEST<T>> : std:: true_type {}; // specialization

// for example, we have a traits that test if T::value_type exists ...
template<typename T, typename U> struct is_container                              : std::false_type {}; // generic definition
template<typename T>             struct is_container<T, typename T::value_type> : std:: true_type {}; // specialization
```

2. The container example and use of `std::void_t`

Yet we do not want to invoke the traits in the following way :

```
std::cout << is_container<int, void>                                   ::value;  // print 0
std::cout << is_container<std::string, std::string::value_type>          ::value;  // print 1
std::cout << is_container<std::vector<int>, std::vector<int>::value_type> ::value;  // print 1
```

Instead we want to invoke the traits in the following way :

```
std::cout << is_container<int>              ::value;                           // print 0
std::cout << is_container<std::string>      ::value;                           // print 1
std::cout << is_container<std::vector<int>> ::value;                           // print 1
```

We need to do two things to fix it :

- add a default type for `U` so that we don't need to fill it, for example `U = void`
- add a mapping which maps whatever `TEST<T>` to the same default type as `U`, that is `map2void<TEST<T>>::type -> void`

→ hence compiler resolves to specialization `my_traits<T,void>` as long as `TEST<T>` exists

```
template<typename T, typename U = void> struct is_container                                          : std::false_type {};
template<typename T>                    struct is_container<T, map2void<typename T::value_type>::type> : std:: true_type {};
template<typename T> struct map2void  { typedef void type; };
```

We want to make it simpler, by replacing `map2void<...>::value` with `map2void<...>`, hence we rewrite it with `using` :

```
template<typename T, typename U = void> struct is_container                                    : std::false_type {};
template<typename T>                    struct is_container<T, map2void<typename T::value_type>> : std:: true_type {};
template<typename T> using map2void = void; // It means replacing all map2void<T> by void.
```

In fact, `map2void<T>` is ready in STL, its name is `std::void_t<T>`.

3. The iterator example and use of `std::declval / decltype`

Another example, lets construct a traits that returns `true` when `T::operator()++` exists.

```
template<typename T, typename = void> struct is_iterator                                          : std::false_type {};
template<typename T>                  struct is_iterator <T, std::void_t<decltype(++std::declval<T>())>> : std:: true_type {};

std::cout << is_iterator<std::vector<int> ::iterator>::value; // print 1
std::cout << is_iterator<std::list<double>::iterator>::value; // print 1
```

In this `is_iterator` example, `TEST<T>` expression is more complicated :

- `std::declval<T>()` returns an expression of object `T` without actually constructing an object `T`
- `std::declval<T>()` is used instead of `T{}` because we don't know the prototype of `T` constructor
- we apply `++std::declval<T>()` as a test for iterator

4. The has-type example / has-member-x example and use of `map2dummy`

`map2dummy` is a variadic template alias to another type, such as `dummy` in this case.

```
struct dummy{};
template<typename...Ts> using map2dummy = dummy;

template<typename T, typename = dummy> struct has_my_type                              : public std::false_type {};
template<typename T>                   struct has_my_type<T, map2dummy<typename T::my_type>> : public std:: true_type {};

struct A { using    my_type = std::string; };
struct B { using other_type = std::string; };

std::cout << has_my_type<A>::value;               // 1
std::cout << has_my_type<B>::value;               // 0
std::cout << has_my_type<A,dummy>::value;         // 1
std::cout << has_my_type<B,dummy>::value;         // 0
std::cout << has_my_type<A,std::vector>::value;   // 0 why is this not working?
std::cout << has_my_type<B,std::vector>::value;   // 0
```

This is the main requirement in the traits.

Why does the second template parameter have to be the same as default parameter of `has_my_type` in order to work? This is because the template resolution goes through the following steps :

1 when we invoke `has_my_type<A>`
- since we don't have a second parameter, it will be filled with default value as `has_my_type<A,dummy>` according to generic version
- then for generic version and for each specialization, we do substitution *(if substitution fails, it is not an error)*
- generic version `has_my_type<T,U>` succeeds with substitution `T=A U=dummy`
- specialization `has_my_type<T,map2dummy<T::my_type>>` succeeds with substitution `T=A`, picks this one as specialization has priority

2 when we invoke `has_my_type<B>`
- since we don't have a second parameter, it will be filled with default value as `has_my_type<B,dummy>` then similarly
- generic version `has_my_type<T,U>` succeeds with substitution `T=B U=dummy`
- specialization `has_my_type<T,map2dummy<T::my_type>>` fails with substitution `T=B`

In other words, the following does not work.

```
template<typename T, typename = void> struct has_my_type                              : public std::false_type {};
template<typename T>                  struct has_my_type<T, map2dummy<typename T::my_type>> : public std:: true_type {};
```

Here is another example : checking for a member with name `x` :

```
template<typename T, typename = dummy> struct has_x                              : public std::false_type {};
template<typename T>                   struct has_x<T, map2dummy<decltype(T::x)>> : public std:: true_type {};

struct A { int x;                 };
struct B { std::string x;         };
struct C { std::string x(int,int); };
struct D { std::string y;         };

std::cout << has_x<A>::value;               // 1
std::cout << has_x<B>::value;               // 1
std::cout << has_x<C>::value;               // 0
std::cout << has_x<D>::value;               // 0
std::cout << has_x<A,dummy>::value;         // 1
std::cout << has_x<B,dummy>::value;         // 1
std::cout << has_x<C,dummy>::value;         // 0
std::cout << has_x<D,dummy>::value;         // 0
std::cout << has_x<A,std::string>::value;   // 0
std::cout << has_x<B,std::string>::value;   // 0
std::cout << has_x<C,std::string>::value;   // 0
std::cout << has_x<D,std::string>::value;   // 0
```

This is the main requirement in the traits.

G5. Index sequence

Please read *Generating Integer Sequences at Compile Time, by Jacek.*

1. How to use `std::index_sequence` and its factory `std::make_index_sequence`?

This is the objective we want to achieve :

```
template<typename C, std::size_t... NS> auto vec2tuple(const C& container)
{
    return std::make_tuple(container[NS]...);
}

template<typename T, std::size_t... NS> auto tuple2tuple(const T& tuple)
{
    return std::make_tuple(std::get<NS>(tuple)...);
}

// Testing program
std::vector<std::string> v = {"111", "222", "333", "444", "555"};
auto t = std::tuple<std::uint32_t, std::uint32_t, std::string, std::string, std::string>{111, 222 , "333", "444", "555"};

auto t0 =   vec2tuple<decltype(v),0,2,4>  (v);
auto t1 =   vec2tuple<decltype(v),4,3,2,1>(v);
auto t2 = tuple2tuple<decltype(t),0,2,4>  (t);
auto t3 = tuple2tuple<decltype(t),4,3,2,1>(t);
```

However we need to input all template parameters to `vec2tuple` and `tuple2tuple`, can we make them auto-deduced? Yes use template class `std::index_sequence` as an extra dummy argument to the functions, template parameters of `std::index_sequence` are all non-type.

```
template<typename C, std::size_t... NS>
auto vec2tuple(const C& container, const std::index_sequence<NS...>& dummy)
{
    return std::make_tuple(container[NS]...);
}

template<typename T, std::size_t... NS>
auto tuple2tuple(const T& tuple, const std::index_sequence<NS...>& dummy)
{
    return std::make_tuple(std::get<NS>(tuple)...);
}

// Testing program then becomes ...
auto t0 =   vec2tuple(v, std::index_sequence<0,2,4>{});
auto t1 =   vec2tuple(v, std::index_sequence<4,3,2,1>{});
auto t2 = tuple2tuple(t, std::index_sequence<0,2,4>{});
auto t3 = tuple2tuple(t, std::index_sequence<4,3,2,1>{});
```

Can we move one step further to make `std::index_sequence` easier? Yes, by introducing factory `std::make_index_sequence`.

```
auto t0 =   vec2tuple(v, std::make_index_sequence<3>{}); // equivalent to std::index_sequence<0,1,2>
auto t1 =   vec2tuple(v, std::make_index_sequence<4>{}); // equivalent to std::index_sequence<0,1,2,3>
auto t2 = tuple2tuple(t, std::make_index_sequence<3>{});
auto t3 = tuple2tuple(t, std::make_index_sequence<4>{});
```

2. How to implement index sequence and its factory (various factories)?

```
template<std::size_t... NS> struct seq{}; // This is my sequence, just an empty class ...

// *** Factory 1 : Contigous sequence *** //
template<std::size_t N, std::size_t... NS> struct seq_gen          { using type = typename seq_gen<N-1,N-1,NS...>::type; };
template               <std::size_t... NS> struct seq_gen<0, NS...> { using type = seq<NS...>;                              };
template<std::size_t N> using make_seq = typename seq_gen<N>::type; // just an alias

// *** Factory 2: Alternative sequence *** //
template<std::size_t N, std::size_t... NS> struct alt_gen          { using type = typename alt_gen<N-2,N-1,NS...>::type;   };
template               <std::size_t... NS> struct alt_gen<0, NS...> { using type = seq<NS...>;   /* where NS = 1,3,5,... */ };
template               <std::size_t... NS> struct alt_gen<1, NS...> { using type = seq<0,NS...>; /* where NS = 2,4,6,... */ };
template<std::size_t N> using make_alt = typename alt_gen<N>::type; // just an alias
```

Lets consider :

```
seq_gen<10>::type = seq_gen<9,9>::type
                  = seq_gen<8,8,9>::type
                  = seq_gen<7,7,8,9>::type
                  = ...
                  = seq_gen<0,0,1,...,7,8,9>::type
                  = seq<0,1,...,7,8,9>

alt_gen<8>::type  = seq<1,3,5,7>
alt_gen<9>::type  = seq<0,2,4,6,8>
```

We can test our implementation in the same way (using `seq` instead of `std::index_sequence`).

```
template<typename C, std::size_t... NS>
auto vec2tuple(const C& container, const seq<NS...>& dummy)
{
    return std::make_tuple(container[NS]...);
}

template<typename T, std::size_t... NS>
auto tuple2tuple(const T& tuple, const seq<NS...>& dummy)
{
    return std::make_tuple(std::get<NS>(tuple)...);
}

// Testing program then becomes ...
auto t0 =   vec2tuple(v, make_alt<5>{});
auto t1 =   vec2tuple(v, make_alt<6>{});
auto t2 = tuple2tuple(t, make_alt<5>{});
auto t3 = tuple2tuple(t, make_alt<6>{});
```

3. Useful application - in YLib sqlite

This is useful for out-streaming tuples, or constructing SQL queries (like `YLibrary`). The following out-streaming example makes use of two variadic template properties :

- variadic `sizeof...` operator
- variadic fold expression with comma operator `(, ...)`

```
template<typename... ARGS>
std::ostream& operator<<(std::ostream& os, const std::tuple<ARGS...>& tuple)
{
    std::apply([&os] (const ARGS&... args)
    {
        std::size_t n{0};

        os << '[';
        ((os << args << (++n!=sizeof...(ARGS)? "," : "")), ...);
        os << ']';

        // by expanding the above fold-expression, we have multiple increment for n ...
    /*  ((os << arg0 << (++n!=sizeof...(ARGS)? "," : "")),
         (os << arg1 << (++n!=sizeof...(ARGS)? "," : "")),
         (os << arg2 << (++n!=sizeof...(ARGS)? "," : "")),
         ...
         (os << argN << (++n!=sizeof...(ARGS)? "," : "")));

    }, tuple);
    return os;
}
```

G6. Generalized constant expression

1. use of `constexpr` for global function

Generalized constant expression `constexpr` is a declaration, telling compiler that the expression is constant, and is known in compile time so that it can be pre-calculated in compile-time. Unlike template metaprogramming, `constexpr` supports double precalculation.

- `const` means unchanged in runtime (but unknown in compile time)
- `constexpr` means known in compile time (also unchanged in runtime), all literal are literally `constexpr`

Compile time calculation is triggered when 6 conditions are fulfilled, otherwise it becomes runtime calculation :

- declare the function `constexpr`
- declare the input arguments `constexpr` or (`const` and can be deduced to be `constexpr`) *I guess the deduction is liked DAG traversal*
- declare the output variable `constexpr` or (`const` and can be deduced to be `constexpr`)
- the function can only access `constexpr` global variables or invoke `constexpr` global functions.
- the function cannot invoke operator that results in internal state change, such as ++operator
  the function cannot invoke `new delete`, no `try-catch` block, no thread local, as these are all runtime context
- in C++11, the function is one line, which is the return statement
  in C++14, this requirement is relaxed

```
constexpr int fctA(int x, int y, int z) { return 100 * x + 10 * y + z; }
          int fctB(int x, int y, int z) { return 100 * x + 10 * y + z; }

int x = 1; const int cx = 1;
int y = 2; const int cy = 2;

const int n0 = fctA( 1, 2, 3); static_assert(n0 = 123, "fail n0");
const int n1 = fctB( 1, 2, 3); static_assert(n1 = 123, "fail n1"); // assert in compilation : fail n1
const int n2 = fctA( x, y, 3); static_assert(n2 = 123, "fail n2"); // assert in compilation : fail n2
const int n3 = fctA(cx,cy, 3); static_assert(n3 = 123, "fail n3");
      int n4 = fctA( 1, 2, 3); static_assert(n4 = 123, "fail n4"); // assert in compilation : fail n4

// The following testing does not work, as gcc supports VLA variable length array.
int array0[n0]; std::cout << sizeof(array0); // compile ok, print 123*4
int array1[n1]; std::cout << sizeof(array1); // compile ok, print 123*4
int array2[n2]; std::cout << sizeof(array2); // compile ok, print 123*4
int array3[n3]; std::cout << sizeof(array3); // compile ok, print 123*4
int array4[n4]; std::cout << sizeof(array4); // compile ok, print 123*4
```

2. use of `constexpr` for class

Compile time calculation is triggered when 6 conditions are fulfilled, otherwise it becomes runtime calculation :

- declare the class constructor `A::A` and member function `A::fct` as `constexpr`
- declare the input arguments (for both `A::A` and `A::fct`) `constexpr` or (`const` and can be deduced to be `constexpr`)
- declare the output variable (for both `A::A` and `A::fct`) `constexpr` or (`const` and can be deduced to be `constexpr`)
- both `A::A` and `A::fct` can only access `constexpr` global variables or invoke `constexpr` global functions.
- both `A::A` and `A::fct` cannot invoke operator that results in internal state change, such as ++operator
  both `A::A` and `A::fct` cannot invoke `new delete`, no `try-catch` block, no thread local, as these are all runtime context
- in C++11, `A::fct` is one line, which is the return statement
  in C++14, this requirement is relaxed

```
struct A
{
    constexpr A(int x, int y) : mx(x), my(y) {}
    constexpr int fct(int z) const { return 100 * mx + 10 * my + z; }
    const int mx;
    const int my;
};

const A obj0( 1, 2);
const A obj1( x, y);
const A obj2(cx,cy);
      A obj3( 1, 2);

const int m0 = obj0.fct(3); static_assert(m0 ==123, "fail m0");
const int m1 = obj1.fct(3); static_assert(m1 ==123, "fail m1"); // assert in compilation : fail m1
const int m2 = obj2.fct(3); static_assert(m2 ==123, "fail m2");
const int m3 = obj3.fct(3); static_assert(m3 ==123, "fail m3"); // assert in compilation : fail m3
      int m4 = obj0.fct(3); static_assert(m4 ==123, "fail m4"); // assert in compilation : fail m4
```

3. use of `if constexpr`

Compile time condition checking `if constexpr` is introduced in C++17. It makes meta template programming easier, for example :

```
template<int N> constexpr int fibonacci()
{
    if constexpr (N>=2) return fibonacci<N-1>() + fibonacci<N-2>();
    else return N; // for N = 0,1
}
```

It can be used to reimplement `std::get` for `std::tuple` :

```
template<typename T0, typename T1, typename T2> struct my_tuple { T0 m0; T1 m1; T2 m2; };
template<int N, typename TUPLE> auto& get(TUPLE& tuple)
{
    if      constexpr (N==0) return tuple.m0;
    else if constexpr (N==1) return tuple.m1;
    else                     return tuple.m2;
}
```

It can be used to replace complicated *SFINAE* :

```
template<typename T, typename std::enable_if< std::is_target<T>::value, int>::type = 0> fct(const T& x) { implement0(); }
template<typename T, typename std::enable_if<!std::is_target<T>::value, int>::type = 0> fct(const T& x) { implement1(); }

// being simplified as :
template<typename T> void fct(const T& x)
{
    if constexpr (std::is_target<T>::value) implement0();
    else                                    implement1();
}

// in fact it does better than SFINAE to support multiple implementations :
template<typename T> void fct(const T& x)
{
    if      constexpr (std::is_target<T>::value)     implement0();
    else if constexpr (std::is_same<T,typeA>::value) implement1();
    else if constexpr (std::is_container<T>::value)  implement2();
    else if constexpr (std::is_iterable<T>::value)   implement3();
    else                                             implement4();
}
```

4. static `inline` versus static `constexpr`

- `inline` allows violation of One Definition Rule (for functions since C++98, for static variable since C++17)
- `constexpr` is implicitly `inline`
- `static const` can be initialized in header if it is declared `inline`, initialization is done in runtime
- `static const` can be initialized in header if it is declared `constexpr`, initialization is done in compile time

```
          T get_init_value_runtime();
constexpr T get_init_value_compile_time();

class sampleA
{
    inline static const     T s0 = get_init_value_runtime();
           static constexpr T s1 = get_init_value_compile_time();
};
class sampleB
{
    static const     T s0 = get_init_value_runtime();
    static constexpr T s1 = get_init_value_compile_time();
};
const     T sampleB::s0 = get_init_value_runtime();             // still inside header, allow multi-identical defintiion
constexpr T sampleB::s1 = get_init_value_compile_time();        // still inside header, allow multi-identical defintiion
```

5. use of `consteval`

With `constexpr` compiler can choose between compiler-time or runtime calculation depending whether the function is invoked with compile-time known objects and arguments, compiler does not inform us which way it picks, no warning is generated. Sometimes, we need to force the compiler to do compile-time calculation and generate error if it fails to do so, we then need `consteval` to declare an **immediate function**, i.e. function that can only be invoked with compile-time known values.

```
    consteval double area(double r) { return r*r*3.1415; }
    constexpr double a0 = area(1.23);                       // compile time calculation
    const     double a1 = area(1.23);                       // compile time calculation
//  constexpr double a2 = area(x);                          // compile error, area() should be invoked by const-expression
    const     double a3 = area(x);                          // compile error, area() should be invoked by const-expression
```

6. use of `constinit`

`constinit` is used in initialization of static variables. First of all, lets revise the differences between :

- global and local (this is about scope of objects)
- static and automatic (this is about lifetime of objects)

Local variables are variables having finite scope *(accessibility)*, such as function scope, class scope, file scope etc. Global variables are those which have scope extended to a file or across files. Global variable can be declared anywhere in a file, outside all function and class scopes. Global variables are global in a single file by default, unless they are "exported" to other files by `extern`. Static variables are variables having lifetime extended out of their scope, starting from first encounter to program termination. They are not located in call stack, instead they are allocated in BSS segment (in order to achieve the extended lifetime).

- local variables can be static or automatic (automatic by default)
- global variable must be static in nature, but it may NOT necessarily be specifed explicitly as `static` :
- global variable not specfied as `static` can be exported to other files by `extern`
- global variable being specfied as `static` can ***NOT*** be exported to other files by `extern` or results in compile error

```
// *** file0.cpp *** //
       std::uint32_t global_var0 = 100;  // global in current file, can    be exported to other files via extern
static std::uint32_t global_var1 = 200;  // global in current file, CANNOT be exported to other files via extern
extern std::uint32_t global_var2;        // try to import global variable from other files
extern std::uint32_t global_var3;        // try to import global variable from other files
void access_global();
void access_static();

std::cout << global_var0;
std::cout << global_var1;
std::cout << global_var2;
//  std::cout << global_var3; // compile error, cannot import static global
for(std::uint32_t n=0; n!=10; ++n) access_static();

// *** file1.cpp *** //
extern std::uint32_t global_var0;
extern std::uint32_t global_var1;
       std::uint32_t global_var2 = 300;
static std::uint32_t global_var3 = 400;

void access_global()
{
    global_var0 += 10;
//  global_var1 += 10; // compile error, cannot import static global
    global_var2 += 10;
}

void access_static()
{
    static std::uint32_t static_var = 500;
    std::cout << ++static_var;
}
```

The object lifetime is started by invocation of constructor on entering its scope for automatic variable, or on first encounter for static variable. There are numerious initialization methods for automatic variable, please refer to C++ document. For static variables, they are either initialized in compile time or runtime, here are the various initializations for static object :

- compile time initialization by `constexpr` (compile time constant, however it also has to be immutable in its lifetime)
- compile time initialization by `constinit` (compile time constant, no need to be immutable in its lifetime, good for `static`)
- runtime initialization (class like `std::string` cannot be init in compile time, as it has raw pointer)

```
template<typename T> struct square
{
    constexpr explicit square(const T& x) : side(x) {}
//  constexpr T area() const { std::cout << "*"; return side*side; }  // compile error : cannot ostream inside constexpr
//  constexpr T peri() const { std::cout << "."; return side*4; }     // compile error : cannot ostream inside constexpr
    constexpr T area() const { return side * side; }
    constexpr T peri() const { return side * 4;    }
    std::uint32_t side;
};

void increment_square()
{
    static constexpr square<std::uint32_t> sq0(12);                // method 1 : compile time initialization by constexpr
    static constinit square<std::uint32_t> sq1(12);                // method 2 : compile time initialization by constinit
//  static constexpr std::string label0 = std::string("abcdef");   // compile error : cannot init string in compile time
    static const     std::string label1 = std::string("abcdef");   // method 3 : runtime initialization

//  sq0.side *= 2; // compile error : cannot modify constexpr
    sq1.side *= 2; // fine
}
```

Part 2 – Multithreading

A	thread object	2	
B	avoid race condition		
• 4	mutex (futex?)	4225	with various locks
• 4	shared-mutex	2112	with shared-lock
• 2	spinlock	2	with atomic-flag
• 2	call once	2	with once-flag
• 5	singleton	5	with once-flag and double checked lock pattern
C	synchronization mechanism		
•	condition variable	63333	
•	promise-future	44444 + 1	
•	promise-shared-future	x	
•	packaged task	x	
•	async	x	
D	synchronization model		
•	producer consumer model	1	with `std::mutex` and `std::condition_variable`
•	producer consumer model	1	with `std::promise` and `std::future`
•	threadpool *(non template)*	4	with 4 methods
•	io-service	1	with `std::packaged_task` and `std::future`
•	semaphore *(non template)*	4	with 4 relationships among those synchronization primitives
E	other issues		
•	speed comparison	1	
•	divide and conquer	1	
•	thread local storage	1	
•	scheduling, thread model and memory model	3	

Compare mutex in `B1` with …

```
B2. shared_mutex = 2 mutexes + 1 integer
B3. spinlock     = 1 atomic_flag
D5. semaphore    = 1 mutex + 1 condition_variable + 1 integer
```

16 classes	is template?	movability	copyability
`std::mutex`	no	no	no
`std::recursive_mutex`	no	no	no
`std::timed_mutex`	no	no	no
`std::recursive_time_mutex`	no	no	no
`std::shared_mutex`	no	no	no
`std::lock_guard`	`class <T>`	no	no
`std::unique_lock`	`class <T>`	yes	no
`std::shared_lock`	`class <T>`	yes	no
`std::atomic_flag`	no	no	no
`std::once_flag`	no	no	no
`std::condition_variable`	no	no	no
`std::promise`	`class <T>`	yes	no
`std::future`	`class <T>`	yes	no
`std::shared_future`	`class <T>`	yes	yes
`std::packaged_task`	`class <T(X,Y,Z)>`	yes	no
`std::async`	`fct <T(X,Y,Z)>`	yes	no

`Non template classes are all non-movable. Only shared_future is copyable.`

Eventually we will find out that all the following refer to the same thing :

- producer consumer model
- threadpool = producer consumer model of tasks
- async service = producer consumer model of tasks
- sync primitives = semaphore, futex, mutex, condition variable (inter-related to each other), `std::promise` and `std::future`

Please note that async service is different from synchronization primitive :

- async service = instant return control of execution regardless whether the task is done, example : queue in threadpool
- sync primitives = mechanism for time alignment of two threads, example : lock in threadpool

A1. Thread

1. Thread management

`std::thread` manages thread by binding thread resource to `std::thread` object. If a thread is still running when `std::thread` object goes out of scope, the thread is left unmanaged, so we have to ensure a thread to finish its task before the `std::thread` object is destructed. It can be done by blocking call `std::thread::join()`. Useful members for current thread :

- `std::this_thread::get_id()`
- `std::this_thread::sleep_for(std::chrono::seconds(1))` or `sleep_until()`
- `std::this_thread::yield()` informs scheduler to reschedule, allow other threads to proceed, used by spinlock in realtime mode
- `builtin_ia32_pause()` which is a better alternative to `std::this_thread::yield()`, the former is no-operation for a few clock cycles

2. Thread construction

Construction of `std::thread` invokes `std::bind` implicitly. Recall the 4 ways to invoke `std::bind` or construct `std::thread`.

```
struct X {};
struct Y {};
void f(const X&, const Y&){}
struct A
{
    void f(const X&, const Y&){}
    void operator()(const X&, const Y&){}
};

A obj; X x; Y y;
std::thread t0(f, std::cref(x), std::cref(y));                      // 1. global function
std::thread t1(&A::f, std::ref(obj), std::cref(x), std::cref(y));   // 2. member function
std::thread t2(std::ref(obj), std::cref(x), std::cref(y));          // 3. functor
std::thread t3([](const X&, const Y&) {}, std::cref(x), std::cref(y));   // 4. lambda
t0.join(); t1.join(); t2.join(); t3.join();
```

B1. Mutex

1. Critical session

[1]Critical session is piece of code which forbids concurrency access or equivalently requires mutual exclusive access, otherwise it can lead to unexpected behaviour, critical session can be protected by various locks such as ***mutex lock*** and ***spinlock***. Both are blocking mechanism. [2]Mutex involves sleeping and waking of threads, therefore saving *CPU* load at the expense of higher latency. [3]Spinlock involves continuous polling, which wastes computation power for the sake of lower latency. [4]If there are *multiple physical cores* and if *critical session is short*, spinlock is a better option.

2. What is a mutex?

- Mutex is a bistate variable, offering atomic set/reset functions.
  - Mutex is governed by ownership (as opposed to semaphore), i.e. the thread locked a mutex is responsible for releasing it later.
- Mutex member functions :

`mutex`	`lock()`	`try_lock()`	`unlock()`		
`recursive_mutex`	`lock()`	`try_lock()`	`unlock()`		
`timed_mutex`	`lock()`	`try_lock()`	`unlock()`	`try_lock_for(duration)`	`try_lock_until(time_point)`
`recursive_timed_mutex`	`lock()`	`try_lock()`	`unlock()`	`try_lock_for(duration)`	`try_lock_until(time_point)`

3. What is a lock?

- Lock is manager of ownership (i.e. responsibility to unlock)
  - `std::lock_guard<T>` manages lock with *RAII*
  - `std::unique_lock<T>` manages lock with movable ownership, which allows us to …
  - `std::unique_lock<T>` works with `std::lock`, which allows `std::defer_lock` and `std::adopt_lock`
  - `std::unique_lock<T>` works with `std::condition_variable`, which transfer lock ownership on waiting
- Lock member functions forward implementation to corresponding mutex member functions :

`std::lock_guard<T>`	`lock()`	`unlock()`			
`std::unique_lock<T>`	`lock()`	`try_lock()`	`unlock()`	`try_lock_for(duration)`	`try_lock_until(time_point)`

```
struct T
{
    void critical_session()
    {
        std::lock_guard<std::mutex> lock(mutex);
        access(shared_resources);
    }

    R shared_resources;
    std::mutex mutex; // T is noncopyable due to std::mutex
};
```

4. What is Deadlock?

When we need to lock multiple resources, like money transfer between two accounts, there are risks of deadlock.

```
struct book
{
    void transfer(const std::string& src_id, const std::string& dst_id, double amount)
    {
        std::lock_guard<std::mutex> lock0(accounts[src_id].mutex);
        std::lock_guard<std::mutex> lock1(accounts[dst_id].mutex);
        accounts[src_id].balance -= amount;
        accounts[dst_id].balance += amount;
    }
    std::map<std::string, account> accounts; // suppose each account has a mutex
};
```

Don't do the following, it makes multithreading useless!

```
struct book
{
    void transfer(const std::string& src_id, const std::string& dst_id, double amount)
    {
        std::lock_guard<std::mutex> lock(mutex);
        accounts[src_id].balance -= amount;
        accounts[dst_id].balance += amount;
    }
    std::mutex mutex;
    std::map<std::string, account> accounts; // suppose no mutex inside account
};
```

Deadlock can be avoided by :

- prioritize resources and lock them in order or
- use variadic template `std::lock`, which either atomically locks all requested mutexes or nothing
- locking happens inside `std::lock`, yet we still need to construct `std::lock_guard` for ownership management
- `std::lock` works with any class that offers `lock()`, `try_lock()` and `unlock()`, so it works with `std::mutex` and `std::unique_lock`
- `std::lock` is exception safe when one of the requested mutexes throws, already-locked mutexes will be released

```
// Method 1 : prioritizing resources
void accounts::transfer(const std::string& src_id, const std::string& dst_id, double amount)
{
    std::lock_guard<std::mutex> lock0(accounts[std::min(src_id, dst_id)].mutex);
    std::lock_guard<std::mutex> lock1(accounts[std::max(src_id, dst_id)].mutex);
    accounts[src_id].balance -= amount;
    accounts[dst_id].balance += amount;
}

// Method 2 : using std::lock with std::lock_guard
void accounts::transfer(const std::string& src_id, const std::string& dst_id, double amount)
{
    std::lock(accounts[src_id].mutex, accounts[dst_id].mutex); // std::lock two std::mutex
    std::unique_lock<std::mutex> lock0(accounts[src_id].mutex, std::adopt_lock);
    std::unique_lock<std::mutex> lock1(accounts[dst_id].mutex, std::adopt_lock);
    accounts[src_id].balance -= amount;
    accounts[dst_id].balance += amount;
}

// Method 3 : using std::lock with std::unique_lock
void accounts::transfer(const std::string& src_id, const std::string& dst_id, double amount)
{
    std::unique_lock<std::mutex> lock0(accounts[src_id].mutex, std::defer_lock);
    std::unique_lock<std::mutex> lock1(accounts[dst_id].mutex, std::defer_lock);
    std::lock(lock0, lock1); // std::lock two std::unique_lock
    accounts[src_id].balance -= amount;
    accounts[dst_id].balance += amount;
}
```

DAG representation of a deadlock

Suppose we model threads and resources by a graph $G = \{V_T, V_R, E\}$:

- V_T set of thread vertices
- V_R set of resource vertices
- E set of edges (each edge is a lock, linking a thread and a resource it attempts to lock)

There will be deadlock when two conditions exist *(1)* cycle is formed and *(2)* threads in lock resources in unorganised manner.

B2. Shared mutex

1. multi-readers-single-writer model

1For multi-readers-single-writer model, readers do not consume (modify) any shared resource, using mutex may overkill as it leads to lock-contention among readers. In fact, we do not need to block readers while other readers are reading, it is better to use shared mutex (also known as reader-writer lock).

2It offers :

- exclusive access for single writer, when it is writing
- shared access for multiple readers, when no writer is writing

2. What is a shared mutex?

```
// functions for writer (producer)
std::shared_mutex::lock()
std::shared_mutex::try_lock()
std::shared_mutex::unlock()
// functions for reader (observer)
std::shared_mutex::lock_shared()
std::shared_mutex::try_lock_shared()
std::shared_mutex::unlock_shared()
```

3. What are corresponding locks?

- for writer, construct `std::lock_guard` or `std::unique_lock`, to invoke `shared_mutex::lock()` and `shared_mutex::unlock()`
- for reader, construct `std::shared_lock`, to invoke `shared_mutex::lock_shared()` and `shared_mutex::unlock_shared()`

```
struct mktdata
{
    void add_tick(TICK&& tick)
    {
        std::lock_guard<std::shared_mutex> lock(mutex);
        ticks.push_back(std::move(tick));
    }

    const TICK& latest_tick() const
    {
        std::shared_lock<std::shared_mutex> lock(mutex);
        return ticks.back();
    }

    mutable std::shared_mutex mutex;
    std::vector<TICK> ticks;
};
```

4. Implementation

1Shared mutex can be implemented using two mutexes : one for global protection and one for protecting reader-count.

```
struct my_shared_mutex // Not verified in MSVS yet
{
    void lock()       { mutex_global.  lock();    }
    void unlock()     { mutex_global.unlock();    }

    void lock_shared()
    {
        std::lock_guard<std::mutex> lock(mutex_reader); ++reader;
        if (reader==1) lock();
    }
    void unlock_shared()
    {
        std::lock_guard<std::mutex> lock(mutex_reader); --reader;
        if (reader==0) unlock();
    }

    int reader = 0;
    std::mutex mutex_global;
    std::mutex mutex_reader;
};
```

2Acquiring a lock from a `shared_mutex` is more costly than a `mutex`. Use it only when contention among readers is serious.

when reading is …	short	long
rare	`spinlock/atomic`	`mutex`
frequent	`spinlock/atomic`	`shared_mutex`

B3. Spinlock and Event

Spinlock is a continuous polling of an atomic flag. When the critical session is short, spinlock is preferred to mutex in order to offer low latency. Spinlock can be implemented with an atomic flag `std::atomic_flag`, which is the only data type in STL that ensures lock-free operations across all CPU architectures. *[Recall that atomic doesn't guarantee lockfree.]*.

The atomic flag takes 2 possible values :

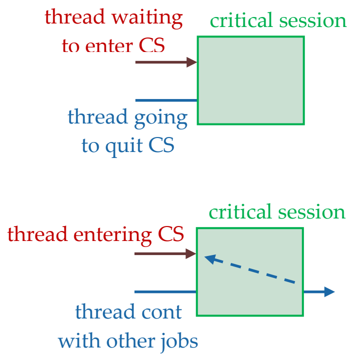
- `ATOMIC_FLAG_INIT = clear = false = unlocked`
- `set =  true =   locked`

```
// Implementation using STL
struct spinlock
{
     void   lock() { while(flag.test_and_set(std::memory_order_acquire)) std::this_thread::yield(); }
     void unlock() { flag.clear(std::memory_order_release); }

     std::atomic_flag flag = ATOMIC_FLAG_INIT;
};

// Implementation using pthread
struct spinlock_p
{
    spinlock_p()  { pthread_spin_init   (&impl, PTHREAD_PROCESS_PRIVATE); }
   ~spinlock_p()  { pthread_spin_destroy(&impl);                          }
    void   lock() { pthread_spin_lock   (&impl);                          }
    void unlock() { pthread_spin_unlock (&impl);                          }

    pthread_spinlock_t impl;
};
```

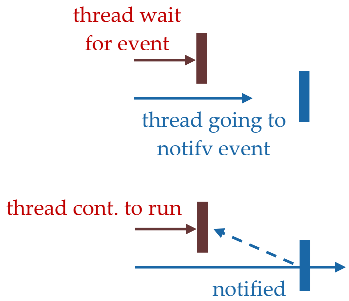


Event object is similar, but it differs from spinlock :

- event suspends threads before a line of code, paused threads are notified by other non-racing threads
- spinlock suspends threads before a block of code, paused threads are notified by other racing threads (winner indeed)

```
// Implementation using STL
struct event
{
    void   wait() { while(!flag.load(std::memory_order_acquire)) std::this_thread::yield(); }
    void notify() { flag.store( true,std::memory_order_release); }
    void  reset() { flag.store(false,std::memory_order_release); }

    std::atomic<bool> flag = false;
};
```



B4. Call once and Once flag

Class `std::once_flag` and functon `std::call_once` can be used together to wrap any function, such that it is invoked only once in multi threading scenario. This is useful for singleton and lazy initialization, the latter means delaying expensive initialization of an object until the first time it is accessed. Don't confuse `std::once_flag` with `std::atomic_flag` (though both are flags).

Please note that :

- `std::once_flag` should be default-initialized
- `std::once_flag` should not be checked with `if` n
- `std::once_flag` offers no member function, just invoke `std::call_once`

```
struct algo
{
     void init();

     void calculate0() { std::call_once(flag, std::bind(&algo::init, this)); do_something0(); }
     void calculate1() { std::call_once(flag, std::bind(&algo::init, this)); do_something1(); }
     void calculate2() { std::call_once(flag, std::bind(&algo::init, this)); do_something2(); }

     std::once_flag flag;
     unsigned long count = 0;
};

algo x;
std::thread t0(&algo::calculate0, std::ref(x));
std::thread t1(&algo::calculate1, std::ref(x));
std::thread t2(&algo::calculate2, std::ref(x));
t0.join; t1.join; t2.join;
```

Once flag is one of the implementations of singleton, lets see in next section.

B5. Singleton

Several *multithread-safe* implementations for singleton :

- singleton with static local variable in static function (Scott Meyers) — *however if we cant declare static local before c++11 …*
- singleton with pointer + `std::once_flag` — *however if we cant use `std::once_flag` …*
- singleton with pointer + mutex — *however it is slow …*
- singleton with pointer + mutex + *Double Checking Locked Pattern (DCLP)* — *there is still problem …*
- singleton with pointer + mutex + *Double Checking Locked Pattern (DCLP)* + atomic load

Approach 1

Lets start with a naïve implementation of singleton in C++11, which multithread-safe and doesn't need *DCLP* nor `std::once_flag`. We can declare `static` variable inside `static` member function. This is multithread safe, as the initialization of `static` local is guaranteed to be atomic, hence only one instance is created. This singleton pattern is attributed to Scott Meyers.

```
class singleton // Verified in MSVS
{
    singleton() = default;
    ~singleton() = default;
    singleton(const singleton&) = delete;
    singleton& operator=(const singleton&) = delete;

public:
    static singleton& get_instance()
    {
        static singleton instance;
        return instance;
    }

    // single instance of each member, shared among all threads
    T0 x;
    T1 y;
    T2 z;
};
```

Approach 2

If we cannot declare static local variable inside static function, we have to use `std::once_flag`.

```
class singleton // Verified in MSVS
{
    singleton() = default;
    ~singleton() { if (p) delete p; };
    singleton(const singleton&) = delete;
    singleton& operator=(const singleton&) = delete;

    static singleton* p;
    static std::once_flag flag;

public:
    static singleton& get_instance()
    {
        std::call_once(flag, [this](){ p = new singleton; });
        return *p;
    }

    T0 x;
    T1 y;
    T2 z;
};

// Don't forget to initialize static variables
singleton* singleton::p{nullptr};
std::once_flag singleton::flag;
```

Approach 3

However we cannot achieve such a neat implementation in C++98. In old days people used to the following mutex implementation instead. Why mutex is needed? Consider multithreads entering `get_instance()`, and if some survive nullity checking, it will result in multiple instances, violating singleton.

```
class singleton // Verified in MSVS
{
    singleton() = default;
    ~singleton() { if (p) delete p; };
    singleton(const singleton&) = delete;
    singleton& operator=(const singleton&) = delete;

    static singleton* p;
    static std::mutex m;
```

```
public:
    static singleton& get_instance()
    {
        std::lock_guard<std::mutex> lock(m);
        if (!p) p = new singleton;
        return *p;
    }

    T0 x;
    T1 y;
    T2 z;
};

// Don't forget to initialize static variables
singleton* singleton::p{nullptr};
std::mutex singleton::m;
```

Approach 4 (failed)

Race condition is solved. As mutex lock is slow, each subsequent `get_instance()` call is expensive, thus we add preliminary checking so as to eliminate the chance of locking a mutex. This is called the double checking locked pattern *(there are two IF checking now)*.

```
singleton& singleton::get_instance()
{
    if (!p) // first checking
    {
        std::lock_guard<std::mutex> lock(m);
        if (!p) p = new singleton; // second checking
    }
    return *p;
}
```

Approach 5

As the first nullity checking is non-atomic, whereas the assignment to `p` is also non-atomic, it is possible that when a thread is in the middle of `p` assignment, while another thread finds that `p` is non-null and tries to return the incomplete instance. It can be solved by making `p` atomic, we have to store and load `p` using atomic functions.

```
class singleton // Verified in MSVS
{
    singleton() = default;
    ~singleton() { if (p) delete p; };
    singleton(const singleton&) = delete;
    singleton& operator=(const singleton&) = delete;

    static std::atomic<singleton*> p;
    static std::mutex m;

public:
    static singleton& get_instance()
    {
        singleton* local_p = std::atomic_load_explicit(&p, std::memory_order_acquire);
        if (!local_p)
        {
            std::lock_guard<std::mutex> lock(m);
            if (!local_p)
            {
                local_p = new singleton;
                std::atomic_store_explicit(&p, local_p, std::memory_order_release);
            }
        }
        return *local_p;
    }

    T0 x;
    T1 y;
    T2 z;
};

std::atomic<singleton*> singleton::p{nullptr};
std::mutex singleton::m;
```

C. Synchronization mechanism between producer and consumer

In multithread programming, different threads should be synchronized ***(i.e. aligned in time)***, so as to avoid unexpected bahaviours. Synchronization usually involves blocking one thread which is runing too fast until it is notified by another thread when things are ready. Here are a list of various synchronization mechanisms :

- condition variable
- promise and future
- promise and shared future
- packaged task and future
- async and future

C1. Condition variable

1. There are six components in a producer consumer model

- product `T` = `class`
- production function = `mapping : X,Y,Z -> T`
- consumption function = `mapping : T -> void`
- producer = `mapping : empty queue -> filled queue` by repeated invocation of `production` and `queue.push`
- consumer = `mapping : filled queue -> empty queue` by repeated invocation of `queue.pop` and `consumption`
- common resource = `pc_queue<T>` with `push` and `pop`

or common resource = `promise<T>` with `set` and `future<T>` with `get`

2. There are three objects protecting the common resource

- `mutex` to protect critical session against race condition, so that :
- producer invokes `production` outside critical session and `queue.push` inside critical session
- consumer invokes `queue.pop` inside critical session and `consumption` outside critical session
- Condition variable *push-able* protects producer from pushing full queue.
- Condition variable *pop-able* protects consumer from poping empty queue.

3. There are three main steps

- producer and consumer race for the mutex :
- the faster thread gets the lock and enters critical session
- the slower thread is blocked *release-lock-and-wait-cv step*
- if the faster thread encounters unfavourable condition that it cannot proceed : ⇩
- it releases the lock, allowing slower thread to proceed and fix the situation, and waits on corresponding condition variable
- when producer trying to push to full queue, it waits on *push-able* condition variable
- when consumer trying to pop from empty queue, it waits on *pop-able* condition variable
- the slower thread proceeds and fixes the condition, then notifies the waiting thread

4. Some remarks

- for thread calling `cond_var.wait(lock)` we should use movable `std::unique_lock`
- for thread calling `cond_var.notify_one()` we can use non-movable `std::lock_guard`
- `cond_var.wait(lock)` does 3 things :
- implicitly releases lock
- wait to be notified
- implicitly requests lock again

5. Asymmetric design (protection against popping empty queue only)

producer

```
T0 = produce();
{
    std::lock_guard<std::mutex> lock(mutex); // blocked


    queue.push(T0);
    cond_var.notify_one();

}   // lock.~lock_guard();
```

consumer

```
{
    std::unique_lock<std::mutex> lock(mutex); // faster thread

    if    (queue.empty()) cond_var.wait(lock);
//  while (queue.empty()) cond_var.wait(lock);          // solution 1
//  cond_var.wait(lock, [](){return !queue.empty();}); // solution 2

    T0 = queue.pop();
}

consume(T0);
```

Two problems with condition variable :

- lost wake up
- spurious (fake) wake up

Lost wakeup and spurious wakeup

Lost wakeup happens when faster thread holding the lock is in the *release-lock-and-wait-cv* step. If this step is non-atomic, the slower thread may notify in between *release-lock* and *wait-cv*, then this notification is lost forever, the faster thread which is also the waiting thread have to wait for next notification. The solution is to put *release-lock* and *wait-cv* into a single atomic operation, this is why we need to pass current lock to conditional variable when we invoke `cond_var.wait(lock)`.

Spurious wakeup happens when faster thread waiting on condition variable, being waken by a notification, but once it re-acquires the lock, it may find that it still encounters the unfavourable condition and it cannot proceed. This is probably because of the racing condition, other running threads probably the one that sent notification, win the race. Therefore it is the responsibility of the waken thread to re-check the condition after being waken up. There are two solutions :

- replace `if (queue.empty()) cond_var.wait(lock)` by while loop `while (queue.empty()) cond_var.wait(lock)`
- replace `if (queue.empty()) cond_var.wait(lock)` by waiting with predicate where `wait` is blocked until predicate returns true `cond_var.wait(lock, [](){ return !queue.empty(); })`

C2-5. Promise, future, shared future, packaged task and async

```
std::promise<T>              + std::future<T>
std::promise<T>              + std::shared_future<T>
std::packaged_task<T(X,Y,Z)> + std::future<T>
std::async<T(X,Y,Z)>         + std::future<T>
```

1. What is promise and future?

- All these pairs are ***one-off*** synchronization between producer and consumer (i.e. no reuse), or regarded as `queue` with size 1.
- Producer owns a `std::promise<T>` which is an obligation to produce an item of type `T`.
- Consumer owns a `std::future<T>` which is a right to consume an item of type `T`.
- Multiple consumers own copyable `std::shared_future<T>` which allows concurrent consumption.
- For speed consideration, we should either make `T` movable or pass `T` by `std::promise<T&>` to `std::future<T&>`.

2. Use of promise and future involves 4 steps

- instantiate a `std::promise<T>` object
- instantiate a `std::future<T>` object from the `std::promise<T>` object
- consumer thread invokes `std::future<T>::get()` and consumption-function, it is blocked until production is done
- producer thread invokes production-function and `std::promise<T>::set()`

3. Some remarks

- `std::promise<T>::set()` is analogous to `queue<T>::push()`
- `std::future<T>::get()` is analogous to `queue<T>::pop()`
- if `std::promise<T>::set()` or `std::future<T>::get()` is called twice, it crashes
- if `std::promise<T>` is destructed without setting value, calling `std::future<T>::get()` will throw exception

4. Packaged task and async

- `std::packaged_task<T(X,Y,Z)>` merges production `T(X,Y,Z)` and `std::promise<T>`
- `std::async<T(X,Y,Z)>` merges production `T(X,Y,Z)` and `std::promise<T>` instantly invoke a new thread
- `std::packaged_task<T(X,Y,Z)>` can be considered a functor that performs `std::bind()` and returns `std::future<T>`
- `std::async<T(X,Y,Z)>` can be considered a function that performs `std::thread()` and returns `std::future<T>`

5. Comparison

	`promise<T>`	`packaged_task<T(X,Y,Z)>`	`async<T(X,Y,Z)>`
what	template class `<T>`	template class `<T(X,Y,Z)>`	template function `<T(X,Y,Z)>`
construction	default init	direct init with `production`	direct init with `production` and `x,y,z`
get future	`auto f = p.get_future();`	`auto f = ptask.get_future();`	`auto f = std::async(...);`
invoke	run `p.set(production(x,y,z))`	run functor `ptask(x,y,z)`	run immediately

6. Demonstration

Lets define movable product `T`, its production function and consumption function.

```
struct T { X x; Y y; Z z; };

T    production(const X& x,const Y& y,const Z& z);
void consumption        (const T& t)              {                std::cout << "\nconsumed " << t.x << t.y << t.z; }
void consumption_partial(const T& t, int part)    { if (part==0)  std::cout << "\nconsumed " << t.x;
                                                     if (part==1)  std::cout << "\nconsumed " << t.y;
                                                     if (part==2)  std::cout << "\nconsumed " << t.z; }
```

Here are producers and consumers. There is no need to pass non-copyable `promise` and `future` by rvalue reference, as no assignment of `promise` nor `future` inside producers and consumers.

```
void producer(pc_queue<T>& queue)
{
    for(int n=0; n!=100; ++n)
    {
        T t = production(rand_X(),rand_Y(),rand_Z());
        queue.push(std::move(t));
    }
}

void consumer(pc_queue<T>& queue)
{
    for(int n=0; n!=100; ++n)
    {
        T t = queue.pop();
        consumption(t);
    }
}
```

```
void producer(std::promise<T>& promise)
{
    T t = production(rand_X(),rand_Y(),rand_Z());
    promise.set_value(std::move(t));
}

void consumer(std::future<T>& future)
{
    T t = future.get();
    consumption(t);
}

void consumer(std::shared_future<T>& sfuture, int part)
{
    T t = sfuture.get();
    partial_consumption(t, part);
}
```

Main program

With `std::launch::deferred` no thread is spawned, production function is deferred until consumer thread invokes `future<T>::get()`, in this case, both production and consumption are run by the same thread.

```
// [Method 1]
pc_queue<T> queue;
std::thread t0(producer, std::ref(queue));
std::thread t1(consumer, std::ref(queue));
t0.join(); t1.join();

// [Method 2]
std::promise<T> promise;
std::future<T>  future = promise.get_future();
std::thread t0(producer, std::ref(promise));
std::thread t1(consumer, std::ref(future));
t0.join(); t1.join();

// [Method 3]
std::promise<T> promise;
std::shared_future<T> sfuture = promise.get_future();
std::thread t0(producer, std::ref(promise));
std::thread t1(consumer, std::ref(sfuture), 0);
std::thread t2(consumer, std::ref(sfuture), 1);
std::thread t3(consumer, std::ref(sfuture), 2);
t0.join(); t1.join(); t2.join(); t3.join();

// [raw material]
auto x = rand_X(); auto y = rand_Y(); auto z = rand_Z();

// [Method 4]
std::packaged_task<T(const X& x,const Y& y,const Z& z)> pack_task(production);
std::future<T> future = pack_task.get_future();
std::thread t(std::ref(pack_task), std::cref(x), std::cref(y), std::cref(z));
consumer(future);
t.join();

// [Method 5]
std::future<T> future = std::async(production, std::cref(x), std::cref(y), std::cref(z));
consumer(future);

// [Method 5 with deferred execution]
std::future<T> future = std::async(std::launch::deferred, production, std::cref(x), std::cref(y), std::cref(z));
consumer(future);
```

Future and promise are just publication pattern. Lets implement our own version future and promise.

```
template<typename T>
struct future
{
    future(const T& x, const std::atomic<bool>& f) : publication(x), flag(f) {}

    const T& get() const
    {
        while(!flag.load(std::memory_order_acquire));
        return publication;
    }

    const T& publication;
    const std::atomic<bool>& flag;
};

template<typename T>
struct promise
{
    promise() : flag(false){}

    future<T> get_future() const
    {
        return { publication, flag };
    }

    void set(const T& x)
    {
        publication = x;
        flag.store(true, std::memory_order_release);
    }

    T publication;
    std::atomic<bool> flag;
};

// *** Successfully tested in gcc *** //
promise<std::uint32_t> p;
auto f = p.get_future();

std::thread t0([&]()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    p.set(123);
});

std::thread t1([&]()
{
    std::cout << "\nwait .... " << std::flush;
    std::cout << "I got " << f.get() << std::flush;
});

t0.join();
t1.join();
```

D1. Producer-consumer using mutex and condition variable

Modulus can be efficiently implemented as a bitwise AND as `n % size = n & mask` where `size = 2^N` and `mask = size-1`.

```
template<typename T> struct pc_queue
{
    void push(T&& item)
    {
        {
            std::unique_lock<std::mutex> lock(mutex);
        //  while (is_full()) cv_pushable.wait(lock);     // cv_pushable.wait(lock, [](){return !is_full();});

            items[next_push & mask] = std::move(item); ++next_push;

        }
        cv_popable.notify_one();
    }

    T pop()
    {
        {
            std::unique_lock<std::mutex> lock(mutex);
            while (is_empty()) cv_popable.wait(lock);     // cv_popable.wait(lock, [](){return !is_empty();});

            T item = std::move(items[next_pop & size]); ++next_pop;

        }
    //  cv_pushable.notify_one();
        return item;
    }

    bool is_empty() const { return next_push == next_pop;        }
    bool is_full()  const { return next_push == next_pop + size; }

    // 5 members for indexing
    static const unsigned short size = 1024;
    static const unsigned short mask = size-1;
    unsigned short next_push = 0;
    unsigned short next_pop  = 0;
    T items[size];

    // 3 members for concurrency
    std::mutex mutex;
    std::condition_variable cv_pushable;
    std::condition_variable cv_popable;
};
```

D2. Producer-consumer using promise and future

Main difference between `pc_model` and `pc_model2` is that, the latter replaces the array of `T` by array of `std::promise<T>` and `std::future<T>`, no explicit mutex nor condition variable is needed. However each `promise-future` pair can be used once only (i.e. no reuse).

```
template<typename T> struct pc_queue2
{
    pc_queue2()
    {
        for(int n=0; n!=size; ++n)
        {
            std::promise<T> promise;
            std::future<T> future = promise.get_future();
            promises.push_back(std::move(promise));
            futures.push_back(std::move(future));
        }
    }

    void push(T&& item)
    {
        auto n = next_push.fetch_add(1); // fetched value is previous value
        promises[n].set_value(std::move(item));
    }

    T pop()
    {
        auto n = next_pop.fetch_add(1); // fetched value is previous value
        return futures[n].get();
    }

    static const unsigned short size = 1024;
//  static const unsigned short mask = size-1;
    std::atomic<unsigned short>  next_push = 0;
    std::atomic<unsigned short>  next_pop  = 0;
    std::vector<std::promise<T>> promises;
    std::vector<std::future<T>>  futures;
};
```

D3. Threadpool for async programming

Threadpool is useful to implement async programming. There are 4 different implementations for threadpool :

- resource friendly, allow context switching, queue protected by mutex and condition var — example `cubquant::threadpool`
- resource friendly, allow context switching, queue protected by counting semaphore — example `YLib::threadpool`
- busy waiting, pinning affinity, with spinlocked task queue — example `YLib::EventQueue`
- busy waiting, pinning affinity, with lockfree queue — example `YLib::lockfree_mpmcq`

The main difference is that :

- for the former two methods, when a thread does not have a task to do, it goes to sleep *(good for num of thread > num of core case)*
- for the latter two methods, when a thread does not have a task to do, it keeps polling the queue *(good for one core per thread)*
- the former two methods maximise throughput
- the latter two methods minimise latency
- all 4 methods adopt preemptive scheduling (the latter two try to stop the preemption by pinning affinity)
- we can replace `std::function` task by `std::coroutine_handle` to adopt cooperative scheduling, please see C++20 doc

Method 1 - Using mutex and condition variable (cubquant)

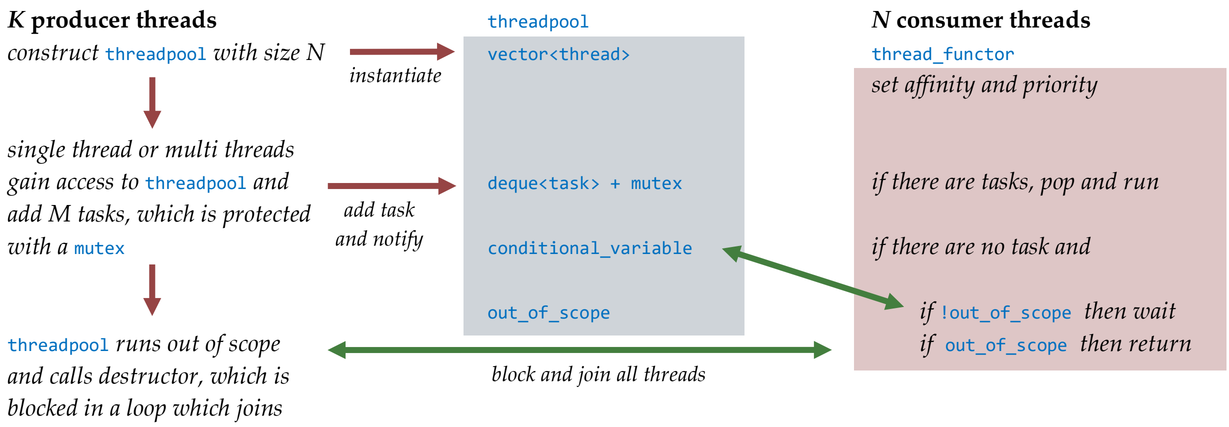
This implementation is copied from "Jakob's Devlog".

Threadpool contains *(1)* a vector of threads *(2)* a deque of tasks (a queue is good enough, yet we may need to iterate through it, that is why a deque) and finally *(3)* combo of synchronization classes, including a conditional variable which must work together with a mutex plus a boolean flag which is a on-off switch to terminate the threadpool. Each task is simply a nullary function which returns void, `std::function<void()>`. Since all functions can be binded to `std::function<void()>`, no template class is needed for this threadpool implementation. ***[I attempted to make it a template class in Lighthouse interview, this is a mistake!]***

Each thread in the threadpool runs a thread functor, it is basically is a while loop that keeps popping tasks from the task deque and execute them as long as the deque contains some tasks. However when the deque becomes empty, the functor will then wait on the conditional variable when the `out_of_scope` boolean flag in the threadpool is false, the functor will return whenever the `out_of_scope` is true, thus the corresponding thread will be destructed. Is `out_of_scope` necessary for switching off the threadpool? Can we simply terminate it as task deque becomes empty? No, thread functors start running inside the construction of threadpool when there is no task, threads will then wait to be notified until new tasks are added, so there is no way to terminate the threadpool unless we have an explicit on-off switch `out_of_scope`. Few more remarks regarding to functor *(1)* at the beginning of functor, we set thread affinity and priority to avoid context switching *(2)* as thread functor has to access task deque and mutex, threadpool has to grant friendship to functor,while functor has a reference to the threadpool *(3)* it is easy to confuse functor with task, the former is agenda for threads, which loops and executes the latter, i.e. task.

Threadpool is considered as multi-producers multi-consumers model which share a deque of tasks as common resource. The deque is protected by a mutex whenever push or pop is invoked. Multiple thread functors (run in different threads) are consumers, when they find the deque empty (hence nothing to consume), instead of keep polling, they wait on a conditional variable, a mutex should be passed at the same time, so as to allow producers to enter the critical section in order to produce. Thats why conditional variable must work with a mutex.

Finally lets go through the threadpool itself. On construction, it instantiates a predetermined number of threads, each runs a thread functor. On destruction, set `out_of_scope` true, notify all threads waiting on conditional variable, then join threads. The last member function is production function for task, it simply locks the mutex, push new task and notify one waiting consumer. Suppose there are N threads and M tasks, then there will be N functors (consumers), racing to consumes the M tasks.



Both green bidirectional arrows are synchronization mechanism.

Implementation 1a - Two remarks and Five bugs !!!

```
class threadpool_condvar
{
public:
    threadpool_condvar(size_t num_threads) : out_of_scope(false)
    {
        for(size_t n=0; n<num_threads; ++n) threads.push_back(std::thread(&thread_pool::fct, this, n));
    }
    ~threadpool_condvar() { stop(); }

    void stop()
    {
        out_of_scope = true;
        condvar.notify_all();
        for(std::thread& thread : threads) thread.join();                    // Bug 4
    }

    void add_task(const std::function<void()>& functor)                      // Producer of tasks
    {
        {
            std::lock_guard<std::mutex> lock(mutex);
            tasks.push_back(functor);
        }
        condvar.notify_one();
    }

    void fct(int thread_affinity)                                            // Consumer of tasks
    {
        set_this_thread_affinity(affinity);
        set_this_thread_priority(THREAD_PRIORITY_HIGHEST);

        // **** 1st loop **** //
    //  while(!out_of_scope || !task.empty())                                // Remark 1
        while(!out_of_scope)
        {
            std::function<void()> task;
            {
                std::unique_lock<std::mutex> lock(pool.mutex);
            //  while(tasks.empty()) condvar.wait(lock);                     // Remark 2
                condvar.wait(lock, [](){ return !tasks.empty(); });          // Bug 1
                task = std::move(tasks.front());                             // Bug 2
                tasks.pop();
            }
            task();                                                          // Bug 5
        }

        // **** 2nd loop **** //
        while(!task.empty())                                                 // Bug 3
        {
            std::function<void()> task;
            {
                std::lock_guard<std::mutex> lock(pool.mutex);
                task = std::move(tasks.front());
                tasks.pop();
            }
            task();
        }
    }

private:
    // *** Threads *** //
    std::vector<std::thread> threads;

    // *** Tasks *** //
    std::queue<std::function<void()>> tasks;

    // *** Synchronization *** //
    mutable std::mutex mutex;
    std::condition_variable cond_var;
    std::atomic<bool> out_of_scope;
};
```

Remark 1 : Decouple out-of-scope checking and empty-task checking

Initially, I used to implement threadpool with single while loop, which checks both `out_of_scope` and `task.empty()`. However this will result in *(1)* redundant checking of `task.empty()` in each loop, which involves slow lock and unlock, *(2)* potential missing notification after `out_of_scope` is set true resulting in blocked threads and deadlock. By decoupling it into two while loops, the first one can focus on live production looping, while the second one can focus on clearing tasks before quiting. There is no `condvar.wait()` in the second loop, hence no missing notification can happen. Besides, as latency is not a necessity on clearing tasks when quiting, all remaining tasks can be invoked by single thread.

Remark 2 : Replace explicit while loop with a predicate in condition variable

Just a better practice. Predicate returns true to continue, returns false to wait.

However there are 5 bugs in the above implementation :

- On stopping the pool, the last notification is emitted ...
- `[Bug 1]` thread may get stuck in condition variable, as predicate returns false when task queue is empty on quiting
- `[Bug 2]` thread got notified may then pop from an empty task queue, which leads to a crash
- `[Bug 3]` threads are racing to pop the task queue, some threads may pop empty queue
- `[Bug 4]` thread may join twice, once in `stop()` and once in destructor, which leads to a crash
- `[Bug 5]` task may throw exception, which is not handled

Implementation 1b - Four bugs fixed

Constructor / destructor are kept unchanged, all data members are kept unchanged. This new implementation is tested with under no exception case, main-thread exception case and task exception case, all work without wakeup-miss, regardless of add-task rate.

```
void stop()
{
    out_of_scope.store(true);
    condvar.notify_all();
    for(auto& x:threads)
    {
        // BUG 4 : need to check joinable to avoid multi-join, otherwise it crashes
        if (x.joinable()) x.join();
    }
}

void add_task(const std::function<void()>& task) // This function is unchanged.
{
    {
        std::lock_guard<std::mutex> lock(mutex);
        tasks.push(task);
    }
    condvar.notify_one();
}

void fct(std::uint32_t id)
{
    // set affinity here (skipped for simplicity)
    // set priority here (skipped for simplicity)

    try // BUG 5 : need to handle exception thrown from task
    {
        // *** 1st loop *** //
        while(!out_of_scope.load())
        {
            std::function<void()> task;
            {
                std::unique_lock<std::mutex> lock(mutex);
                condvar.wait(lock, [this]()
                {
                    // BUG 1 : add out_of_scope to avoid wakeup-miss on termination
                    return !tasks.empty() || out_of_scope.load();
                });

                if (out_of_scope.load()) break; // BUG2 : threads woke up by notify_all() may pop an empty queue
                task = std::move(tasks.front());
                tasks.pop();
            }
            task();
        }

        // *** 2nd loop *** //
        {
            std::lock_guard<std::mutex> lock(mutex); // BUG 3 : only one thread is responsible for clearing queue
            while(!tasks.empty())
            {
                std::function<void()> task;
                {
                    task = std::move(tasks.front());
                    tasks.pop();
                }
                task();
            }
        }
    }
    catch(std::exception& e)
    {
        std::cout << "\nexception caught in worker " << id << ", e = " << e.what() << std::flush;
    }
}
```

Please refer to C++20 document, I will generalise this threadpool (with passing test) to :

- handle `std::jthread` for cooperative cancellation
- handle `std::coroutine_handle` for cooperative scheduling

Method 2 - Using semaphore in YLib

However the previous implementation is inflexible, besides wake up time for condition variable is slow, we want to customise each component so as to make it faster. Here comes a template version, with 3 template parameters :

- task type `T`
- queue type `Q`
- synchronization primitive `S` (hence we have better alternatives to condition variable)

First of all, type `std::function` is slow, we need faster alternative for tasks. Concept `std::invocable` is then picked, as it supports :

- function pointer
- member pointer
- lambda
- binded `std::function`

Secondly, the task queue can be locked or unlocked, locked queue may have various locking mechanisms. Thirdly, synchronization primitive can be anything that supports `wait` and `notify`, which may be futex, semaphore, condition variable, promise and future. In this implementation, again we decouple `while` loop into two, with no waiting in the second loop to avoid missing notification.

```
template<std::invocable T, template<typename> typename Q = locked_queue, typename S = std::semaphore>
class threadpool
{
public:
    threadpool(std::uint32_t num_threads, const std::vector<std::uint32_t>& affinity) : threadpool(num_threads)
    {
        for(std::uint32_t n=0; n!=num_threads; ++n)
        {
            threads.emplace_back(std::thread(&threadpool<T,Q>::thread_fct, this));
        }
        for(auto& x:threads)
        {
            set_thread_affinity(x.native_handle(), affinity);
            set_thread_priority(x.native_handle(), SCHED_RR);
        }
    }

   ~threadpool()
    {
        for(std::uint32_t n=0; n!=threads.size(); ++n) threads[n].join();
    }

public:
    void stop()
    {
        run.store(false);
        for(std::uint32_t n=0; n!=threads.size(); ++n) sync.notify();
    }

    template<typename... ARGS>
    void emplace_task(ARGS&&... args)
    {
        bool done = false;
        while(!done) // for lockfree_mpmcq, multiple emplace may be needed when it is nearly full
        {
            done = task_queue.emplace(std::forward<ARGS>(args)...);
        }
        sync.notify();
    }

private:
    void thread_fct()
    {
        // *** 1st loop *** //
        while(run.load())
        {
            sync.wait();
            std::optional<T> task = task_queue.pop();
            if (task) (*task)();
        }

        // *** 2nd loop *** //
        while(task_queue.peek_size() > 0)
        {
            // no waiting
            std::optional<T> task = task_queue.pop();
            if (task) (*task)();
        }
    }

private:
    std::atomic<bool> run;
    std::vector<std::thread> threads;
    Q<T> task_queue;
    S sync;
};
```

Synchronization primitives can :

- ensure happen-before relationship between A and B *(particularly in publication pattern)* if ...
- producer performs A before invoking `notify()` or `V()` of synchronization primitive
- consumer invokes `wait()` or `P()` of synchronization primitive before performing B
- hence most of the time, A refers ro production while B refers to consumption

Here are some synchronization primitives :

- no synchronization (so that we can reuse the same `threadpool` for busy waiting)
- futex, which is blocked when a target variable `futex` equals to a predefined value `blocking_value`
- *counting semaphore* in posix library (`std::semaphore` is not available in `gcc10` yet)
- *counting semaphore* implemented with *binary semaphore* (mutex in posix library is probably a binary semaphore)
- *counting semaphore* implemented with mutex plus condition variable (pretty standard algorithm)

The wake up time for different synchronization primitives are slighty different. Here are the time measurement in Ubuntu 4.5GHz machine, running release version in real time mode.

- futex 1700 ns
- *semaphore* in posix library 1700 ns
- *semaphore* with *binary semaphore* 1700 ns
- *semaphore* with mutex + condvar 2300 ns

```
class no_sync
{
public:
    no_sync_futex() = default;
   ~no_sync_futex() = default;

    inline void wait()   {}
    inline void notify() {}
};


class sync_futex // Please refer to reckless-log for correct usage of futex
{
public:
    sync_futex() : blocking_value(0){}
   ~sync_futex() = default;

    inline void wait()
    {
        // This thread is blocked when futex == blocking_value.
        syscall(SYS_futex, &futex, FUTEX_WAIT, blocking_value, NULL, NULL, 0);
        futex.fetch_sub(1);
    }

    inline void notify()
    {
        // Potential hazard for mpmc scenario, the following 2 steps are not atomic.
        futex.fetch_add(1);
        syscall(SYS_futex, &futex, FUTEX_WAKE, 1, NULL, NULL, 0);
    }

private:
    std::atomic<std::int32_t> futex;
    const std::int32_t blocking_value;
};


class sync_semaphore
{
public:
    sync_semaphore()
    {
        sem_init(&semaphore, 0, 0);
        // arg[1] : 0 for multi-thread, 1 for multi-process
        // arg[2] : initial value
    }

   ~sync_semaphore()         {  sem_destroy(&semaphore);   }
    inline void wait()       {  sem_wait(&semaphore);      }
    inline void notify()     {  sem_post(&semaphore);      }
    inline auto peek_value()
    {
        std::int32_t x;
        sem_getvalue(&semaphore, &x);
        return x;
    }

private:
    sem_t semaphore;
};
```

```
class sync_HansBarz // implement counting semaphore with binary semaphore using Hans W Barz algo in 1983
{
public:
    sync_HansBarz() : count(0)
    {
        pthread_mutex_init(&cs_mutex, NULL);
        pthread_mutex_init(&pv_mutex, NULL);
        pthread_mutex_lock(&pv_mutex);
    }

   ~sync_HansBarz()
    {
        pthread_mutex_unlock (&pv_mutex);
        pthread_mutex_destroy(&pv_mutex);
        pthread_mutex_destroy(&cs_mutex);
    }

    inline void wait()
    {
        pthread_mutex_lock(&pv_mutex); // P() or equivalently, wait ...
        pthread_mutex_lock(&cs_mutex);
        --count;
        if (count > 0)
        {
            pthread_mutex_unlock(&pv_mutex);
        }
        pthread_mutex_unlock(&cs_mutex);
    }

    inline void notify()
    {
        pthread_mutex_lock(&cs_mutex);
        ++count;
        if (count == 1)
        {
            pthread_mutex_unlock(&pv_mutex); // V() or equivalently, notify ...
        }
        pthread_mutex_unlock(&cs_mutex);
    }

private:
    std::int32_t count;
    pthread_mutex_t cs_mutex; // for critical session protection (regarded as a mutex)
    pthread_mutex_t pv_mutex; // for P() V() signaling           (regarded as a binary semaphore)
};


class sync_condvar // implement counting semaphore with mutex and condition variable (standard algo)
{
public:
    sync_condvar() : count(0) {}
   ~sync_condvar() = default;

    void wait()
    {
        std::unique_lock<std::mutex> lock(mutex);
        while(count == 0) cv.wait(lock);
        --count;
    }

    void notify()
    {
        {
            std::lock_guard<std::mutex> lock(mutex);
            ++count;
        }
        cv.notify_one();
    }

private:
    std::mutex mutex;
    std::condition_variable cv;
    std::uint32_t count;
};
```

Method 3 & 4 - Using polling (locked queue or lockfree queue)

The implementation is straight foward, just plugin an appropriate synchronization primitive and queue in the threadpool. Beware that we should assign one core per thread, otherwise busy waiting with spinlock or lockfree is a waste of CPU resources.

```
template<std::invocable T> using spinlock_threadpool = threadpool<T, no_sync_primitive, spinlocked_queue>;
template<std::invocable T> using lockfree_threadpool = threadpool<T, no_sync_primitive, lockfree_queue>; // become a disruptor
```

D4. IO service using promise and future

What is an IO service?

- `io_service` is a queue of async-tasks, with execution delayed until `io_service::run()` is called
- On completion of async-tasks, `boost::io_service` invokes registered callbacks.
  On completion of async-tasks, this `io_service` does not invoke callback, instead product `T` is returned via `std::forward<T>`.

Comparison

	Threadpool	io-service
producer of tasks	`threadpool::add_task()`	`io_service::add_async_task()`
consumer of tasks	`threadpool::fct()`	`io_service::run()`

Implementation

We can also extend the following class so that each task is invoked at a user-specfied timepoint. *See Atom interview Q3.*

```
template<typename T> struct io_service
{
     std::future<T> add_async_task(const std::function<T()>& fct) // producer of task
     {
          std::packaged_task<T()> task(fct);
          ptasks.push_back(std::move(task));
          return ptasks.back().get_future();
     }

     void run() // consumer of task
     {
          for(auto& x:ptasks) x();
     }

     std::vector<std::packaged_task<T()>> ptasks;
};

// *** Test program *** //
struct PV
{
     int value;
     std::string ccy;
};

struct DEAL
{
     std::string id;
     schedule sch;
     payoff pay;

     PV calculate(const model& model)
     {
          ...
          return make_present_value(BlackScholes(S,r,v,K,T), currency::USD);
     }
};

DEAL deal0{"HSI0601"};
DEAL deal1{"HSI0701"};
DEAL deal2{"HSI0801"};

io_service<PV> service;
auto future0 = service.add_async_task(std::bind(&DEAL::calculate, std::ref(deal0))); // non-blocking
auto future1 = service.add_async_task(std::bind(&DEAL::calculate, std::ref(deal1)));
auto future2 = service.add_async_task(std::bind(&DEAL::calculate, std::ref(deal2)));
std::thread t(&io_service<PV>::run, &service); // non-blocking

auto pv0 = future0.get(); // blocks here
auto pv1 = future1.get();
auto pv2 = future2.get();
t.join();
```

D5. Counting semaphore / Binary semaphore / mutex

Semaphore is a synchronization primitive that :

- protect a counter
- increment and decrement by two **atomic** functions, officially known as `P()` and `V()`
- `P()` or `wait()` if counter is greater than zero, decrements it, otherwise this thread **waits**
- `V()` or `notify()` if there are waiting threads, **notifies** one of them, otherwise increment counter
- `P()` or `wait()` corresponds to `pc_queue<T>pop` called by consumer
- `V()` or `notify()` corresponds to `pc_queue<T>push` called by producer
- for integer counter, it is called counting semaphore
- for binary counter, it is called binary semaphore
- binary semaphore is not mutex :
- there is no lock ownership in semaphore, `P()` and `V()` can be invoked by different threads (it is a signal mechanism)
- there is lock ownership in mutex, `lock()` and `unlock()` must be invoked by the same thread
- both binary semaphore and counting semaphore can be used to synchronize multiple threads

Relationship among binary semaphore / counting semaphore and mutex

We can conclude 4 relationships among those synchronization primitives :

- binary semaphore can be implemented by counting semaphore : by constraining counter to boolean
- counter semaphore can be implemented by binary semaphore : by *Hans W Barx* algorithm in 1983 (other algos are incorrect)
- counter semaphore can be implemented by mutex together with condition variable (we cannot use mutex only)
- please refer to the synchronization primitives discussed in previous section
- mutex can be implemented by semaphore as the following :

```
struct semaphore_as_mutex
{
    enum { MAX_NUM = 3 };

    // Only allows MAX_NUM threads running concurrently
    void fct(const std::string& name)
    {
        s.decrement();
        // critical session starts ...
        std::this_thread::sleep_for(std::chrono::seconds(2));
        // critical session ends ...
        s.increment();
    }

    semaphore s{ MAX_NUM };
};
```

specialize N=2

binary semaphore ⇐ ⇒ counter semaphore

see above ⇓ *Hans W Barz* ⇑ *refer to synchronization primitives*

mutex mutex + condition variable

Let's instantiate 30 threads and pass them through a semaphore with counter 3. The threads are blocked at `s.decrement()` and only 3 can pass through it at the same time. Thus it needs $30 / 3 \times 2$ seconds in total.

```
semaphore_as_mutex tester;
std::vector<std::thread> threads;
for(int n=0; n!=30; ++n)
{
    auto id = std::string("thread_").append(std::to_string(n));
    threads.push_back(std::thread{&semaphore_tester::fct, &tester, id});
}
for(auto& x:threads) x.join();
```

E1. Comparison among different locks

We will compare different mechanisms for critical session protection :

- mutex lock
- spinlock
- atomic variable
- no protection

In each case, there are two threads :

- one runs an increment routine
- one runs a decrement routine
- final answer should be zero

```
class my_task // Approach 1 : using mutex
{
    void update(bool incremental, unsigned long num)
    {
        for (unsigned long n=0; n!=num; ++n)
        {
            std::lock_guard<std::mutex> lock(m); if (incremental) ++count; else --count;
        }
    }
    signed long count;
    std::mutex m;
};

class my_task // Approach 2 : using spinlock
{
    void update(bool incremental, unsigned long num)
    {
        for (unsigned long n=0; n!=num; ++n)
        {
            std::lock_guard<alg::spinlock> lock(m); if (incremental) ++count; else --count;
        }
    }
    signed long count;
    alg::spinlock spin;
};

class my_task // Approach 3 : using atomic
{
    void update(bool incremental, unsigned long num)
    {
        for (unsigned long n=0; n!=num; ++n)
        {
            if (incremental) count.fetch_add(1, std::memory_order_relax);
            else             count.fetch_sub(1, std::memory_order_relax);
        }
    }
    st::atomic<signed long> count;
};

class my_task // Approach 4 : no protection
{
    void update(bool incremental, unsigned long num)
    {
        for (unsigned long n=0; n!=num; ++n)
        {
            if (incremental) ++count; else --count;
        }
    }
    signed long count;
};

// my_task contains mutex, hence my_task is non-copyable, need to pass by std::ref()
my_task task;
std::thread t0(&my_task::update, std::ref(task), true,  10000000);
std::thread t1(&my_task::update, std::ref(task), false, 10000000);
t0.join(); t1.join();
```

This is not tested in MSVS, I simply quote the results from http://demin.ws/blog/english/2012/05/05/atomic-spinlock-mutex.

approach	*relative time*	*remark*
using mutex	*22s*	*no parallelism, plus* ***lock contention***
using spinlock	*0.54s*	
using atomic	*0.45s*	
no synchronization	*0.07s*	*incorrect answer is obtained, i.e. final answer ≠ 0*

E2. Divide and conquer

Divide and conquer can be done by concurrent programming using `std::async`.

```
template<typename ITER> auto concurrent_divide_n_conquer(ITER begin, ITER end)
{
    auto size = end - begin;
    auto mid = begin + size/2;
    if (size == 1) return *begin;

    // half of the task is delegated to std::async
    auto f = std::async(concurrent_divide_n_conquer<ITER>, mid, end);

    // half of the task is done itself
    auto x = concurrent_divide_n_conquer<ITER>(begin, mid);
    return x + f.get(); // std::this_thread is blocked here
}

std::vector<int> v;
for(int n=0; n!=20; ++n) v.push_back(10 + rand()%20);
std::cout << "\nanswer = " << concurrent_divide_n_conquer(&sum1, &sum2, v.begin(), v.end());
```

E3. Thread local storage

Global variable is defaulted to be a single instance shared among multiple threads. By declaring global variable as `thread_local` each thread has its instance of the global variable.

```
thread_local std::atomic<unsigned long> global_variable = 0;
//           std::atomic<unsigned long> global_variable = 0;

void function()
{
    for(int n=0; n!=100; ++n) global_variable.fetch_add(1);
    std::cout << "thread " << std::this_thread::get_id() << " ans = " << global_variable;
}

std::vector<std::thread> threads;
for(int n=0; n!=5; ++n) threads.push_back(std::thread(function));
for(auto& x:threads) x.join();

std::cout << "main thread " << std::this_thread::get_id() << " ans = " << global_variable;
```

If `thread_local` isn't declared, all threads share the same counter, final count equals to 500. If `thread_local` is declared, each thread has its own copy, final count in main thread is not modified at all, hence remaining at 0.

not declaring thread_local	declaring thread_local
thread 10131 ans = 154	thread 10131 ans = 100
thread 10672 ans = 243	thread 10672 ans = 100
thread 10369 ans = 367	thread 10369 ans = 100
thread 10273 ans = 474	thread 10273 ans = 100
thread 10374 ans = 500	thread 10374 ans = 100
main thread 10082 ans = 500	main thread 10082 ans = 0

E4. Three important aspects of concurrency *Please search Cooperative vs Preemptive, Bobby Priambodo.*

Scheduling can be preemptive (dictated by scheduler) and cooperative (running thread giving up resources willingly for the sake of others). The former is for maximizing throughput when number of threads is greater than the number of cores, and the latter is for minimising latency when number of threads equal to the number of cores, pinning affinity is possible. The former usually involves waiting (probably with mutex, condition variable or async etc), while the latter usually involves spinning lock or atomic. Waiting is essential for releasing unused cpu resources to someone who really need it.

preemptive	*cooperative*
scheduler with context switch	yielding willingly minimise context switch
high throughput	low latency
# threads > # cores	# threads = # cores (pinning affinity)
involves waiting	involves spinning (busy waiting)

Thread model

- kernel thread like `std::thread` in `YLib` with OS scheduler
- user thread (also known as green thread) is pseudo thread with user program as scheduler (such as coroutine?)

Memory model

- the concurrency mechanism for shared resources among threads
- involves : mutex, spinlock, semaphore, condition variable, promise, future, coroutine, synchronization primitives, atomic ...