Макрос

Макросы – это препроцессорные «функции», те лексемы, созданные с помощью директивы #define, которые принимают параметры подобно функциям. Макросы записываются в виде #define ИМЯ_МАКРОСА (ПАРАМЕТРЫ) через пробел ОПРЕДЕЛЕНИЕ МАКРОСА. Макрос может состоять не только из одного выражения.

Также мы можем задавать область видимости для макроса (с помощью statement expression), чтобы мы могли безопасно объявлять новые переменные внутри макроса. Однако данная конструкция является GNU расширением языка.

```
#define SWAP(type, a, b) ({type temp = a; a = b; b = temp; })
```

Внутри такой конструкции так же удобно использовать typeof (C), decltype (C++), для того чтобы определять новую переменную такого же типа, как и параметры

```
#define SWAP(a, b) ({decltype(a) temp = a; a = b; b = temp; })
```

Макросы можно записывать в несколько строк, но тогда каждая строка кроме последней должна заканчиваться символом \

Параметр макроса можно превратить в строку добавив #.

```
#define PRINT_VALUE(value) printf("Value of %s is %d", #value, value);
```

Для того чтобы приклеить макрос к чему-то еще, надо написать ##

```
#define PRINT_VALUE (number) printf(“%d”, value_##number);
Int value_one = 10, value_two = 20;
PRINT_VALUE(one);
PRINT_VALUE(two);
```

Правила работы с макросами:

- Параметрами макросов не должны быть выражения и вызовы функций.
- Все аргументы макроса и сам макрос должны быть заключены в скобки
- Многострочные макроса должны иметь свою область видимости

Auto

У auto есть специальное правило вывода типа. Когда инициализатор для переменной, объявленной как auto, заключён в фигурные скобки, выведенный тип – std::initializer_list.

Вывод типа auto отличается от вывода типа шаблона в том, что auto предполагает, что инициализатор в фигурных скобках представляет собой std::initializer_list, в то время как вывод типа шаблона этого не делает.

Auto помогает избегать ошибок при переходе из 32-битной системы Windows в 64-битную.

//В общем, auto нужно для избежания ошибок при явном определении типа, если тип сложный для понимания.//

Если auto выводит нежелательный тип, нужно воспользоваться явно типизированным инициализатором.

«Невидимые» прокси-типы могут привести auto к выводу неверного типа инициализирующего выражения (прокси-тип/прокси-класс – класс, цель которого – эмуляция и дополнение поведения некоторого другого типа)

Семантика перемещения

Семантика перемещения

Std::move ничего не перемещает, а выполняет безусловное приведение своего аргумента к rvalue. Использование std::move сообщает компилятору, что объект предназначается для перемещения

Семантика перемещения непригодна в следующих случаях:

- Отсутствие перемещающих операций
- Перемещение не быстрее
- Перемещение неприменимо
- Исходный объект является lvalue

Семантика перемещения (move semantics) - это концепция в языке C++, которая позволяет эффективно перемещать ресурсы (например, динамически выделенную память) из одного объекта в другой, без необходимости выполнения глубокого копирования. В C++ семантика перемещения реализуется с использованием rvalue-ссылок (T&&) и особого конструктора перемещения (move constructor) и оператора присваивания перемещением (move assignment operator).

Прямая передача

Std::forward ничего не передает, а выполняет приведение своего аргумента к rvalue при соблюдении определенных условий (когда аргумент связан с rvalue)

Перечисления с областью видимостью

В качестве общего правила объявления имени в фигурных скобках ограничивает видимость этого имени областью видимости, определяемой этими скобками. Но в C++98 имена в перечислениях имеют ту же область видимости, что и перечисление, в C++11 были введены перечисления с областью видимости (scoped enum). Такие перечисления с областью видимости объявляют как **enum class** (обычные объявляются обычным **enum**). Перечисления с областью видимости строже типизированы, в них нет неявных преобразования элементов в другой тип, для того чтобы преобразовать тип нужно воспользоваться явным приведением типа

Универсальные ссылки

В c++ используемые значения мы можем разделить на две группы: lvalue и **rvalue**. lvalue представляет именованное значение, например, переменные, параметры, константы. С lvalue ассоциирован некоторый адрес в памяти, в котором на постоянной основе хранится некоторое значение. И мы можем lvalue присвоить некоторое значение. А **rvalue -** это то, что можно только присваивать, например, литералы или результаты выражений.

Универсальная ссылка — это не особая разновидность ссылок, а некоторый специальный механизм автоматического вывода аргумента шаблона и для того чтобы он был использован, необходимо выполнение трех условий.

1. Наличие шаблона функции с типовым параметром (обозначим его через T).
2. Параметр функции объявлен как T&&.
3. Аргумент шаблона выводится автоматически, исходя из типа аргумента вызова функции.

Noexcept

Это модификатор, который применяется к функциям, которые гарантированно не вызывают исключения. Использование noexcept позволяет компиляторам генерировать лучший объектный код.

Разница между способами в C++98 и в C++11 сказать, что функция не генерирует исключения

```
int f (int x) throw(); //C++98
int f (int x) noexcept; //C++11
```

Если во время выполнения некоторое исключение покинет f, тем самым будет нарушена спецификация исключений f. При спецификации исключений C++98 стек вызовов сворачивается до вызывающего f кода, и после некоторых действий выполнение программы прекращается. При спецификации исключений C++11 поведение времени выполнения немного иное: стек только, ***возможно***, сворачивается перед завершением выполнения программы.

Разница между сворачиванием стека и ***возможным*** сворачиванием оказывает большое влияние на генерацию кода. В случае функции, объявленной как noexcept, оптимизаторам не надо ни поддерживать стек в сворачиваемом состоянии, ни гарантировать, что объекты в такой функции будут уничтожены в порядке, обратном созданию, если вдруг такую функцию покинет исключение.

Заранее noexcept функции: delete, delete[], деструкторы.

Constinit(C++20), consteval(C++20), constexpr(C++11)

Constexpr – спецификатор типа, введенный для обозначения константных выражений, которые могут быть вычислены во время компиляции программы. Вычисление в момент компиляции происходит при условии, что передаваемые значения могут быть посчитаны на этапе компиляции

Consteval – это функции, которые вычисляется исключительно во время компиляции. В отличие от constexpr, которые могут вызываться как в run-time, так и в compile-time, эти функции не получится даже вызвать во время выполнения, будет ошибка.

Constinit – это переменная, которая инициализируется статически, причем если компилятор не проинициализирует переменную на этапе компиляции, код не соберется. Переменные с ключевым словом constinit не обязаны быть константами

SFINAE

SFINAE – substitution [of template parameter] fail is not an error = при определении перегрузок функции ошибочные инстанциации шаблонов не вызывают ошибку компиляции

```
struct L {
typedef double internal_type;
};
template <typename T>
typename T::intenal_type
foo(const T& t){cout << "foo(T)" << std::endl; return 0;}
int main() {
foo(L()); // OK
foo(0); //Error
return 0; }
```

Decltype

Для данного имени или выражения decltype возвращает тип этого имени или выражения (почти всегда без изменений)

Для lvalue-выражений, более сложных, чем имена, decltype гарантирует, что возвращаемые значение будет lvalue-ссылкой

Declval

Declval преобразует любой тип в ссылочный, что позволяет использовать методы в выражениях decltype без необходимости использования конструкторов. Заголовок <utility>

```
#include <utility>
#include <iostream>

struct Default { int foo() const { return 1; } };

struct NonDefault
{
    NonDefault() = delete;
    int foo() const { return 1; }
};

int main()
{
    decltype(Default().foo()) n1 = 1;                   // тип n1 - int
// decltype (NonDefault (). foo ()) n2 = n1; // ошибка: нет конструктора по умолчанию
    decltype(std::declval<NonDefault>().foo()) n2 = n1; // тип n2 - int
    std::cout << "n1 = " << n1 << '\n'
              << "n2 = " << n2 << '\n';
}
```

Смягчение уровня доступа через using (Наверное, про using enum) (C++20)

Конструкция **using enum** делает видимыми без квалификации все константы из **enum**

```
enum class fruit { orange, apple };
enum class color { red, orange };
void f() {
    using enum fruit; // OK
    using enum color; // <-- ошибка – конфликт
}
```

Default (C++11); delete (C++11)

Ключевое слово **default**. Используется для делегирования компилятору определения функции класса. Ключевое слово **default** может применяться только к специальным функциям класса, которые при объявлении класса генерируются компилятором автоматически.

К таким функциям относятся:

- конструктор по умолчанию (default constructor);
- конструктор копирования (copy constructor);

- конструктор перемещения (move constructor);
- оператор присваивания копированием (copy assignment operator);
- оператор присваивания перемещением (move assignment operator);
- деструктор (destructor).

Ключевое слово **delete** используется в случаях, когда нужно запретить автоматическое приведение типов в конструкторах и методах класса. **Delete** предотвращает определение или вызов ненужных функций.

Для запрета на неявные преобразования типов используется также **explicit**.

Explicit ставится как ключевое слово перед функцией

А с **default** и **delete**: class_name (…) = **default**/**delete**.

Сырые строки

строки, в которых обратный слеш не модифицирует следующий за ним символ.

Пользовательские литералы (в целом немного о литералах)

Литерал – значение, вставленное непосредственно в код (н-р, return 5 // 5 – целочисленный литерал)

Сырой литерал приходит на помощь, когда входное число надо разобрать посимвольно.

Используя данный тип литералов, можно написать пользовательский литерал преобразующий двоичное число в десятичное. Например вот так:

```
unsigned long long operator "" _b(const char* str)
{
    unsigned long long result = 0;
    size_t size = strlen(str);

    for (size_t i = 0; i < size; ++i)
    {
        assert(str[i] == '1' || str[i] == '0');
        result |= (str[i] - '0') << (size - i - 1);
    }

    return result;
}

// ...

std::cout << 101100_b << std::endl; // выведет 44
```

Лямбда-выражения

Auto [параметр, указывающий на захват переменных] (параметры) {тело функции}

Если надо получить все внешние переменные из области, где определено лямбда-выражение, по значению, то в квадратных скобках указывают символ «равно». Но в таком случае внешние значения изменить нельзя.

Если надо получить внешние переменные по ссылке, то в квадратных скобках указывается символ амперсанда &. В этом случае лямбда-выражение может изменять значения этих переменных.

В предыдущем случае мы смогли получить внешнюю переменную и изменить ее значение. Но иногда бывает необходимо изменять копию переменной, которую использует лямбда-выражение, а не саму внешнюю переменную. В этом случае мы можем поставить после списка параметров ключевое слово **mutable**.

По умолчанию выражения **[=]/[&]** позволяют захватить все переменные из окружения. Но также можно захватить только определенные переменные. Чтобы получить внешние переменные, применяется выражение **[&имя_переменной] или [имя_переменной]**

С 20 стандарта появились еще угловые скобки для шаблонных параметров.

Многопоточное программирование

Отличие std::async от std::thread в том, что можно указать стратегию выполнения std::launch::async, std::launch::deferred или их комбинацию. План выполнения может быть немедленным и отложенным.

- std::launch::async означает, что функция должна выполняться асинхронно, те в другом потоке

- std::launch::deferred означает, что ф-ция f может выполняться только тогда, когда для фьючерса, возвращенного std::async, вызывается метод get или wait, те выполнение ф-ции f откладывается до тех пор, пока не будет выполнен такой вызов. Когда вызываются методы get или wait, ф-ция f выполняется синхронно, то есть вызывающая ф-ция блокируется до тех пор, пока f не завершит работу.
- Стратегия запуска по умолчанию для std::async допускает как асинхронное, так и синхронное выполнение задачи

Также std::async позволяет получить доступ к вызову

std::thread t(doAsyncWork);
auto fut = **std::async**(doAsyncWork);

Фьючерс, возвращаемый std::async, предлагает функцию get, с помощью которой можно получить доступ к вызову, также get имеет доступ к исключениям. То есть если при работе с потоками функция генерирует исключение, то программа завершается аварийно.

Фундаментальное отличие подхода на основе задач и на основе потоков заключается в более высоком уровне абстракции первого подхода. Он освобождает от деталей управления потоками

ВИДЫ ПОТОКОВ:

- Аппаратные потоки – потоки, которые выполняют фактические вычисления
- Программные потоки (потоки ОС или системные потоки) – потоки, управляемые ОС во всех процессах и планируемыми для выполнения аппаратными потоками. Обычно программных потоков можно создать больше чем аппаратных, поскольку, когда программный поток заблокирован (н-р, при ожидании мьютекса), пропускная способность может быть повышена путем выполнения незаблокированных других потоков.
- Std::thread – объекты в процессе C++, которые действуют как дескрипторы (указатели на объекты в куче) для лежащих в их основе программных потоков. Есть потоки, которые представляют собой нулевые дескрипторы, к ним относятся:
  - Объекты, сконструированные по умолчанию
  - Объекты, из которых выполнили перемещение
  - Объекты, у которых разорвана связь с потоком выполнения

hardware_concurrency - returns the number of concurrent threads supported by the implementation

Некоторые полезные функции, предоставляемые <thread>, в пространстве имен std::this_thread:

1.get_id: возвращает id текущего потока

2.yield: говорит планировщику выполнять другие потоки, может использоваться при активном ожидании

3.sleep_for: блокирует выполнение текущего потока в течение установленного периода

4.sleep_until: блокирует выполнение текущего потока, пока не будет достигнут указанный момент времени

Класс jthread представляет собой один поток выполнения. Он имеет то же поведение, что и std::thread, за исключением того, что jthread автоматически join'ится при уничтожении и предлагает интерфейс для остановки потока.

Подключить больше потоков, чем может предоставить система, не получится, это вызовет ошибку. Даже если вы не исчерпали потоки, может возникнуть проблема с превышением подписки (oversubscription). Это происходит, когда имеется больше готовых к запуску программных потоков, чем аппаратных. С этой проблемой помогает справиться std::async.

std::async – подход на основе задач
std::thread – подход на основе потоков

std :: async: выполняет асинхронную операцию.
std :: future: обеспечивает доступ к результатам асинхронной операции.
std :: prom: упаковывает результат асинхронной операции.
std :: packaged_task: связывает функцию и связанное с ней обещание для возвращаемого типа.

Future и std::promise.

Фьючерс представляет собой один из концов связи, по которому вызываемая функция передает результаты вызывающей. Вызываемая функция записывает результат вычислений в коммуникационный канал (в основном с помощью std::promise), а вызывающая функция читает результат с помощью фьючерса.

Общее состояние обычно представлено объектом в динамической памяти, но его тип, интерфейс и реализация в стандарте языка не указаны, в общем состоянии храниться результат вызываемой функции.

Std::package_task подготавливает функцию к асинхронному выполнению, заворачивая ее таким образом, что ее результат помещается в общее состояние.

Можно получить фьючерс с помощью get_future.

Semaphore (C++20).

Семафор (semaphore) — примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующей. Служит для построения более сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.

Семафоры могут быть двоичными и вычислительными. Вычислительные семафоры могут принимать целочисленные неотрицательные значения и используются для работы с ресурсами, количество которых ограничено, либо участвуют в синхронизации параллельно исполняемых задач. Двоичные семафоры могут принимать только значения 0 и 1 и используются для взаимного исключения одновременного нахождения двух или более процессов в своих критических секциях.

counting_semaphore - это примитив синхронизации, который может управлять доступом к общему ресурсу. В отличие от мьютекса std::mutex, counting_semaphore допускает более одного параллельного доступа к одному и тому же ресурсу.

Mutex.

Мьютекс — базовый элемент синхронизации и в C++11 представлен в заголовочном файле <mutex>:

- mutex: обеспечивает базовые функции lock() и unlock() и не блокируемый метод try_lock()
- recursive_mutex: может войти «сам в себя»
- timed_mutex: в отличие от обычного мьютекса, имеет еще два метода: try_lock_for() и try_lock_until()
- recursive_timed_mutex: это комбинация timed_mutex и recursive_mutex
- shared_mutex — это примитив синхронизации, который может использоваться для защиты общих данных от одновременного доступа нескольких потоков. В отличие от других типов мьютексов, которые обеспечивают эксклюзивный доступ, shared_mutex имеет два уровня доступа:
  - общий доступ - несколько потоков могут совместно владеть одним и тем же мьютексом.
  - эксклюзивный доступ (исключительная блокировка) - только один поток может владеть мьютексом

Mutex является примитивом синхронизации.

Примитивы синхронизации — механизмы, позволяющие реализовать взаимодействие потоков, например, единовременный доступ только одного потока к критической области.

Примитивы синхронизации преследуют различные задачи:

- Взаимное исключение потоков — примитивы синхронизации гарантируют то, что единовременно с критической областью будет работать только один поток.
- Синхронизация потоков — примитивы синхронизации помогают отслеживать наступление тех или иных конкретных событий, то есть поток не будет работать, пока не наступило какое-то событие. Другой поток в таком случае должен гарантировать наступление данного события.

Классы «обертки» позволяют непротиворечиво использовать мьютекс в RAII-стиле с автоматической блокировкой и разблокировкой в рамках одного блока. Эти классы:

- lock_guard: когда объект создан, он пытается получить мьютекс (вызывая lock()), а когда объект уничтожен, он автоматически освобождает мьютекс (вызывая unlock())
- unique_lock: в отличие от lock_guard, также поддерживает отложенную блокировку, временную блокировку, рекурсивную блокировку и использование condition variables (условных переменных)
- scoped_lock. deadlock-avoiding RAII wrapper for multiple mutexes (analogue lock_quard)
- shared_lock — это аналог std::unique_lock для получения общего доступа к данным, защищаемым с помощью shared_mutex. Он позволяет отсроченную блокировку, попытку блокировки с таймаутом и передачу права владения блокировкой.

<condition_variable>

Класс condition_variable — это примитив синхронизации, который может использоваться для блокировки потока или нескольких потоков до тех пор, пока другой поток не изменит общую переменную (не выполнит условие) и не уведомит об этом condition_variable.

Поток, который намеревается изменить общую переменную, должен:

- захватить std::mutex (обычно через std::lock_guard)

- выполнить модификацию, пока удерживается блокировка мьютекса
- выполнить notify_one или notify_all на std::condition_variable (блокировка не должна удерживаться для уведомления)

Даже если общая переменная является атомарной, всё равно требуется использовать мьютекс для корректного оповещения ожидающих потоков.

Класс condition_variable используется для ожидания события при наличии mutex типа unique_lock<mutex>.

Методы:

- Wait
- Wait_for
- Wait_until
- Notify_all
- Notify_one

Класс condition_variable_any используется для ожидания события, которое имеет любой тип mutex.

Защёлки и барьеры.

Защелки latches и барьеры barriers — это механизм синхронизации потоков, который позволяет блокировать любое количество потоков до тех пор, пока ожидаемое количество потоков не достигнет барьера. Защелки нельзя использовать повторно, барьеры можно использовать повторно.

Ошибки, которые могут возникнуть при работе с потоками.

Deadlock — ситуация, при которой несколько потоков находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать выполнение.

Состояние гонки — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

Голодание потоков — это ситуация, в которой поток не может получить доступ к общим ресурсам, потому что на эти ресурсы всегда претендуют какие-то другие потоки, которым отдаётся предпочтение.

Сопрограммы

Сопрограммы можно рассматривать как обобщение понятия подпрограмм (routines, функций) в срезе выполняемых над ними операций. Принципиальное различие между сопрограммами и подпрограммами заключается в том, что сопрограмма обеспечивает возможность явно приостанавливать свое выполнение, отдавая контроль другим программным единицам и возобновлять свою работу в той же точке при получении контроля обратно, с помощью дополнительных операций, сохраняя локальные данные (состояние выполнения), между последовательными вызовами, тем самым обеспечивая более гибкий и расширенный поток управления.

Для оперирования сопрограммами стандарт вводит три ключевых оператора:

- co_await. Унарный оператор, позволяющий, в общем случае, приостановить выполнение сопрограммы и передать управление вызывающей стороне, пока не завершатся вычисления представленные операндом;
- co_yield. Унарный оператор, частный случай оператора co_await, позволяющий приостановить выполнение сопрограммы и передать управление и значение операнда вызывающей стороне;
- co_return. Оператор завершает работу сопрограммы, возвращая значение, после вызова сопрограмма больше не сможет возобновить свое выполнение

Сопрограммой не может быть:

- Функция main;
- Функция с оператором return;
- Функция помеченная constexpr;
- Функция с автоматическим выведением типа возвращаемого значения (auto);
- Функция с переменным числом аргументов (variadic arguments, не путать с variadic templates);
- Конструктор;
- Деструктор.

На тип возвращаемого значения наложены ограничения:

- Должен быть классом
- Должен содержать идентификатор std::coroutine_handle
- Должен содержать promise_type

User types.

С сопрограммами ассоциировано несколько интерфейсных типов, позволяющих настраивать поведение сопрограммы и контролировать семантику операторов.

Promise.

Объект типа Promise позволяет настраивать поведения сопрограммы как программной единицы. Должен определять:

- Поведение сопрограммы при первом вызове;
- Поведение при выходе из сопрограммы;
- Стратегию обработки исключительных ситуаций;
- Необходимость в дополнительном уточнении типа выражения операторов co_await;
- Передача промежуточных и конечных результатов выполнения вызывающей стороне.
- Также тип promise участвует в разрешении перегрузки операторов new и delete, что позволяет настраивать динамическое размещение фрейма сопрограммы.
- Объект типа promise создаётся и хранится в рамках фрейма сопрограммы для каждого нового вызова.

Тип должен иметь строгое имя promise_type

Awaitable.

Объекты типа Awaitable определяют семантику потока управления сопрограммы. Позволяют:

- Определить, следует ли приостанавливать выполнение сопрограммы в точке вызова оператора co_await;
- Выполнить некоторую логику после приостановления выполнения сопрограммы для дальнейшего планирования возобновления ее работы (асинхронные операции);
- Получить результат вызова оператора co_await, после возобновления работы.

Исключения

```
…
if (this->col != this->rows) {
        throw "Wrong dimension\n";
    }
…
try {
    M = M1 + M1;
    fout << M;
}
 catch (const char* error_message) {
        cout << error_message;
   }
```

Умные указатели

Умные указатели – оболочки вокруг встроенных указателей, которые действуют так же, как и встроенные, но позволяют избежать многих проблем. К умным указателям относятся: std::auto_ptr, std::unique_ptr, std::shared_ptr, std::weak_ptr. std::auto_ptr является устаревшим указателем, так как не учитывает семантику перемещения, ему на смену пришел std::unique_ptr. Std::unique_ptr воплощает в себе семантику исключительного владения, он является только перемещаемым указателем (то есть передает владение объекта). Std::unique_ptr также заботится об очищении памяти. Размер этого указателя равен размеру обычного.

Std::unique_ptr легко преобразуется в std::shared_ptr

Std::shared_ptr совместное владение (никакой конкретный указатель не владеет объектом). Объект удаляет только тогда, когда удаляются все указатели на этот объект. У этого указателя есть счётчик ссылок. Размер в два раза больше. Память для счетчика ссылок должна выделяться динамически. Инкремент и декремент – атомарные операции.

Std::weak_ptr справляется с проблемами, которые возникают при удалении объекта, на который указывает weak_ptr

Chrono

Заголовок <chrono> пространство имен std.

Библиотека реализует следующие концепции:

- Интервалы времени (duration)

Шаблонный класс std::chrono::duration является типов интервала времени. Интервал времени в chrono – некоторое количество периодов, которое характеризуется типом, н-р, int64_t или float.

Продолжительность периода измеряется в секундах и представляется в виде натуральное дроби с помощью std::ratio.

При работе с ними неявная инициализация запрещена (seconds s = 5; //ERROR), только явная работает (seconds s{8}; //OK).

Время можно складывать, вычитать и сравнивать

Можно неявно преобразовывать часы в минуты, минуты в секунды и тк, но не наоборот. Можно округлять

- Момент времени (time_point)

Этот класс предназначен для представления моментов времени. Момент времени может быть охарактеризован как интервал времени, измеренным на каком-либо таймере, начиная с некоторой точки отсчёта.

- Таймеры (clock)

В библиотеке три таймера:

  - System_clock. Представляет время системы. Этот таймер обычно не подходит для измерения интервалов
  - Steady_clock. Представляет так называемые устойчивые часы, то есть ход которых не подвержен внешним изменениям. Хорошо подходит для измерения интервалов
  - High_resolution_clock. Таймер с минимально возможным периодом отсчетов, доступным в системе.

У этого класса есть статическая переменная is_steady, по которой можно узнать, является ли таймер монотонным. Есть функция now, возвращающая текущий момент времени в виде time_point. Time_point нельзя сложить друг с другом, однако можно вычесть, что помогает при засекании времени. Также можно вызвать ф-цию time_since_epoch, чтобы получить интервал времени, прошедший с момента начало отсчета

```
auto now = system_clock::now();
system_clock::duration tse = now.time_since_epoch();
```

Преобразование time_point в число, например для сериализации или вывода на экран, можно осуществить через С-тип time_t:

```
auto now = system_clock::now();
time_t now_t = system_clock::to_time_t(now);
auto now2 = system_clock::from_time_t(now_t);
```

Атомарные операции

Атомарными (неделимыми) называются операции, выполнение которых не может быть прервано переключением потоков — такая операция гарантированно завершится до переключения.

Атомарная операция имеет два ключевых свойства, которые помогают использовать несколько потоков для правильного управления объектом без использования mutex блокировок.

- Поскольку атомарная операция является неделимой, вторая атомарная операция с тем же объектом из другого потока может получить состояние объекта только до или после первой атомарной операции.
- Основываясь на своем memory_order аргументе, атомарная операция устанавливает требования к упорядочению для видимости последствий других атомарных операций в том же потоке. Следовательно, она подавляет оптимизации компилятора, которые нарушают требования к упорядоченности.

Операции:

- Store
- Load
- Exchange. Заменяет одно значение другим и возвращает первое
- Fetch_add. Добавляет значение к существующему значению, хранящемуся в объекте atomic
- Fetch_and. Выполняет побитовую операцию & со значением и существующим значением, хранящимся в объекте atomic.

- Fetch_or.
- Fetch_sub. Вычитает значение из существующего значения, хранящегося в объекте atomic.
- Fetch_xor.

Трехстороннее сравнение

На основе переопределенных операторов сравнения может сгенерировать оставшиеся. В качестве базовых используются == и <=>

Оператор трёхстороннего сравнения (<=>) был введен в C++20 и предоставляет механизм для сравнения двух объектов и определения их отношения (меньше, равно или больше).

Оператор <=> может быть перегружен в пользовательском классе для определения семантики сравнения объектов этого класса (В итоге компилятор может сгенерировать код для шести вариантов сравнения: <, <=, ==, !=, >, >=. В результате мы избавляемся от огромного количества кода, ведь для каждого оператора сравнения нам нужно было бы переопределять свой метод). Когда оператор <=> вызывается, он возвращает значение типа std::strong_ordering, который является перечислением с тремя возможными значениями:

- std::strong_ordering::less - указывает, что левый операнд меньше правого операнда.
- std::strong_ordering::equal - указывает, что левый операнд равен правому операнду.
- std::strong_ordering::greater - указывает, что левый операнд больше правого операнда.

Концепции.

Концепции — это функция языка C++20, которая ограничивает параметры шаблона во время компиляции. Они помогают предотвратить неправильный создание экземпляров шаблонов, указать требования к аргументам шаблона в доступной для чтения форме и предоставить более краткие ошибки компилятора, связанные с шаблоном.

Пример

```
template <class T>
concept Integral = std::is_integral<T>::value;
```

В ограничениях можно использовать выражения и даже вызывать функции. Но функции должны быть constexpr — они вычисляются на этапе компиляции

Для ограничений есть отличная возможность: проверка корректности выражения — того, что оно компилируется без ошибок. Посмотрите на ограничение Addable. В скобках написано a + b. Условия ограничения выполняются тогда, когда значения a и b типа T допускают такую запись, то есть T имеет определённую операцию сложения:

```
template<class T>
concept Addable =
requires (T a, T b) {
   a + b;
};
```

Ограничение может требовать не только корректность выражения, но и чтобы тип его значения чему-то соответствовал. Здесь мы записываем:

- выражение в фигурных скобках,
- ->,
- другое ограничение.

```
template<class T> concept C1 =
requires(T x) {
   {x + 1} -> std::same_as<int>;
};
```

Ограничение для функции можно написать в трёх разных местах:

```
// Вместо слова class или typename в шаблонную декларацию.
// Поддерживаются только концепты.
template<Incrementable T>
void f(T arg);

// Использовать ключевое слово requires. В таком случае их можно вставить
// в любое из двух мест.
// Годится даже неименованное ограничение.
template<class T>
requires Incrementable<T>
void f(T arg);
```

```
template<class T>
void f(T arg) requires Incrementable<T>;
```

Четвертый способ

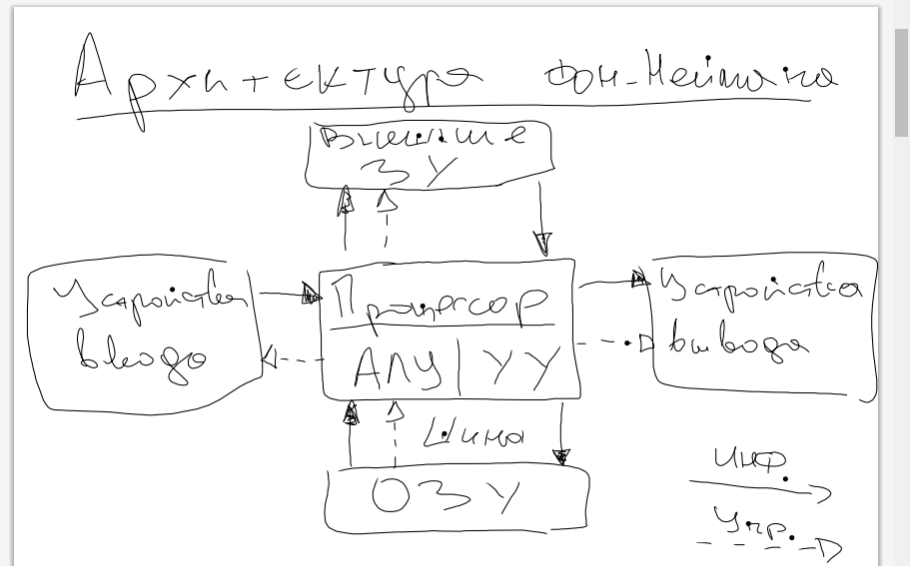
```
void f(Incrementable auto arg);
```

Важное отличие: когда мы пишем requires, можем записать любое ограничение, а в остальных случаях — только имя концепта.

Многие концепции уже определены в заголовочном файле <concepts>

Архитектура, принципы, CISC, RISC

Архитектура фон Неймана



Принципы:

- Двоичного кода
- Адресности памяти
- Хранимости программы
- Однородности памяти
- Программного управления

Таксономия Флинна

- SISD - single instruction stream/single data stream. Традиционный компьютер фон-Неймановской архитектуры с одним процессором, который выполняет последовательно одну инструкцию за другой, работая с одним потоком данных. Не является параллельной
- SIMD - **s**ingle **i**nstruction, **m**ultiple **d**ata. Типичный представитель – векторные процессоры и матричные процессоры
- MISD - **m**ultiple **i**nstruction, **s**ingle **d**ata. Физически не существует
- MIMD - **m**ultiple **i**nstruction, **m**ultiple **d**ata. Может быть любая реализация (можно подогнать все)
    - Классификация по памяти:
        - UMA (общая память). Способы реализации SMP
        - NUMA (раздельная память)

Архитектуры процессоров CISC RISC

- CISC – Complex Instruction Set Computer
    - \+ Компактность инструкций
    - \+ Поддержка высокоуровнего программирования
    - ± Команда выполняется за несколько тактов

- Высокая нерегулярность потока команд
- Высокая стоимость
- Сложность распараллеливания

● RISC – Reduced Instruction Set Computer

+ Единая кодировка и длина

± Ограниченный набор команд
± Только чтение/запись из памяти

- Высокий размер программ
- Одно действие за такт

Способы параллельной обработки

1. Конвейерная + векторная
2. SIMD
3. MIMD
4. Многопроцессорные SIMD

Модули в C++20

Модуль — это набор файлов исходного кода, которые компилируются независимо от исходных файлов. Макросы, директивы препроцессора и неэкспортированные имена, объявленные в модуле, не видны за пределами модуля.

Import нужен для подключения модулей в файл. После него нужна « **;** »

Операторы приведения

С помощью операции приведения можно указать компилятору преобразовать значение одного типа в другой тип.

Одно из эффективных применений приведения — это когда код выполняет сужающее преобразование и вы знаете, что преобразование не приводит к тому, что программа выдает неверные результаты. Фактически это сообщает компилятору, что вы знаете, что делаете, и перестаете беспокоить вас с предупреждениями об этом. Другой способ использования — приведение из указателя на производный класс к классу указателя на базовый. Другой способ использования — отбрасывает константность переменной, чтобы передать ее в функцию, требующую аргумента, не являющегося константным.

- static_cast— для приведения, которые проверяются только во время компиляции. static_cast возвращает ошибку, если компилятор обнаруживает, что вы пытаетесь выполнить приведение между полностью несовместимыми типами.

```
double d = 1.58947;
int i = d;  // warning C4244 possible loss of data
int j = static_cast<int>(d);       // No warning.
string s = static_cast<string>(d); // Error C2440:cannot convert from
                                   // double to std:string
```

- dynamic_cast— для безопасных приведения указателя к базовой точке к указателю на производный от среды выполнения. Объект dynamic_cast более безопасный, но проверка среды выполнения влечет за собой некоторые издержки.
- const_cast. Оператор приведения const_cast удаляет или добавляет квалификаторы const и volatile с исходного типа данных
- reinterpret_cast. Оператор приведения reinterpret_cast используется для приведения несовместимых типов. Отличие состоит в том, что reinterpret_cast не может снимать квалификаторы const и volatile, а также не может делать небезопасное приведение типов не через указатели, а напрямую по значению. Например, переменную типа int к переменной типа double привести при помощи reinterpret_cast нельзя.

Std::regex (регулярные выражения)

Регулярные выражения (иногда называемые регулярными выражениями или регулярными выражениями) представляют собой текстовый синтаксис, который представляет шаблоны, которые могут быть сопоставлены в используемых строках.

Используемая грамматика регулярного выражения определяется с помощью одного из значений std::regex_constants::syntax_option_type перечисления.

По умолчанию предполагается, что если грамматика не указана ECMAScript

- Повторения

- a{2} == “aa”
- a{2,} == “aa”, “aaa” …
- a* == “”, “a’, “aa”, …
- a? == “’, “a”
- a+ == a {1,}

- Объединения
  - a{2,}b = “aab”, “aaab”, …

…

Для создания использовать шаблон basic_regex или одну из его специализаций

```
const regex r(R"((\w+):(\w+);)");
```

Чтобы найти совпадения с объектов регулярки, использовать regex_match, regex_search. std::regex_match возвращает true только тогда, когда совпадает вся входная последовательность, в то время как std::regex_search вернет true, даже если только часть последовательности соответствует регулярному выражению.

Чтобы заменить – regex_replace

Чтобы выполнить итерацию по нескольким совпадениям использовать шаблоны regex_iterator

Свертка (C++??)

Если ссылка на ссылку возникают в контексте, где это разрешено (во время инстанцирования шаблона, генерации типа auto, создание и применение typedef и объявлений псевдонимов, и decltype), то ссылки сворачиваются в единственную ссылку согласно следующему правилу:

Если любая из ссылок является lvalue, результат представляют собой lvalue-ссылку.
В противном случае результат представляет собой rvalue-ссылку.

Designated initializers (обозначенные инициализаторы) (C++20)

Агрегат – массив или классовый тип, если он не имеет:

- закрытых незащищенных нестатических членов
- пользовательских или унаследованных конструкторов
- виртуальных закрытых защищенных базовых классов
- виртуальных функций

```
struct point {
    double x = {0};
    double y = {0};
}
…
point p{.x=3, .y{4}};
```

Такие инициализаторы позволяют при конструировании объекта структуры явно написать, какому полю присваивается какое значение

Char8_t, u8string (C++20)

```
char8_t c = u8’@’;
auto s = u8”Hello”s;
string s2 = u8”Hello”; // C++17 C++20
```

<type_traits>

Common_type. Определяет общий тип одного или нескольких типов
Conditional. Выделяет один из 2 типов в зависимости от указанного условия
Enable_if. Условно создает экземпляр типа для разрешения перегрузки SFINAE
Is_same. Проверяет совпадает ли два типа.

Range based for (C++20). Ranges. Диапазоны

На высоком уровне диапазон — это то, что вы можете выполнить итерацию. Диапазон представлен итератором, который помечает начало диапазона, и sentinel, который отмечает конец диапазона. Sentinel может иметь тот же тип, что и начальный итератор, или может отличаться. Контейнеры, такие как vector и list, в стандартной библиотеке C++ являются диапазонами. Диапазон абстрагирует итераторы таким образом, чтобы упростить и усилить возможность использования стандартной библиотеки шаблонов (STL).

С помощью диапазонов можно вызвать std::ranges::sort(myVector);, который обрабатывается так же, как если бы вы вызвали std::sort(myVector.begin(), myVector.end());

Кратко говоря, с помощью ranges визуально становится легче работать с функциями из <algorithm>, а также пропускать промежуточные этапы. Н-р, вот код для создания вектора квадратов из элементов другого вектора, делимых на три, без использования диапазонов

```
std::vector<int> input = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
std::vector<int> intermediate, output;
std::copy_if(input.begin(), input.end(), std::back_inserter(intermediate), [](const int i) { return i%3 == 0; });
    std::transform(intermediate.begin(), intermediate.end(), std::back_inserter(output), [](const int i) {return i*i; });
```

А вот код с использованием диапазонов.

```
// requires /std:c++20
std::vector<int> input = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto output = input
        | std::views::filter([](const int n) {return n % 3 == 0; })
        | std::views::transform([](const int n) {return n * n; });
```

Использование диапазонов также позволяет избежать лишнего выделения памяти.

Представления (views) – это диапазоны, которые дешево копировать и перемещать (за константное время). Обычно представление создается с помощью адаптера

Filesystem (C++17)

Библиотека <filesystem> предоставляет средства для выполнения операций с файловыми системами и их компонентами, такими как пути, обычные файлы и каталоги.

Средства библиотеки файловой системы могут быть недоступны, если иерархическая файловая система недоступна для реализации или если она не предоставляет необходимых возможностей. Некоторые функции могут быть недоступны, если они не поддерживаются базовой файловой системой. In those cases, errors must be reported.

Поведение не определено, если вызовы функций в этой библиотеке приводят к гонке файловой системы, то есть когда несколько потоков, процессов или компьютеров чередуют доступ и модификацию к одному и тому же объекту в файловой системе.

Path. Класс path хранит объект типа string_type, называемый myname здесь для целей экспозиции, подходящий для использования в качестве пути. string_type — это синоним basic_string<value_type>для , где value_type — синоним для wchar_t в Windows или char POSIX.

Итераторы

Категории итераторов:

- Output. *r, ++r, r++
- Input. !=, *, ->, ++r, r++, *r++
- Forward. i++, *i++
- Bidirectional.--a, a--, *a--
- Random access. r+=n, a + n, r -= n, a <(==, !=, >) b

Что-то про C++11

В C++11 была введена унифицированная инициализация – единый синтаксис инициализации, который может, как минимум концептуально, использоваться везде и выражать все. Чел, из чьей книги я это взяла, называет также фигурную инициализацию, при этом считает, что унифицированная инициализация – идея, а фигурная – синтаксическая конструкция. Фигурная инициализация помогает избежать сужающего преобразования

В 11 стандарте делали генерацию псевдослучайных чисел

Что-то откуда-то для чего-то

RAII объекты/классы. RAII – Resourse Acquisition is Initialization = Захват ресурса есть инициализация.