Вопросы:

1. Что делает цикл foreach (цикл, основанный на диапазоне)? Какой у него синтаксис?

foreach (или «цикл, основанный на диапазоне»), который предоставляет более простой и безопасный способ итерации по массиву (или любой другой структуре типа списка), или итерации по набору.

Синтаксис цикла foreach:

for (<объявление_элемента> : <массив>)

statement;

2. Что делает цикл for_each (цикл с функцией)? Какой у него синтаксис?

for_each — алгоритм обхода по умолчанию элементов массива. Этот алгоритм по порядку применяет заданный функциональный объект f к результату азыменования каждого итератора в диапазоне [first, last).

Для использования for_each необходимо подключить #include <algorithm>.

Конструкции:

std::for_each(begin(numbers), end(numbers), [](int number) {

// ....

});

std::for_each(begin(numbers), end(numbers), f) {

// ....

});

3. Что содержится в STL?

В языке программирования C++ термин Стандартная Библиотека означает коллекцию классов и функций, написанных на базовом языке. Стандартная Библиотека поддерживает несколько основных контейнеров, функций для работы с этими контейнерами, объектов-функций, основных типов строк и потоков (включая интерактивный и файловый ввод-вывод), поддержку некоторых языковых особенностей, и часто используемые функции для выполнения таких задач, как, например, нахождение квадратного корня числа.

Стандартная библиотека шаблонов (STL) — подмножество стандартной библиотеки C++ и содержит контейнеры, алгоритмы, итераторы, объекты-функции и т. д. Хотя некоторые программисты используют термин «STL» вместе (или попеременно) с термином «Стандартная библиотека C++».

4. Что такое пара? Как задать пару?

Класс ***pair*** (пара) стандартной библиотеки C++ позволяет нам определить одним объектом пару значений, если между ними есть какая-либо семантическая связь. Эти значения могут быть одинакового или разного типа. Для использования данного класса необходимо включить заголовочный файл <utility>.

Пример инициализации ***pair***:

std::pair <std::string, std::string> author("Alex", "Kati");

5. Какие контейнерные классы Вы знаете? С помощью каких библиотек они подключаются? Чем отличаются статические контейнеры от динамических?

Последовательные контейнеры:

- std::vector - библиотека подключения < vector >;
- std::deque - библиотека подключения < deque >;
- std::array - библиотека подключения < array >;
- std::list - библиотека подключения < list >;
- std::forward_list - библиотека подключения < forward_list >;
- std::basic_string - библиотека подключения < basic_string >.

Ассоциативные контейнеры:

- std::set - библиотека подключения <set>;
- std::multiset - библиотека подключения <set>;
- std::map - библиотека подключения < map >;
- multimap - библиотека подключения < map >;

6. Обязательный вопрос. Преподаватель даст запрос вида: «Я хочу обрабатывать свои данные с помощью контейнера, который будет позволять ...».

Подберите оптимальный контейнер по этому запросу.

Map позволяет ассоциировать элементы с чем угодно. Например, у вас есть список продуктов для покупки (или список книг в библиотеке). Вам необходимо хранить, сколько и чего нужно купить (или сколько книг с каким названием). Контейнер <map> позволяет нам хранить эти данные в виде ключ-значение (книга-количество, продукт-вес\объем...).

Итерация/перемещение по элементам массива (или какой-нибудь другой структуры) является довольно распространенным действием в программировании.

7. Что такое итератор? Какова обобщённая логика работы итератора?

итератор (iterator): обеспечивает для алгоритма средство доступа к содержимому контейнера.

Итератор — это объект, разработанный специально для перебора элементов контейнера (например, значений массива или символов в строке), обеспечивающий во время перемещения по элементам доступ к каждому из них.

Например, контейнер на основе массива может предлагать прямой итератор, который проходится по массиву в прямом порядке, и реверсивный итератор, который проходится по массиву в обратном порядке.

После того, как итератор соответствующего типа создан, программист может использовать интерфейс, предоставляемый данным итератором, для перемещения по элементам контейнера или доступа к его элементам, не беспокоясь при этом о том, какой тип перебора элементов задействован или каким образом в контейнере хранятся данные. И, поскольку итераторы в языке C++ обычно используют один и тот же интерфейс как для перемещения по элементам контейнера (оператор ++ для перехода к следующему элементу), так и для доступа (оператор * для доступа к текущему элементу) к ним, итерации можно выполнять по разнообразным типам контейнеров, используя последовательный метод.

8. Чем отличаются функции .front() и .back() от .begin() и .end()?

front() - возвращает **ссылку** на первый элемент контейнера.

begin() - возвращает **итератор** первого элемента контейнера.

back() - возвращает **ссылку** на последний элемент контейнера.

end() – возвращает **итератор** после последнего элемента контейнера.

9. Чем отличаются функции .rbegin() и .rend()от .begin() и .end()?

Фукции ***.begin()*** и ***.end()*** обходят массив от начала до конца.

Реверсивные итераторы предназначены для обхода массива с конца. Реверсивные итераторы возвращают методы ***rbegin()*** и ***rend()***.

10. Что такое стек? Какой порядок работы у стека? Какие контейнеры могут использоваться в качестве стека?

stack(стек) — это контейнерный класс, элементы которого работают по ринципу LIFO («Last In, First Out» – «Последним Пришёл, Первым Ушёл»), т.е. элементы вставляются (вталкиваются) в конец контейнера и удаляются (выталкиваются) оттудаже (из конца контейнера).

Обычно в стеках используется deque в качестве последовательного контейнера по умолчанию (что немного странно, поскольку vector был бы более подходящим вариантом), но вы также можете использовать vector или list.

11. Что такое очередь? Какой порядок работы у очереди? Какие контейнеры могут использоваться в качестве очереди?

queue (очередь) — это контейнерный класс, элементы которого работают по принципу FIFO («First In, First Out» – «Первым Пришёл, Первым Ушёл»), т.е. элементы вставляются (вталкиваются) в конец контейнера, но удаляются (выталкиваются) из начала контейнера.

По умолчанию в очереди используется deque в качестве последовательного контейнера, но также может использоваться и list.

12. Какие есть функции, помогающие работать с контейнерами как со стеками или очередями?

Существует много полезных функций, которые можно применить ко всем или почти всем видам контейнеров. Пусть у нас есть произвольный контейнер под названием test. Тогда для него потенциально могут быть применимы функции:

test.at(i) - равносильно записи test[i], но при этом, если i-ого элемента не существует, программа не вылетит. Например, test.at(0) вернёт нулевой (самый начальный) элемент контейнера.

- test.assign(n, m) - записывает в контейнер n элементов со значением m.
- test.assign(start, end) - копирует в контейнер значения другого контейнера от start до
- end. Внимание!!! start и end - интераторы на элементы другого контейнера.
- test.front() - возвращает ссылку на первый элемент контейнера.
- test.back() - возвращает ссылку на последний элемент контейнера.
- test.begin() - возвращает итератор первого элемента контейнера.
- test.end() – возвращает итератор после последнего элемента контейнера.
- test.clear() - очищает контейнер.
- test.erase(iter) или test.erase(start,end) - удаляет элемент с итератором iter или промежуток элементов с итераторами start и end.
- test.size() - возвращает количество элементов в контейнере.
- test.swap(test2) - меняет местами содержимое контейнеров test и test2
- test.insert(iter, b) - вставляет в test переменную b перед элементом с итератором iter, и возвращает итератор вставленного элемента
- test.insert(iter, n, b) - вставляет n копий b перед элементом с итератором iter.
- test.insert(iter, start, end) - копирует элементы между итераторами start и end другого контейнера и вставляет их в контейнер test перед элементом с итератором iter.

13. Что такое немодифицирующие алгоритмы? Какие немодифицирующие алгоритмы Вы знаете и что они делают (хотя бы 3)?

Немодифицирующие алгоритмы сохраняют как порядок следования обрабатываемых элементов, так и их значения. Они работают с итераторами ввода и прямыми итераторами и поэтому могут вызываться для всех стандартных контейнеров. В таблице 1 перечислены алгоритмы стандартной библиотеки STL, не изменяющие состояния контейнера.

for_each() - выполняет операцию с каждым элементом;

count() - возвращает количество элементов;

count_if() - возвращает количество элементов, удовлетворяющих заданному

критерию;

find() - ищет первый элемент с заданным значением;

search() - ищет первое вхождение подинтервала;

lexicographical_compare() - проверяет, что один интервал меньше другого по лексикографическому критерию.

14. Что такое модифицирующие алгоритмы? Какие модифицирующие алгоритмы Вы знаете и что они делают (хотя бы 3)?

модифицирующий алгоритм - присваивает возвращаемое значение последовательным элементам в выходном диапазоне.

Модифицирующие алгоритмы. Алгоритмы, такие как remove, replace, copy или sort активно производят побочные эффекты, а именно изменяют элементы последовательности. Обычно они делают это путем присваивания значений через разыменованные итераторы. copy, например, присваивает элементы входного диапазона элементам выходного диапазона. Если модифицированной последовательностью является входная последовательность, то стандарт говорит об изменяющем на месте алгоритме; если изменения влияют на выходной диапазон, то он говорит о копирующих алгоритмах. Например, replace_if является изменяющим на месте алгоритме, в то время как replace_copy_if является копирующим алгоритмом.

15. Что такое процесс, поток, сколько потоков в вашей ОС, какой функцией STL это можно посмотреть?

Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого.

Поток (thread) — это, сущность операционной системы, процесс выполнения на процессоре набора инструкций, точнее говоря программного кода. Общее назначение потоков — параллельное выполнение на процессоре двух или более различных задач.

Количество одновременных поддерживаемых потоков 4

Функция STL:

thread::hardware_concurrency();

16. Как сделать чтоб родительский поток ждал завершение дочернего потока с использованием STL?

В заголовочном файле <thread> определено пространство имен this_thread который содержит в себе функции для работы с конкретным потоком. Три из этих функций для того чтобы на некоторое время остановить выполнение потока:

- sleep_until - передается переменная класса chrono:time_point и блокируется выполнение потока пока системные часы не дойдут до этого времени;
- sleep_for - передается переменная класса chrono::duration и выполнение потока блокируется пока не прошло столько времени сколько было преданно;

- yield - останавливает выполнение потока на некоторое время предоставляя возможность выполнится другим потокам;
- get_id - и как метод класса thread возвращает id потока.

Касательно двух функций (sleep_until, sleep_for): гарантируется, что поток не будет пробужден раньше срока, который вы задали ему с помощью этих функций.

17. Что такое Id потока?

ID потока

У каждого потока есть свой уникальный номер который отличается от других потоков этой программы. Бывают моменты, когда необходимо узнать, в каком потоке исполнения вы находитесь. Вариантов может быть масса: логгирование, сравнение идентификаторов различных потоков и т.п. Для решения этой задачи стандарт предлагает решение в виде уникального идентификатора id, который ассоциирован с каждым потоком исполнения.

18. Блокировки процессов какие знаете, зачем используются?

Когда появляется возможность параллельного исполнения кода, очень часто возникает необходимость конкурентного доступа к разделяемому ресурсу. Будь то контейнер, переменная интегрального типа или умный указатель, – не важно. Важно то, что для получения предсказуемого результата от нашей операции над разделяемым ресурсом, мы должны упорядочить доступ к нему. В любой мульти-потоковой среде есть свои примитивы для выполнения этой задачи. C++ не стал исключением:

mutex - является наиболее часто используемым примитивом для защиты разделяемого ресурса. Видимо эта популярность вызвана его простотой. C++ предоставляет нам 3 типа операций над базовыми мьютексами:

- lock – если мьютекс не принадлежит никакому потоку, тогда поток, вызвавший lock, становится его обладателем. Если же некий поток уже владеет мьютексом, то текущий поток(который пытается овладеть им) блокируется до тех пор, пока мьютекс не будет освобожден и у него не появится шанса овладеть им.
- try_lock - если мьютекс не принадлежит никакому потоку, тогда поток, вызвавший try_lock, становится его обладателем и метод возвращает true. В противном случае возвращает false. try_lock не блокирует текущий поток.
- unlock – освобождает ранее захваченный мьютекс.

19. Что такое что делает lock_guard и unique_lock?

Классы, контролирующие блокировки mutex-а.

Класс std::lock_guard – это простой класс, конструктор которого вызывает метод lock для заданного объекта, а деструктор вызывает unlock. Также в конструктор класса std::lock_guard можно передать аргумент std::adopt_lock - индикатор, означающий, что mutex уже заблокирован и блокировать его заново не надо. std::lock_guard не содержит никаких других методов, его нельзя копировать, переносить или присваивать.

Класс unique_lock – это универсальная оболочка владения мьютексом, предоставляющая отсроченную блокировку, ограниченные по времени попытки блокировки, рекурсивную блокировку, передачу владения блокировкой и использование с condition variables.

20. Что такое condition_variables?

std::condition_variable - это объект синхронизации, предназначенный для блокирования одного потока, пока он не будет оповещен о наступлении некоего события из другого.

21. Что такое атомарные переменные?

Атомарные переменные часто используются для выполнения простых операций без использования мьютексов. На атомарных переменных реализуют lock-free алгоритмы и структуры данных. Стоит отметить, что они весьма сложны в разработке и отладке, и не на всех системах работают быстрее аналогичных алгоритмов и структур данных с локами.

22. Расскажите про сегменты памяти (адресное пространство) которые выделяются при создании процесса вашего приложения.

Сегменты памяти:

сегмент кода (или «текстовый сегмент»), где находится скомпилированная программа. Сегмент кода обычно доступен только для чтения;

сегмент bss (или «неинициализированный сегмент данных»), где хранятся глобальные и статические переменные, инициализированные нулем;

сегмент данных (data) (или «сегмент инициализированных данных»), где хранятся инициализированные глобальные и статические переменные;

куча (heap), откуда выделяются динамические переменные;

стек вызовов, где хранятся параметры функции, локальные переменные и другая информация, связанная с функциями.

23. В какой памяти находится код программы.

Сегмент кода Текстовый (программный) сегмент содержит машинные команды, образующие исполняемый код программы. Это часть виртуального адресного пространства объектного файла или программы, состоящая из исполняемых инструкций.

24. К какой памяти виртуальной адресной странице вашего процесса самый медленный доступ.

Сегмент кучи (или просто «куча») отслеживает память, используемую для динамического выделения.

Разместите адреса (заполните адресное пространство)(заполните следующую таблицу)для каждого примера по выделенному адресному пространству ОС для вашего процесса:

<table>
<tr><td colspan="2">Stack</td></tr>
<tr><td>Heap</td><td></td></tr>
<tr><td>bss</td><td rowspan="2">Data</td></tr>
<tr><td>data</td></tr>
<tr><td>0</td><td>text</td></tr>
</table>

1)

```
#include <iostream>

#include <stdlib.h>

const int CONSTT{3};

int g_val{4};

int g_val1;

int main () {

int v = 3;


std::cout << "\n stack is at " << &v;

std::cout << "\n heap is at " << malloc(8);

std::cout << "\n data is at " << &g_val;

std::cout << "\n bss is at " << &g_val1;

std::cout << "\n data is at " << &CONSTT;

std::cout << "\nText is at " << reinterpret_cast<void*>(main) << "\n";

}
```

2)

```
#include <iostream>

int i,j,k,l,m;

int ii(3),jj(2),kk(1),ll(5),mm(6);

int main() {

int i,j,k,l,m;

std::cout << "\n stack is at " << &i;
```

```
std::cout << "\n stack is at " << &j;
std::cout << "\n stack is at " << &k;
std::cout << "\n stack is at " << &l;
std::cout << "\n stack is at " << &m;
std::cout << "\n data is at " << &ii;
std::cout << "\n data is at " << &jj;
std::cout << "\n data is at " << &kk;
std::cout << "\n data is at " << &ll;
std::cout << "\n data is at " << &mm;
std::cout << "\n bss is at " << &::i;
std::cout << "\n bss is at " << &::j;
std::cout << "\n bss is at " << &::k;
std::cout << "\n bss is at " << &::l;
std::cout << "\n bss is at " << &::m;
std::cout << "\nText is at " << reinterpret_cast<void*>(main) << "\n";
}
```

3)

```
#include <iostream>
struct Man {
int weight;
double height;
};
int funk(int i, int j) {
return i+j;
}
int main() {
constexpr int i(0);
Man man;
Man &ref = man;
Man *pman = &man;
Man *pmand = new Man;
```

```
(*pman).weight = 70;
std::cout << "\n stack is at " << &i;

std::cout << "\n stack is at " << &man;
std::cout << "\n stack is at " << &ref;
std::cout << "\n stack is at " << &pmand;
std::cout << "\n heap is at " << pmand;
std::cout << "\n heap is at " << new Man;
std::cout << "\n 0 is at " << reinterpret_cast<void*>(funk);
std::cout << "\nText is at " << reinterpret_cast<void*>(main) << "\n";
}
```

4)

```
#include <array>
#include <vector>
struct Solanka {
int weight;
std::string s;
std::vector<int> v;
std::array<int,1> a;
};
int funk(int i, int j){
std::cout << "\nstack is at " << &i;
std::cout << "\nstack is at " << &j;
return i+j;
}
void funkv() {
std::cout << "\nheap is at " << new Solanka;
}
int main() {
int i(0);
Solanka sol;
```

```
std::cout << "\nstack is at " << &i;
std::cout << "\nstack is at " << &sol;
std::cout << "\nstack is at " << &sol.s;
sol.v.push_back(6);
std::cout << "\nstack is at " << &sol.v;
std::cout << "\nstack is at " << &sol.a;
std::cout << "\nstack is at " << &sol.s;
std::cout << "\nheap is at " << &sol.v.back();
std::cout << "\nstack is at " << &sol.a.back();
funkv();
funk(1,2);
std::cout << "\n0 is at " << reinterpret_cast<void*>(funkv);
std::cout << "\n0 is at " << reinterpret_cast<void*>(funk);
std::cout << "\nText is at " << reinterpret_cast<void*>(main) << "\n";
}
```