Оглавление

1.Базовые конструкции программирования: синтаксис и семантика языков высокого уровня; переменные, типы, выражения и присваивания; простейший ввод/вывод; 2

2.Условные предложения и итеративные конструкции 4

3.Функции и передача параметров; структурная декомпозиция. 5

4.Алгоритмы и решение задач: стратегии решения задач, роль алгоритмов в решении задач, стратегии реализации алгоритмов, стратегии отладки, понятие алгоритма, свойства алгоритмов. 7

5.Базовые структуры данных: примитивные типы; массивы; структуры; 8

7.Представление данных в памяти компьютера: биты, байты, слова; представление символьных данных. 10

8.Представление числовых данных и системы счисления; 11

9.Обзор операционных систем: роль и задачи операционных систем; простое управление файлами. 11

10.Введение в распределенные вычисления: предпосылки возникновения и история сетей и Интернета. 15

11.Человеко-машинное взаимодействие: введение в вопросы проектирования (UI/UX). 17

12.Методология разработки программного обеспечения: основные понятия и принципы проектирования; структурная декомпозиция; стратегии тестирования и отладки; разработка сценариев тестирования (test cases); 19

13.Среды разработки; инструменты тестирования и отладки. 22

14.Социальный контекст компьютинга: история компьютинга и компьютеров; эволюция идей и компьютеров; социальный эффект компьютеров и Интернета; профессионализм, кодекс этики и ответственное поведение; авторские права, интеллектуальная собственность и компьютерное пиратство. 25

15.Объектно-ориентированное программирование: объектно-ориентированное проектирование, инкапсуляция и скрытие информации; разделение интерфейса и реализации; классы, наследники и наследование; полиморфизм; иерархии классов. 30

16.Основные вычислительные алгоритмы: алгоритмы поиска и сортировки. 31

17.Основы программирования, основанного на событиях. 34

18.Введение в компьютерную графику: использование простых графических API. 34

19.Обзор языков программирования: история языков программирования; краткий обзор парадигм программирования. 35

1.Базовые конструкции программирования: синтаксис и семантика языков высокого уровня; переменные, типы, выражения и присваивания; простейший ввод/вывод;

Языки программирования – это формальные искусственные языки. Как и естественные языки, они имеют алфавит, словарный запас, грамматику и синтаксис, а также семантику.

Алфавит – разрешенный к использованию набор символов, с помощью которого могут быть образованы слова и величины данного языка.

Синтаксис – система правил, определяющих допустимые конструкции языка программирования из букв алфавита.

Семантика – система правил однозначного толкования каждой языковой конструкции, позволяющих производить процесс обработки данных.

Взаимодействие синтаксических и семантических правил определяет основные понятия языка, такие как операторы, идентификаторы, константы, переменные, функции, процедуры и т.д. В отличие от естественных, язык программирования имеет ограниченный запас слов (операторов) и строгие правила их написания, а правила грамматики и семантики, как и для любого формального языка, явно однозначно и четко сформулированы.

Языки программирования, ориентированные на команды процессора и учитывающие его особенности, называют языками низкого уровня. «Низкий уровень» не означает неразвитый, имеется в виду, что операторы этого языка близки к машинному коду и ориентированы на конкретные команды процессора.

Языком самого низкого уровня является ассемблер. Программа, написанная на нем, представляет последовательность команд машинных кодов, но записанных с помощью символьных мнемоник. В таком случае программист получает доступ ко всем возможностям процессора. Однако, при этом требуется хорошо понимать устройство компьютера, а использование такой программы на компьютере с процессором другого типа невозможно. Такие языки программирования используются для написания небольших системных приложений, драйверов устройств, модулей стыковки с нестандартным оборудованием.

Языки программирования, имитирующие естественные, обладающие укрупненными командами, ориентированные «на человека», называют языками высокого уровня. Чем выше уровень языка, тем ближе структуры данных и конструкции, использующиеся в программе, к понятиям исходной задачи. Особенности конкретных компьютерных архитектур в них не учитываются, поэтому исходные тексты программ легко переносимы на другие платформы, имеющие трансляторы этого языка. Разрабатывать программы на языках высокого уровня с помощью понятных и мощных команд значительно проще, число ошибок, допускаемых в процессе программирования, намного меньше. В настоящее время насчитывается несколько сотен таких языков (без учета их диалектов).

Программы оперируют с различными данными, которые могут быть простыми и структурированными. Простые данные - это целые и вещественные числа, символы и указатели (адреса объектов в памяти). Структурированные данные - это массивы и структуры;

Переменные - это именованные области памяти, используемые для хранения данных. Каждая переменная имеет свой тип данных, который определяет, какой вид информации может быть сохранен в данной переменной.

В языке различают понятия "тип данных" и "модификатор типа". **Тип данных** - это, например, целый, а модификатор - со знаком или без знака. Целое со знаком будет иметь как положительные, так и отрицательные значения, а целое без знака - только положительные значения. В языке Си можно выделить пять базовых типов, которые задаются следующими ключевыми словами:

- char - символьный;
- int - целый;
- float - вещественный;
- double - вещественный двойной точности;
- void - не имеющий значения.

Дадим им краткую характеристику:

1. Переменная типа char имеет размер 1 байт, ее значениями являются различные символы.
2. Размер переменной типа int в стандарте языка Си не определен. В большинстве систем программирования размер переменной типа int соответствует размеру целого машинного слова. В этом случае знаковые значения этой переменной могут лежать в диапазоне от -32768 до 32767.
3. Ключевое слово float позволяет определить переменные вещественного типа. Их значения имеют дробную часть, отделяемую точкой, например: -5.6, 31.28 и т.п. Вещественные числа

могут быть записаны также в форме с плавающей точкой, например: -1.09e+4. Число перед символом "e" называется мантиссой, а после "e" - порядком. Она может принимать значения в диапазоне от 3.4e-38 до 3.4e+38.

4. Ключевое слово double позволяет определить вещественную переменную двойной точности. Она занимает в памяти в два раза больше места, чем переменная типа float. Переменная типа double может принимать значения в диапазоне от 1.7e-308 до 1.7e+308.
5. Ключевое слово void (не имеющий значения) используется для нейтрализации значения объекта, например, для объявления функции, не возвращающей никаких значений.

Объект некоторого базового типа может быть модифицирован. С этой целью используются специальные ключевые слова, называемые модификаторами. В стандарте ANSI языка Си имеются следующие модификаторы типа:

- unsigned
- signed
- short
- long

Модификаторы записываются перед спецификаторами типа, например: unsigned char. Если после модификатора опущен спецификатор, то компилятор предполагает, что этим спецификатором является int. Таким образом, следующие строки:

```
long a;
long int a;
```

Все переменные до их использования должны быть определены (объявлены). При этом задается тип, а затем идет список из одной или более переменных этого типа, разделенных запятыми. Например:

```
int a, b, c;
char x, y;
```

Переменные в языке Си могут быть инициализированы при их определении:

```
int a = 25, h = 6;
char g = 'Q', k = 'm';
float r = 1.89;
long double n = r*123;
```

В языке возможны глобальные и локальные объекты. Первые определяются вне функций и, следовательно, доступны для любой из них. Локальные объекты по отношению к функциям являются внутренними. Они начинают существовать, при входе в функцию и уничтожаются после выхода из нее. int a; /* Определение глобальной переменной */

```
int function (int b, char c);   /* Объявление функции (т.е. описание ее заголовка)*/

void main (void)
{                       //Тело программы
        int d, e;               //Определение локальных переменных
        float f;                //Определение локальной переменной
        ...
}
```

Операции ввода/вывода в языке Си организованы посредством библиотечных функций.
Самый простой механизм ввода - чтение по одному символу из стандартного входного потока с помощью функции getchar():

```
x = getchar( );
```

присваивает переменной x очередной вводимый символ. Переменная x должна иметь символьный или целый тип.
Другая функция - putchar(x) выдает значение переменной x в стандартный выходной поток:

```
int putchar(int);
```

Объявления getchar( ) и putchar( ) сделаны в заголовочном файле stdio.h, содержащем описания заголовков библиотечных функций стандартного ввода/вывода. Подключение осуществляется с помощью директивы препроцессора:

```
#include <stdio.h>
```

Функция printf( ) обеспечивает форматированный вывод. Ее можно записать в следующем формальном виде:

```
printf ("управляющая строка", аргумент _1, аргумент _2,...);
```

Управляющая строка содержит компоненты трех типов: обычные символы, которые просто копируются в стандартный выходной поток (выводятся на экран дисплея); спецификации преобразования, каждая из которых вызывает вывод на экран очередного аргумента из последующего списка; управляющие символьные константы.
Каждая спецификация преобразования начинается со знака % и заканчивается некоторым символом, задающим преобразование. Между знаком % и символом преобразования могут встречаться другие знаки в соответствии со следующим форматом:
На месте символа преобразования могут быть записаны:

c - значением аргумента является символ;
d или **i** - значением аргумента является десятичное целое число;
e - значением аргумента является вещественное десятичное число в экспоненциальной форме вида 1.23e+2;
f - значением аргумента является вещественное десятичное число с плавающей точкой;
s - значением аргумента является строка символов (символы строки выводятся до тех пор, пока не встретится символ конца строки или же не будет, выведено число символов, заданное точностью);
u - значением аргумента является беззнаковое целое число;
p - значением аргумента является указатель;
n - применяется в операциях форматирования. Аргумент, соответствующий этому символу спецификации, должен быть указателем на целое. В него возвращается номер позиции строки (отображаемой на экране), в которой записана спецификация %n.

Функция printf( ) использует управляющую строку, чтобы определить, сколько всего аргументов и каковы их типы. Аргументами могут быть переменные, константы, выражения, вызовы функций; главное, чтобы их значения соответствовали заданной спецификации.
Функция scanf( ) обеспечивает форматированный ввод. Ее можно записать в следующем формальном виде:

```
scanf("управляющая строка", аргумент_1, аргумент_2,...);
```

Аргументы scanf( ) должны быть указателями на соответствующие значения. Для этого перед именем переменной записывается символ &. Назначение указателей будет рассмотрено далее.
Управляющая строка содержит спецификации преобразования и используется для установления количества и типов аргументов.

2.Условные предложения и итеративные конструкции

Операторы цикла
Циклы организуются, чтобы выполнить некоторый оператор или группу операторов определенное число раз. В языке Си три оператора цикла: for, while и do - while. Первый из них формально записывается, в следующем виде:

```
for (выражение_1; выражение_2; выражение_3) тело_цикла
```

Тело цикла составляет либо один оператор, либо несколько операторов, заключенных в фигурные скобки { ... } (после блока точка с запятой не ставится). В выражениях 1, 2, 3 фигурирует специальная переменная, называемая управляющей. По ее значению устанавливается необходимость повторения цикла или выхода из него.
Выражение_1 присваивает начальное значение управляющей переменной, выражение_3 изменяет его на каждом шаге, а выражение_2 проверяет, не достигло ли оно граничного значения, устанавливающего необходимость выхода из цикла.
Примеры:

```
for (i = 1; i < 10; i++)
{   ...
}

for (ch = 'a'; ch != 'p';) scanf ("%c", &ch);
/* Цикл будет выполняться до тех пор, пока с клавиатуры
не будет введен символ 'p' */
```

Любое из трех выражений в цикле for может отсутствовать, однако точка с запятой должна оставаться. Таким образом, for (; ;) {...} - это бесконечный цикл, из которого можно выйти лишь другими способами. Допускаются вложенные конструкции, т.е. в теле некоторого цикла могут встречаться другие операторы for.
Оператор while формально записывается в таком виде:

```
while (выражение) тело_цикла
```

Выражение в скобках может принимать ненулевое (истинное) или нулевое (ложное) значение. Если оно истинно, то выполняется тело цикла и выражение вычисляется снова. Если выражение ложно, то цикл while заканчивается.

Оператор do-while формально записывается следующим образом:

```
do {тело_цикла} while (выражение);
```

Основным отличием между циклами while и do - while является то, что тело в цикле do - while выполняется по крайней мере один раз. Тело цикла будет выполняться до тех пор, пока выражение в скобках не примет ложное значение. Если оно ложно при входе в цикл, то его тело выполняется ровно один раз.

Допускается вложенность одних циклов в другие, т.е. в теле любого цикла могут появляться операторы for, while и do - while.

В теле цикла могут использоваться новые операторы break и continue. Оператор break обеспечивает немедленный выход из цикла, оператор continue вызывает прекращение очередной и начало следующей итерации.

Операторы условных и безусловных переходов

Для организации условных и безусловных переходов в программе на языке Си используются операторы: if - else, switch и goto. Первый из них записывается следующим образом:

```
if (проверка_условия) оператор_1; else оператор_2;
```

Если условие в скобках принимает истинное значение, выполняется оператор_1, если ложное - оператор_2. Если вместо одного необходимо выполнить несколько операторов, то они заключаются в фигурные скобки. В операторе if слово else может отсутствовать.

В операторе if - else непосредственно после ключевых слов if и else должны следовать другие операторы. Если хотя бы один из них является оператором if, его называют вложенным. Согласно принятому в языке Си соглашению слово else всегда относится к ближайшему предшествующему ему if.

Оператор switch позволяет выбрать одну из нескольких альтернатив. Он записывается в следующем формальном виде:

```
        switch (выражение)
    {
        case константа_1:  операторы_1;
                break;

        case константа_2:  операторы_2;
                break;
      .....          .....
        default:            операторы_default;
        }
```

Здесь вычисляется значение целого выражения в скобках (его иногда называют селектором) и оно сравнивается со всеми константами (константными выражениями). Все константы должны быть различными. При совпадении выполнится соответствующий вариант операторов (один или несколько операторов). Вариант с ключевым словом default реализуется, если ни один другой не подошел (слово default может и отсутствовать). Если default отсутствует, а все результаты сравнения отрицательны, то ни один вариант не выполняется.

Для прекращения последующих проверок после успешного выбора некоторого варианта используется оператор break, обеспечивающий немедленный выход из переключателя switch.

Допускаются вложенные конструкции switch.

Рассмотрим правила выполнения безусловного перехода, который можно представить в следующей форме:

```
goto метка;
```

Метка - это любой идентификатор, после которого поставлено двоеточие. Оператор goto указывает на то, что выполнение программы необходимо продолжить начиная с оператора, перед которым записана метка. Метку можно поставить перед любым оператором в той функции, где находится соответствующий ей оператор goto. Ее не надо объявлять.

3.Функции и передача параметров; структурная декомпозиция.

Программы на языке Си обычно состоят из большого числа отдельных функций (подпрограмм). Как правило, эти функции имеют небольшие размеры и могут находиться как в одном, так и в нескольких файлах. Все функции являются глобальными. В языке запрещено определять одну функцию внутри другой. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные.

В общем случае функции в языке Си необходимо объявлять. Объявление функции (т.е. описание заголовка) должно предшествовать ее использованию, а определение функции (т.е. полное описание) может быть помещено как после тела программы (т.е. функции main( )), так и до него. Если функция определена до тела

программы, а также до ее вызовов из определений других функций, то объявление может отсутствовать. Как уже отмечалось, описание заголовка функции обычно называют прототипом функции.
Функция объявляется следующим образом:

```
тип имя_функции(тип имя_параметра_1, тип имя_параметра_2, ...);
```

Тип функции определяет тип значения, которое возвращает функция. Если тип не указан, то предполагается, что функция возвращает целое значение (int).
При объявлении функции для каждого ее параметра можно указать только его тип (например: тип функция (int, float, ...), а можно дать и его имя (например: тип функция (int a, float b, ...)).
В языке Си разрешается создавать функции с переменным числом параметров. Тогда при задании прототипа вместо последнего из них указывается многоточие.
Определение функции имеет следующий вид:

```
тип имя_функции(тип имя_параметра_1, тип имя_параметра_2,...)
{
тело функции
}
```

Передача значения из вызванной функции в вызвавшую происходит с помощью оператора возврата return, который записывается следующим образом:

```
return выражение;
```

Таких операторов в подпрограмме может быть несколько, и тогда они фиксируют соответствующие точки выхода. Например:

```
int f(int a, int b)
{
if (a > b) { printf("max = %d\n", a); return a; }
printf("max = %d\n", b); return b;
}
```

Вызвать эту функцию можно следующим образом:

```
c = f(15, 5);
c = f(d, g);
f(d, g);
```

Вызвавшая функция может, при необходимости, игнорировать возвращаемое значение. После слова return можно ничего не записывать; в этом случае вызвавшей функции никакого значения не передается. Управление передается вызвавшей функции и в случае выхода "по концу" (последняя закрывающая фигурная скобка).
В языке Си аргументы функции передаются по значению, т.е. вызванная функция получает свою временную копию каждого аргумента, а не его адрес. Это означает, что вызванная функция не может изменить значение переменной вызвавшей ее программы. Однако это легко сделать, если передавать в функцию не переменные, а их адреса. Например:

```
void swap(int *a, int *b)
{
int *tmp = *a;

*a = *b;
*b = *tmp;
}
```

Вызов swap(&b, &c) (здесь подпрограмме передаются адреса переменных b и c) приведет к тому, что значения переменных b и c поменяются местами.
Если же в качестве аргумента функции используется имя массива, то передается только адрес начала массива, а сами элементы не копируются. Функция может изменять элементы массива, сдвигаясь (индексированием) от его начала.

Аргументы функции main( )

В программы на языке Си можно передавать некоторые аргументы. Когда вначале вычислений производится обращение к main( ), ей передаются три параметра. Первый из них определяет число командных аргументов при обращении к программе. Второй представляет собой массив указателей на символьные строки, содержащие эти аргументы (в одной строке - один аргумент). Третий тоже является массивом указателей на символьные строки, он используется для доступа к параметрам операционной системы (к переменным окружения).
Любая такая строка представляется в виде:

```
переменная = значение\0
```

Библиотечные функции

В системах программирования подпрограммы для решения часто встречающихся задач объединяются в библиотеки. К числу таких задач относятся: вычисление математических функций, ввод/вывод данных, обработка строк, взаимодействие со средствами операционной системы и др. Использование библиотечных подпрограмм избавляет пользователя от необходимости разработки соответствующих средств и

предоставляет ему дополнительный сервис. Включенные в библиотеки функции поставляются вместе с системой программирования. Их объявления даны в файлах *.h (это так называемые включаемые или заголовочные файлы).
#include <включаемый_файл_типа_h>
Существуют также средства для расширения и создания новых библиотек с программами пользователя.
Структурная декомпозиция в языке программирования С может быть выполнена с использованием структур или функций.
Использование структур позволяет группировать связанные данные в одну единицу. Примером структуры может служить структура, представляющая информацию о сотруднике:

```
struct Employee {
char name[50];
int age;
double salary;
};
```

С использованием функций можно разбить программу на отдельные модули, каждый из которых отвечает за определенную функцию или логический блок процесса.
Для более сложных программ структурная декомпозиция может включать разбиение на модули, каждый из которых содержит несколько функций и структур.
В целом, структурная декомпозиция в языке С позволяет разбивать программу на более мелкие, понятные и легко поддерживаемые части.

Структурная декомпозиция может быть различной в зависимости от контекста, в котором она применяется. Однако, наиболее распространенными видами структурной декомпозиции являются:

Функциональная декомпозиция - процесс разбиения сложной системы на более простые и понятные функциональные блоки. Каждый блок выполняет определенную функцию и может быть дальше декомпозирован на более низкие уровни.

Иерархическая декомпозиция - разделение системы на иерархические уровни или уровни подробности. Каждый уровень представляет собой подсистему, состоящую из более низкоуровневых подсистем. Это позволяет управлять сложностью системы и облегчает понимание и анализ ее компонентов.

Объектно-ориентированная декомпозиция - разделение системы на объекты и классы, отражающие структуру и поведение системы. Каждый объект или класс выполняет определенную функцию и взаимодействует с другими объектами или классами, образуя систему.

Параллельная декомпозиция - разделение системы на параллельные процессы или потоки выполнения. Это позволяет распределить вычислительную нагрузку и повысить производительность системы.

Ресурсная декомпозиция - разделение системы на ресурсы, такие как оборудование, материалы или персонал. Это помогает управлять ресурсами и оптимизировать их использование.

4.Алгоритмы и решение задач: стратегии решения задач, роль алгоритмов в решении задач, стратегии реализации алгоритмов, стратегии отладки, понятие алгоритма, свойства алгоритмов.

Алгоритмы в информатике используются для решения различных задач, от простых до сложных. Они представляют собой последовательность шагов, которые необходимо выполнить для достижения определенной цели.

Стратегии решения задач определяют подходы к решению задачи и помогают разработать эффективный алгоритм. Некоторые из таких стратегий могут включать пошаговое разбиение задачи на подзадачи, использование циклов и условных операторов, использование рекурсии и т.д. Каждая задача может требовать своей собственной стратегии решения.

Роль алгоритмов в решении задач заключается в том, что они предлагают формальное и структурированное решение задачи. Алгоритмы позволяют автоматизировать процесс решения задачи, упрощают его и облегчают его понимание.

Стратегии реализации алгоритмов определяют методы и техники, используемые при создании кода для реализации алгоритма. Эти стратегии включают выбор подходящего языка программирования, определение структуры данных, выбор алгоритмических методов и т.д.

Стратегии отладки помогают в поиске и исправлении ошибок в алгоритмах и коде. Они включают в себя использование отладочных инструментов, тестирование на различных сценариях, анализ кода и т.д.

Алгоритм - это точная и последовательная спецификация процесса решения задачи. Он определяет шаги, которые необходимо выполнить для достижения желаемого результата. Алгоритм должен быть понятным, ясным и исполняемым.

Свойства алгоритмов в С включают корректность, т.е. алгоритм должен быть правильным и давать правильные результаты для всех возможных входных данных. Также алгоритм должен быть эффективным, т.е. выполняться за разумное время и использовать разумное количество памяти. Другим свойством алгоритма является указанность, т.е. каждый шаг должен быть ясно определен и понятен. Помимо этих свойств, алгоритм должен быть выполнимым, т.е. можно реализовать его в конкретном языке программирования.

В целом, алгоритмы играют ключевую роль в решении задач, предлагая структурированный и понятный подход к решению сложных задач. Они помогают автоматизировать процесс решения задачи и повышают эффективность программного решения.

5.Базовые структуры данных: примитивные типы; массивы; структуры;

В языке Си можно выделить пять **базовых типов**, которые задаются следующими ключевыми словами:

- char - символьный;
- int - целый;
- float - вещественный;
- double - вещественный двойной точности;
- void - не имеющий значения.

Дадим им краткую характеристику:

Переменная типа char имеет размер 1 байт, ее значениями являются различные символы из кодовой таблицы.

Размер переменной типа int в стандарте языка Си не определен. В большинстве систем программирования размер переменной типа int соответствует размеру целого машинного слова. В этом случае знаковые значения этой переменной могут лежать в диапазоне от -32768 до 32767.

Ключевое слово float позволяет определить переменные вещественного типа. Их значения имеют дробную часть, отделяемую точкой, например: -5.6, 31.28 и т.п. Вещественные числа могут быть записаны также в форме с плавающей точкой, например: -1.09e+4. Число перед символом "e" называется мантиссой, а после "e" - порядком. Она может принимать значения в диапазоне от 3.4e-38 до 3.4e+38.

Ключевое слово double позволяет определить вещественную переменную двойной точности. Она занимает в памяти в два раза больше места, чем переменная типа float (т.е. ее размер 64 бита). Переменная типа double может принимать значения в диапазоне от 1.7e-308 до 1.7e+308.

Ключевое слово void (не имеющий значения) используется для нейтрализации значения объекта, например, для объявления функции, не возвращающей никаких значений.

Массивы

Массив состоит из элементов одного и того же типа. Ко всему массиву целиком можно обращаться по имени. Кроме того, можно выбирать любой элемент массива. Для этого необходимо задать индекс, который указывает на его относительную позицию. Число элементов массива назначается при его определении и в дальнейшем не изменяется. Если массив объявлен, то к любому его элементу можно обратиться следующим образом: указать имя массива и индекс элемента в квадратных скобках. Массивы определяются так же, как и переменные:

```
int a[100];
char b[20];
float d[50];
```

В первой строке объявлен массив a из 100 элементов целого типа: a[0], a[1], ..., a[99] (индексация всегда начинается с нуля). Во второй строке элементы массива b имеют тип char, а в третьей - float.

Двумерный массив представляется как одномерный, элементами которого также являются массивы. Например, определение char a[10][20]; задает такой массив. По аналогии можно установить и большее число измерений. Элементы двумерного массива хранятся по строкам, т.е. если проходить по ним в порядке их расположения в памяти, то быстрее всего изменяется самый правый индекс. Например, обращение к девятому элементу пятой строки запишется так: a[5][9].

Пусть задан массив:

```
int a[2][3];
```

Тогда элементы массива a будут размещаться в памяти следующим образом: a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2].

В языке Си существует сильная взаимосвязь между указателями и массивами. Любое действие, которое достигается индексированием массива, можно выполнить и с помощью указателей, причем последний вариант будет работать быстрее.

Определение

```
int a[5];
```

задает массив из пяти элементов a[0], a[1], a[2], a[3], a[4]. Если объект *y определен как

```
int *y;
```
то оператор y = &a[0]; присваивает переменной y адрес элемента a[0]. Если переменная y указывает на очередной элемент массива a, то y+1 указывает на следующий элемент, причем здесь выполняется соответствующее масштабирование для приращения адреса с учетом длины объекта.
Так как само имя массива есть адрес его нулевого элемента, то оператор y = &a[0]; можно записать и в другом виде: y = a. Тогда элемент a[1] можно представить как *(a+1). С другой стороны, если y - указатель на массив a, то следующие две записи: a[i] и *(y+i) - эквивалентны.
Язык Си позволяет инициализировать массив при его определении. Для этого используется следующая форма:
```
тип имя_массива[...] ... [...] = {список значений};
```
Примеры:
```
int a[5] = {0, 1, 2, 3, 4};
char ch[3] = {'d', 'e', '9'};
int b[2][3] = {1, 2, 3, 4, 5, 6};
```
Структура - это объединение одного или нескольких объектов (переменных, массивов, указателей, других структур и т.д.). Как и массив, она представляет собой совокупность данных. Отличием является то, что к ее элементам необходимо обращаться по имени и что различные элементы структуры не обязательно должны принадлежать одному типу.
Объявление структуры осуществляется с помощью ключевого слова struct, за которым идет ее тип и далее список элементов, заключенных в фигурные скобки:
```
struct тип { тип элемента_1 имя элемента_1;
     .....
тип элемента_n имя элемента_n;
};
```
Именем элемента может быть любой идентификатор. Как и выше, в одной строке можно записывать через запятую несколько идентификаторов одного типа.
Рассмотрим пример:
```
sruct date { int day;
   int month;
int year;
};
```
Следом за фигурной скобкой, заканчивающей список элементов, могут записываться переменные данного типа, например:
```
struct date {...} a, b, c;
```
(при этом выделяется соответствующая память). Описание без последующего списка не выделяет никакой памяти; оно просто задает форму структуры. Введенное имя типа позже можно использовать для объявления структуры, например:
```
struct date days;
```
Теперь переменная days имеет тип date.
При необходимости структуры можно инициализировать, помещая вслед за описанием список начальных значений элементов.
Разрешается вкладывать структуры друг в друга, например:
```
struct man { char name[20], fam[20];
struct date bd;
int age;
};
```
Определенный выше тип data включает три элемента: day, month, year, содержащий целые значения (int). Структура man включает элементы name, fam, bd и voz. Первые два - name[20] и fam[20] - это символьные массивы из 20 элементов каждый. Переменная bd представлена составным элементом

6.Базовые структуры данных: строки и операции над строками.

Язык Си не поддерживает отдельный строковый тип данных, но он позволяет определить строки двумя различными способами. В первом используется массив символов, а во втором - указатель на первый символ массива.
Определение char a[10]; указывает компилятору на необходимость резервирования места для максимум 10 символов. Константа a содержит адрес ячейки памяти, в которой помещено значение первого из десяти объектов типа char. Процедуры, связанные с занесением конкретной строки в массив a, копируют ее по одному символу в область памяти, на которую указывает константа a, до тех пор, пока не будет скопирован нулевой символ, оканчивающий строку. Когда выполняется функция типа printf("%s", a), ей передается значение a, т.е. адрес первого символа, на который указывает a. Если первый символ - нулевой, то работа функции printf() заканчивается, а если нет, то она выводит его на экран, прибавляет к адресу единицу и

снова начинает проверку на нулевой символ. Такая обработка позволяет снять ограничения на длину строки: строка может иметь любую длину, но в пределах доступной памяти.
Инициализировать строку при таком способе определения можно следующим образом:

```
char array[7] = "Строка";
char s[ ] = {'С', 'т', 'р', 'о', 'к', 'а', '\0'};
```

Для ввода и вывода строк символов помимо scanf( ) и printf() могут использоваться функции gets( ) и puts( ) (их прототипы находятся в файле stdio.h).
Если string - массив символов, то ввести строку с клавиатуры можно так:

```
gets(string);
```

(ввод оканчивается нажатием клавиши <Enter>). Вывести строку на экран можно следующим образом:

```
puts(string);
```

Отметим также, что для работы со строками существует специальная библиотека функций, прототипы которых находятся в файле string.h.
Наиболее часто используются функции strcpy( ), strcat( ), strlen( ) и strcmp( ).
Если string1 и string2 - массивы символов, то вызов функции strcpy( ) имеет вид:

```
strcpy(string1, string2);
```

Эта функция служит для копирования содержимого строки string2 в строку string1. Массив string1 должен быть достаточно большим, чтобы в него поместилась строка string2. Так как компилятор не отслеживает этой ситуации, то недостаток места приведет к потере данных.
Вызов функции strcat( ) имеет вид:

```
strcat(string1, string2);
```

Эта функция присоединяет строку string2 к строке string1 и помещает ее в массив, где находилась строка string1, при этом строка string2 не изменяется. Нулевой байт, который завершал первую строку, заменяется первым байтом второй строки.
Функция strlen( ) возвращает длину строки, при этом завершающий нулевой байт не учитывается. Если a - целое, то вызов функции имеет вид:

```
a = strlen(string);
```

Функция strcmp( ) сравнивает две строки и возвращает 0, если они равны.
Асимптотическая сложность операций работы со строками в языке С зависит от конкретной операции. Вот несколько примеров:

1. Длина строки: получение длины строки в С имеет линейную асимптотическую сложность O(n), где n - длина строки.
2. Копирование строки: копирование строки в С вызывает линейную асимптотическую сложность O(n), где n - длина строки.
3. Сравнение строк: сравнение строк в С имеет линейную асимптотическую сложность O(n), где n - длина строк.
4. Объединение строк: объединение строк в С может быть выполнено за линейное время O(n), где n - суммарная длина строк.
5. Поиск подстроки: поиск подстроки в С имеет линейно-логарифмическую асимптотическую сложность O(n * log(m)), где n - длина исходной строки, m - длина искомой подстроки.

Реальное время выполнения операций может также зависеть от конкретной реализации функций в библиотеке языка С или от особенностей системы выполнения кода.

7.Представление данных в памяти компьютера: биты, байты, слова; представление символьных данных.

Представление данных в памяти компьютера осуществляется с помощью битов, байтов, слов и символов.

Бит (binary digit) является наименьшей единицей данных в компьютерной системе и может принимать значение 0 или 1. Биты группируются в байты.

Байт (byte) представляет собой последовательность из 8 битов и является базовой единицей хранения информации в компьютерах. Байты используются для хранения чисел, символов и других данных.

Слово (word) представляет собой фиксированное количество байтов, которое обычно является наиболее эффективным для обработки компьютером. Слова широко используются в процессорах и позволяют выполнять операции над данными определенного размера.

Для представления символьных данных в компьютерах применяются различные кодировки, такие как ASCII (American Standard Code for Information Interchange) и Unicode. ASCII использует 7 бит для представления каждого символа, в то время как Unicode использует одно или несколько байтов для представления каждого символа, позволяя учитывать большое количество символов из разных письменностей.

В зависимости от системы и применяемой кодировки, символы могут занимать разное количество байтов в памяти компьютера. Например, в кодировке ASCII латинские буквы будут занимать 1 байт, а в кодировке UTF-8 каждый символ занимает от 1 до 4 байтов в зависимости от его кода.

8.Представление числовых данных и системы счисления;

В языке программирования C представление числовых данных осуществляется с использованием различных типов данных. Некоторые из основных типов данных, используемых для представления чисел, включают в себя:

1. int: представляет целые числа со знаком. Размер этого типа данных обычно зависит от архитектуры компьютера, но он обычно равен 4 байтам.
2. float: представляет вещественные числа с одинарной точностью. Этот тип данных используется для представления чисел с десятичными дробями. Размер этого типа данных обычно равен 4 байтам.
3. double: представляет вещественные числа с двойной точностью. Этот тип данных используется для представления чисел с более высокой точностью, чем тип float. Размер этого типа данных обычно равен 8 байтам.
4. char: представляет одиночные символы. Этот тип данных может быть использован для представления чисел в диапазоне от -128 до 127, используя таблицу кодировки ASCII.
5. unsigned int: представляет целые числа без знака. Размер этого типа данных также зависит от архитектуры компьютера, но он обычно равен 4 байтам.

В C также можно использовать различные системы счисления для представления чисел, включая:

Десятичная система счисления: основание этой системы равно 10. В десятичной системе используются цифры от 0 до 9.

Шестнадцатеричная система счисления: основание этой системы равно 16. В шестнадцатеричной системе используются цифры от 0 до 9 и буквы A, B, C, D, E, F для представления чисел от 10 до 15.

Восьмеричная система счисления: основание этой системы равно 8. В восьмеричной системе используются цифры от 0 до 7.

В C можно использовать специальные префиксы для указания системы счисления числа. Например, префикс "0x" указывает на шестнадцатеричное число, а префикс "0" указывает на восьмеричное число. Например:

```
int decimal = 10; // десятичное число
int hexadecimal = 0x10; // шестнадцатеричное число (16 в десятичной системе)
int octal = 010; // восьмеричное число (8 в десятичной системе)
```

Таким образом, в языке программирования C представление числовых данных и системы счисления осуществляется с помощью различных типов данных и специальных префиксов для указания системы счисления числа.

9.Обзор операционных систем: роль и задачи операционных систем; простое управление файлами.

Операционная система (сокр. ОС) представляет собой совокупность взаимосвязанных программ, предназначенных для управления ресурсами компьютера, ноутбука или смартфона. Таким образом, главная задача ОС – управление всеми элементами девайса. С помощью нее человек может взаимодействовать со своим оборудованием. Кроме того, операционная система позволяет правильно распределять вычислительные ресурсы между процессами.

Благодаря операционной системе разработчики программного обеспечения (ПО) могут пользоваться удобным интерфейсом и с помощью этого создавать различные программы. При этом стоит понимать, что программы разрабатываются строго под конкретную ОС.

В большей части устройств ОС выступает в качестве самого важного элемента ПО. Причем операционные системы имеют разный набор функций и ограничений. Но некоторые типы ОС дают возможность по собственному желанию увеличивать функционал своего устройства при помощи установки всевозможных программ.

Наиболее важный элемент ОС – это ядро. Оно осуществляет контроль над правильным выполнением процессов и регулирует имеющиеся у устройства ресурсы. Например, когда пользователь взаимодействует с компьютером, в нем запускаются процессы, для которых, конечно же, требуются определенные ресурсы, а доступ к ним невозможен без отлаженной работы ОС.

ОС выполняет две основные задачи, которые и определяют ее предназначение:

- Управляет всеми ресурсами системы. Операционная система обеспечивает функционирование и правильную координацию процессов устройства;
- Упрощает для пользователя работу с устройством.

ОС позволяет эффективно взаимодействовать со всевозможными девайсами и использовать различные приложения.
Функции операционных систем определяются разработчиками и зависят от самих комплектующих устройства, но можно выделить ряд свойств, которые присущи всем ОС:

- выполнение запросов ПО;
- работа с программами и загрузка их в оперативную память;
- обеспечение многозадачности и надежности вычислительных процессов;
- стандартизированный доступ к устройствам ввода-вывода;
- контроль над процессором, видеоадаптером, оперативной памятью и другими элементами девайса;
- отладка и логирование ошибок;
- предоставление удобного интерфейса;
- правильная координация ресурсов устройства и их распределение между запущенными процессами.

Существует несколько классификаций ОС.
В зависимости от типа ядра:

- ОС с монолитным ядром;
- ОС с микроядром;
- ОС с гибридным ядром.

В зависимости от количества единовременно решаемых задач:

- однозадачные;
- многозадачные;

В зависимости от количества пользователей:

- однопользовательские;
- многопользовательские.

В зависимости от количества поддерживаемых процессоров:

- однопроцессорные
- многопроцессорные

В зависимости от возможности работы в компьютерной сети:

- локальные – автономные ОС, которые не позволяют работать с компьютерными сетями;
- сетевые – ОС с поддержкой компьютерных сетей.

В зависимости от роли в сетевом взаимодействии:

- серверные – ОС, открывающие доступ к ресурсам сети и осуществляющие управление сетевой инфраструктурой;
- клиентские – ОС, которые имеют возможность получения доступа к ресурсам сети.

В зависимости от типа лицензии:

- открытые – ОС с открытым исходным кодом, который можно изучать и редактировать;
- проприетарные – ОС, связанные с определенным правообладателем и, как правило, имеющие закрытый исходный код.

В зависимости от сферы использования:

- ОС мэйнфреймов – больших компьютеров;
- ОС серверов;
- ОС персональных компьютеров;
- ОС мобильных устройств;
- встроенные ОС;
- ОС маршрутизаторов.

Файл – именованная область внешней памяти, предназначенная для считывания и записи данных.
Файлы хранятся в памяти, не зависящей от энергопитания. Исключением является электронный диск, когда в ОП создается структура, имитирующая файловую систему.
Файловая система (ФС) — это компонент ОС, обеспечивающий организацию создания, хранения и доступа к именованным наборам данных — файлам.
Файловая система включает:

- Совокупность всех файлов на диске.
- Наборы структур данных, используемых для управления файлами (каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске).
- Комплекс системных программных средств, реализующих различные операции над файлами: создание, уничтожение, чтение, запись, именование, поиск.

Задачи, решаемые ФС, зависят от способа организации вычислительного процесса в целом. Самый простой тип – это ФС в однопользовательских и однопрограммных ОС. Основные функции в такой ФС нацелены на решение следующих задач:

- Именование файлов.
- Программный интерфейс для приложений.
- Отображения логической модели ФС на физическую организацию хранилища данных.
- Устойчивость ФС к сбоям питания, ошибкам аппаратных и программных средств.

Задачи ФС усложняются в однопользовательских многозадачных ОС, которые предназначены для работы одного пользователя, но дают возможность запускать одновременно несколько процессов. К перечисленным выше задачам добавляется новая задача — совместный доступ к файлу из нескольких процессов.
Файл в этом случае является разделяемым ресурсом, а значит ФС должна решать весь комплекс проблем, связанных с такими ресурсами. В частности: должны быть предусмотрены средства блокировки файла и его частей, согласование копий, предотвращение гонок, исключение тупиков. В многопользовательских системах появляется еще одна задача: Защита файлов одного пользователя от несанкционированного доступа другого пользователя.
Еще более сложными становятся функции ФС, которая работает в составе сетевой ОС ей необходимо организовать защиту файлов одного пользователя от несанкционированного доступа другого пользователя.
Основное назначение файловой системы и соответствующей ей системы управления файлами– организация удобного управления файлами, организованными как файлы: вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной нам записи, используется логический доступ с указанием имени файла и записи в нем.
Термины «файловая система» и «система управления файлами» необходимо различать: файловая система определяет, прежде всего, принципы доступа к данным, организованным как файлы. А термин «система управления файлами» следует употреблять по отношению к конкретной реализации файловой системы, т.е. это комплекс программных модулей, обеспечивающих работу с файлами в конкретной ОС.
Типы файлов
Обычные файлы: содержат информацию произвольного характера, которую заносит в них пользователь или которая образуется в результате работы системных и пользовательских программ. Содержание обычного файла определяется приложением, которое с ним работает.
Обычные файлы могут быть двух типов:

1. **Программные** (исполняемые) – представляют собой программы, написанные на командном языке ОС, и выполняют некоторые системные функции.
2. **Файлы данных** – все прочие типы файлов: текстовые и графические документы, электронные таблицы, базы данных и др.

Каталоги – это, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений, а с другой стороны – это особый тип файлов, которые содержат системную справочную информацию о наборе файлов, сгруппированных пользователями по какому-либо неформальному признаку.
Специальные файлы – это фиктивные файлы, ассоциированные с устройствами ввода/вывода, которые используются для унификации механизма доступа к файлам и внешним устройствам. Специальные файлы позволяют пользователю осуществлять операции ввода/вывода посредством обычных команд записи с файлов или чтения из файлов.

Файловая структура – вся совокупность файлов на диске и взаимосвязей между ними.
Виды файловых структур:

- **простая**, или **одноуровневая**: каталог представляет собой линейную последовательность файлов.
- **иерархическая** или **многоуровневая**: каталог сам может входить в состав другого каталога и содержать внутри себя множество файлов и подкаталогов. Иерархическая структура может быть двух видов: «Дерево» и «Сеть». Каталоги образуют «Дерево», если файлу разрешено входить только в один каталог (ОС MS-DOS, Windows) и «Сеть» – если файл может входить сразу в несколько каталогов (UNIX).

Файлы идентифицируются именами. Пользователи дают файлам **символьные имена**, при этом учитываются ограничения ОС как на используемые символы, так и на длину имени. В ранних файловых системах эти границы были весьма узкими. Так в популярной **файловой системе FAT** длина имен ограничивается известной схемой 8.3 (8 символов — собственно имя, 3 символа — расширение имени), а в ОС UNIX System V имя не может содержать более 14 символов.
Однако пользователю гораздо удобнее работать с длинными именами, поскольку они позволяют дать файлу действительно мнемоническое название, по которому даже через достаточно большой промежуток времени можно будет вспомнить, что содержит этот файл. Поэтому современные файловые системы, как правило, поддерживают длинные символьные имена файлов.
При переходе к длинным именам возникает проблема совместимости с ранее созданными приложениями, использующими короткие имена. Чтобы приложения могли обращаться к файлам в соответствии с принятыми ранее соглашениями, файловая система должна уметь предоставлять эквивалентные короткие имена (псевдонимы) файлам, имеющим длинные имена. Таким образом, одной из важных задач становится проблема генерации соответствующих коротких имен.
Символьные имена могут быть трех типов: простые, составные и относительные:

1. **Простое имя** идентифицирует файл в пределах одного каталога, присваивается файлам с учетом номенклатуры символа и длины имени.
2. **Полное имя** представляет собой цепочку простых символьных имен всех каталогов, через которые проходит путь от корня до данного файла, имени диска, имени файла. Таким образом, полное имя является **составным**, в котором простые имена отделены друг от друга принятым в ОС разделителем.
3. Файл может быть идентифицирован также **относительным именем**. Относительное имя файла определяется через понятие «текущий каталог». В каждый момент времени один из каталогов является текущим, причем этот каталог выбирается самим пользователем по команде ОС. Файловая система фиксирует имя текущего каталога, чтобы затем использовать его как дополнение к относительным именам для образования полного имени файла.

Атрибуты – это информация, описывающая свойства файлов. Примеры возможных атрибутов файлов:

- Признак «только для чтения» (Read-Only);
- Признак «скрытый файл» (Hidden);
- Признак «системный файл» (System);
- Признак «архивный файл» (Archive);
- Тип файла (обычный файл, каталог, специальный файл);
- Владелец файла;
- Создатель файла;
- Пароль для доступа к файлу;
- Информация о разрешенных операциях доступа к файлу;
- Время создания, последнего доступа и последнего изменения;
- Текущий размер файла;
- Максимальный размер файла;
- Признак «временный (удалить после завершения процесса)»;
- Признак блокировки.

В файловых системах разного типа для характеристики файлов могут использоваться разные наборы атрибутов (например, в однопользовательской ОС в наборе атрибутов будут отсутствовать характеристики, имеющие отношение к пользователю и защите (создатель файла, пароль для доступа к файлу и т.д.).
Пользователь может получать доступ к атрибутам, используя средства, предоставленные для этих целей файловой системой. Обычно разрешается читать значения любых атрибутов, а изменять – только некоторые, например можно изменить права доступа к файлу, но нельзя изменить дату создания или текущий размер файла.
Права доступа к файлу
Определить права доступа к файлу — значит определить для каждого пользователя набор операций, которые он может применить к данному файлу. В разных файловых системах может быть определен свой список дифференцируемых операций доступа. Этот список может включать следующие операции:

- создание файла.
- уничтожение файла.
- запись в файл.
- открытие файла.
- закрытие файла.
- чтение из файла.
- дополнение файла.
- поиск в файле.
- получение атрибутов файла.
- установление новых значений атрибутов.
- переименование.
- выполнение файла.
- чтение каталога и др.

В самом общем случае **права доступа** могут быть описаны матрицей прав доступа, в которой столбцы соответствуют всем файлам системы, строки — всем пользователям, а на пересечении строк и столбцов указываются разрешенные операции:

Имена пользователей	Имена файлов			
	dok.txt	prog.exe	baza.dbf	unix.ppt
User1	*читать*	*выполнять*	–	*выполнять*
User2	*читать*	*выполнять*	–	*выполнять* *читать*
User3	*читать*	–	–	*выполнять* *читать*
User4	*читать* *писать*	–	*создать*	–

В некоторых системах пользователи могут быть разделены на отдельные категории. Для всех пользователей одной категории определяются единые права доступа, например в системе UNIX все пользователи подразделяются на три категории: владельца файла, членов его группы и всех остальных.

10.Введение в распределенные вычисления: предпосылки возникновения и история сетей и Интернета.

Как и многие другие передовые технологии, интернет в его первичной форме зародился в военных штабах Америки. Холодная война грозила США множеством бед, в том числе, и запуском межконтинентальных ядерных ракет. Советские технологии внушали страх и трепет, а запуск первого автономного летательного аппарата в 1957 году не только подарил всему англоязычному миру неологизм «sputnik», но и заставил оборонную промышленность работать на опережение.
Одним из следствий этой работы стало создание ARPA, агентства перспективных исследовательских проектов. В компетенции агентства была разработка технологий, которые могли бы дать США весомое преимущество в Холодной войне и возглавить гонку вооружений, а в случае открытой конфронтации — помочь в защите страны.

К 1962 году агентством были сформированы первые тезисы о некой взаимосвязанной сетевой системе, которую Дж.К.Р. Ликлайдер из Массачусетского технологического института назвал «галактической сетью». Вкратце, галактическая сеть предполагала мгновенное получение доступа к любой информации в электронном виде, находящейся на множестве взаимоудаленных компьютеров. Теперь требовалось понять, как именно эти компьютеры будут связаны.

В то же самое время, благодаря исследованиям ВВС США, заказанным с целью определить, каким образом вооруженные силы смогут сохранять командование в случае ядерного удара, к «галактической сети» прибавился еще один кирпичик будущего интернета: технология коммутации пакетов (packet switching).

Идея коммутации пакетов состоит в том, что пакет данных, содержащий в себе информацию как об источнике, так и о конечном пункте «путешествия», может быть отправлен из одного местоположения в другое. Если на каком-то участке пути пакет терялся, отправитель мог с легкостью отправить его снова. Для телефонных сетей того времени это было крайне полезной находкой.

Поскольку США требовалось построить децентрализованную модель командования, при которой, независимо от степени нанесенного врагом ущерба, ВС не теряли бы контроль над собственным оружием, самолетами и бомбардировщиками и могли продолжать оборону, внедрение технологии коммутации пакетов решало эту задачу.

В 1968 году в США началось строительство ARPANET, сети-предшественника современного интернета, которой было суждено стать плацдармом для обкатки множества технологий, которые мы используем и по сей день.

В 1969 году, 29 октября, состоялся первый успешный сеанс связи между двумя компьютерами. Первый находился в Массачусетсе, второй – в Калифорнии, оба принадлежали местным техническим университетам. Расстояние между компьютерами составляло порядка 640 км.
В 21:00 по местному времени Чарли Клайн попытался подключиться к компьютеру Стэнфорда и передать 5 символов, слово ”LOGIN”, однако сеть оборвалась, и удалось ввести только два первых символа. В 22:30 соединение было восстановлено, и вторая попытка увенчалась успехом. Кстати, успешность передачи фиксировалась в ходе прямого телефонного разговора с Биллом Дюваллем. С третьей попытки все символы слова были получены, и интернет, можно сказать, наконец «родился».

Эксперимент показал, что компьютеры не только могут быть физически соединены друг с другом, но и способны обмениваться данными и программами. Тем не менее, способ связи с помощью низкоскоростной телефонной сети уже тогда не мог считаться надежным и приемлемым.

ARPANET был построен на базе четырех миникопьютеров Honeywell DP-516, оснащенных 24 кбайт оперативной памяти и расположенных в университетах Санта-Барбары, Лос-Анджелеса, Стэнфорда и Юты. Данные передавались со скоростью до 56 Кбит/с.

К 1971 году количество компьютеров, подключенных к ARPANET, выросло почти в 6 раз: теперь основу сети составляли 23 хоста. Люди ступали на новые, еще неизведанные земли, и каждый новый шаг был инновацией. Так, в первые же дни работы ARPANET был придуман и внедрен протокол NCP, Network Control Protocol. Однако еще одно изобретение стало, как сейчас это принято называть, киллер-фичей интернета: электронная почта.

Приблизительно в 1972 году Рэй Томлинсон, сотрудник корпорации BBN, разработал первое ПО для отправки и чтения сообщений внутри ARPANET. Это первое, весьма грубое и неудобное по современным меркам, приложение заложило еще один компонент, без которого немыслим современный интернет: взаимодействие не только компьютеров, но и людей за ними.
К слову о коммуникации: первый протокол связи, NCP, работал только с определенными компьютерами, для которых был создан. Представьте себе, что сейчас, зайдя в интернет с iPhone, вы сможете общаться только с владельцами техники Apple. К тому же NCP мог обслуживать в лучшем случае несколько десятков компьютеров, в то время как присоединиться к ARPANET желали сотни пользователей. Это стало толчком к разработке более универсального и совершенного протокола, которым, как вы уже догадались, стал TCP/IP за авторством Винта Серфа и Боба Кана. Уже в 1974 году необходимость в NCP отпала сама собой, и большинство клиентов ARPANET стали использовать TCP/IP.

Тем не менее, энтузиастам из США со временем стало тесно в пределах родной страны. Требовалось расширить горизонты коммуникации и установить связь со странами по ту сторону Атлантики. Здесь стоит сделать небольшое отступление и поговорить о том, каким образом была впервые налажена прямая «кабельная» связь между Америкой и Европой.
Первый телеграфный кабель был проложен по дну Атлантического океана более чем за 100 лет до описываемых событий, в 1857 году. Тем не менее, по ряду причин (отсутствие опыта у участников экспедиции, несовершенная изоляция и пр.) уже через несколько недель этот кабель приказал долго жить. Спустя 10 лет по дну Атлантики было проложено уже несколько телеграфных кабелей с куда лучшей изоляцией, а к 1919 году их общее число увеличилось до 13.

Именно с помощью телефонного кабеля была впервые налажена связь США и Норвегии со скоростью 2,4 Кбит/с по ARPANET.

Таким образом, проект, разработанный военными для военных, стал чем-то большим и постепенно обрел независимость. Спутники, с помощью которых уже в конце 1970-х было налажено трансатлантическое соединение (SATNET), принадлежали уже не ВС США, а консорциуму стран, подключенных к сети.

Еще одним изобретением 70-х стал стандарт Ethernet, разработанный для более удобной организации локальной сети и передачи данных между компьютерами на высокой скорости. Этот стандарт в практически неизменном виде используется и сегодня.

Помимо Ethernet, еще одним важным нововведением 1970-х стал протокол UUCP (Unix to Unix Copy). Он позволял быстро обмениваться файлами между компьютерами под управлением Unix и со временем «превратился» в Usenet, сеть, с помощью которой миллионы людей до сих пор обмениваются новостями, отправляют электронную почту и файлы.

Переломный момент случился уже в следующем десятилетии и фактически создал тот интернет, которым мы сейчас пользуемся. Достаточно будет сказать, что в 1977 году к сети было подключено около 111 компьютеров, а в 1989 их число превысило 100 000.

Одним из важнейших «упорядочивающих» моментов стало создание системы доменных имен, DNS.

Ранее для хранения числовых адресов компьютеров, подключенных к ARPANET и сопоставления их с именами узлов, использовался текстовый файл HOSTS.TXT на компьютере Стэнфордского исследовательского института. Все адреса назначались сугубо вручную, и для того, чтобы запросить имя хоста и адрес, а также добавить компьютер в файл HOSTS, требовалось позвонить по телефону в сетевой информационный центр.

К началу 80-х стало понятно, что ввиду роста числа подключенных компьютеров такой способ поддержания централизованной таблицы хостов становится чрезвычайно медленным и громоздким. Требовалось автоматизировать систему именования.

Первая версия сервера имен BIND была написана студентами Беркли Дугласом Терри, Марком Пейнтером, Дэвидом Ригглом и Сонгниан Чжоу в 1984 году. В середине-конце 1980-х активно создавались и утверждались новые спецификации DNS, а к 90-м годам BIND был перенесен на платформу Windows NT. BIND до сих пор широко распространен в Unix-системах (и не только) и по-прежнему является одним из самых широко используемых ПО DNS в интернете. Внедрение DNS принесло массу других нововведений, например – принудительное использование протокола TCP/IP для подключения.
Уже в 1983 ARPANET состояла из 4000 хостов. Несмотря на изначальную сугубо военную направленность проекта, многие другие организации посчитали его крайне удобным. Среди них были университеты, предприятия, городские службы и многие, многие другие. Таким образом, ARPANET был разделен на две части: одноименную для гражданских нужд и военную MILNET. Несмотря на произошедшее разделение, Министерство обороны США продолжало поддерживать ARPANET, пусть даже военные организации более не использовали эту сеть для своих нужд.

CSNET начала работу в 1981 году и стала предшественницей высокоскоростной NSFNET, объединившей национальные научные фонды и ставшей основой современного интернета.
К слову о скорости. Прежних 50-56 кбит/с было крайне мало для эффективной коммуникации внутри постоянно растущей сети компьютеров. Благодаря компании MCI Corporation, сеть была существенно модернизирована путем внедрения новых линий T-1. Они позволяли передавать данные на скорости до 1,5 Мбит/с. В свою очередь, IBM разработали более совершенные маршрутизаторы, а компания Merit взяла под свой контроль все вопросы по управлению сетью. Уже к концу 1980-х в разработке находилась линия T-3, которая позволяла разогнать сети до 45 Мбит/с.
6 августа 1991 года свет увидела Всемирная паутина, знаменитая World Wide Web, и интернет официально «встал на крыло», а к 1993 году, появился первый графический браузер Mosaic.

11.Человеко-машинное взаимодействие: введение в вопросы проектирования (UI/UX).

UX/UI-дизайн — это работа над интерфейсом приложения или сайта, чтобы пользователю было интуитивно понятно и визуально приятно контактировать с ним.
UX (user experience) **—** «пользовательский опыт». UX — это то, как пользователь взаимодействует с интерфейсом. UX-дизайнер отвечает за удобство использования сайта или приложения.

Например, вы зашли на сайт маркетплейса. При хорошем UX-дизайне вы сразу разберётесь, как открыть каталог, выставить фильтры, сделать заказ и т. д. У вас останется положительное впечатление от взаимодействия с сайтом, потому что вам было просто на нём ориентироваться.

UI (user interface) — «пользовательский интерфейс». UI — это то, как выглядит интерфейс и его элементы. UI-дизайнер отвечает за наглядность сайта или приложения: вид меню, кнопок, читаемость шрифта и т. д.

Например, на том же сайте банка есть кнопки меню, надписи, баннеры, поисковик. При хорошем UI-дизайне вы легко можете прочитать текст на сайте, а надписи на кнопках не собьют вас с толку. Цвета достаточно контрастные, не сливаются с фоном, вёрстка аккуратная, слова не наезжают друг на друга.

Отличия UX от UI

Между UX- и UI-дизайном есть разница, но одно невозможно без другого. Специалисты UX и UI работают на благо пользователя — делают его взаимодействие с продуктом удобным с помощью внешнего дизайна и внутренней структуры.

Характеристики UX:

- фокус на удобстве и логичности взаимодействия,
- влияет на структуру приложения или сайта,
- цель — решать задачи пользователей.

Характеристики UI:

- фокус на элементах взаимодействия — кнопках, меню и т. д.,
- влияет на дизайн — шрифты, картинки, вёрстку и т. д.,
- цель — сделать удобный дизайн.

Веб-дизайн — это более широкое понятие, которое включает в себя UX/UI-дизайн. Веб-дизайнер занимается визуализацией макета сайта и комбинацией контента. Он может создавать шаблоны для email-рассылок, придумывать и рисовать интернет-баннеры.

На этом этапе дизайнер создаёт черновой вариант сайта или приложения — wireframe. В этом цифровом эскизе дизайнер пробует разные сочетания форм изображений, плашек и текста.

Прототипирование — это процесс совместной работы UX- и UI-дизайнеров. На этом этапе специалисты продумывают, что будет на страницах продукта и как расположить важные элементы на них.

Интерактивный прототип помогает:

- узнать, как пользователи взаимодействуют с сайтом или приложением,
- выявить проблемы в юзабилити,
- понять, как лучше удовлетворить потребности пользователей.

Иконки — это небольшие картинки, которые становятся частью интерфейса. Иногда дизайнеры могут сами их отрисовать, но чаще пользуются уже готовыми вариантами.

Задачи UX/UI-дизайнеров:

1. Анализ целевой аудитории

Специалист изучает, чего пользователи ждут от ресурса и как взаимодействуют с похожими сервисами. Он анализирует, кто будет работать с продуктом — формирует образ потенциального пользователя. После этого дизайнер создаёт примерный интерфейс и дизайн будущего сайта или приложения.

2. Разработка решений интерфейса

UX/UI-дизайнер создаёт схему взаимодействия пользователя с продуктом: от первого клика до завершающего действия. Специалист разрабатывает несколько возможных сценариев поведения пользователя, которые лягут в основу прототипа.

3. Создание прототипа, его тестирование и дизайн

Специалист разрабатывает прототипы — выстраивает логику взаимодействия с продуктом и создаёт его дизайн. Дизайнер тестирует приложение или сайт, чтобы заранее выявить ошибки и проблемы, с которыми столкнутся пользователи.

Требования к качественному UX/UI-дизайну

Пользователи хотят, чтобы сайт или приложение быстро решали задачи, выглядели приятно, были понятны и работали без сбоев. Поэтому к хорошему UX/UI-дизайну предъявляют определённые требования:

- ясность — нет двусмысленности, все части продукта решают задачи пользователя или направляют его,
- лаконичность — нет перегруженности лишними элементами,
- узнаваемость элементов — например, галочка для подтверждения будет зелёного цвета, а крестик для отмены — красного,
- отзывчивость — быстрая реакция интерфейса на действия пользователя,
- постоянство — элементы должны вести себя одинаково на каждой странице продукта,
- эстетика — интерфейс должен быть привлекательным и не отвлекать от решения задач,
- эффективность — пользователь сделает минимум действий прежде, чем попадёт в нужный раздел,
- забота — вежливые сообщения в случае ошибки или сбоя повышают лояльность пользователя.

12.Методология разработки программного обеспечения: основные понятия и принципы проектирования; структурная декомпозиция; стратегии тестирования и отладки; разработка сценариев тестирования (test cases);

Проектирование программного обеспечения — процесс создания проекта программного обеспечения (ПО), а также дисциплина, изучающая методы проектирования. Проектирование ПО является частным случаем проектирования продуктов и процессов.
Целью проектирования является определение внутренних свойств системы и детализации её внешних (видимых) свойств на основе выданных заказчиком требований к ПО (исходные условия задачи). Эти требования подвергаются анализу.
Проектирование ПО включает следующие основные виды деятельности:

- выбор метода и стратегии решения;
- выбор представления внутренних данных;
- разработка основного алгоритма;
- документирование ПО;
- тестирование и подбор тестов;
- выбор представления входных данных.

Первоначально программа рассматривается как чёрный ящик. Ход процесса проектирования и его результаты зависят не только от состава требований, но и выбранной модели процесса, опыта проектировщика.
Модель предметной области накладывает ограничения на бизнес-логику и структуры данных.
В зависимости от класса создаваемого ПО, процесс проектирования может обеспечиваться как «ручным» проектированием, так и различными средствами его автоматизации. В процессе проектирования ПО для выражения его характеристик используются различные нотации — блок-схемы, ER-диаграммы, UML-диаграммы, DFD-диаграммы, а также макеты.
Проектированию обычно подлежат:

- Архитектура ПО;
- Устройство компонентов ПО;
- Пользовательские интерфейсы.

В российской практике проектирование ведется поэтапно в соответствии со стадиями, регламентированными ГОСТ 2.103-68 :

1. Техническое задание(по ГОСТ 2.103-68 к стадиям разработки не относится),
2. Техническое предложение,
3. Эскизный проект,
4. Технический проект,
5. Рабочий проект.

На каждом из этапов формируется свой комплект документов, называемый проектом (проектной документацией).
В зарубежной практике регламентирующими документами, например, являются Software Architecture Document, Software Design Document.
Декомпозиция
Метод проектирования удовлетворяет критерию декомпозиции, если он помогает разложить задачу на несколько менее сложных подзадач, объединяемых простой структурой, и настолько независимых, что в дальнейшем можно отдельно продолжить работу над каждой из них.
Такой процесс часто будет циклическим, поскольку каждая подзадача может оказаться достаточно сложной и потребует дальнейшего разложения.
Следствием требования декомпозиции является разделение труда (division of labor): как только система будет разложена на подсистемы, работу над ними следует распределить между разными разработчиками или группами разработчиков. Это трудная задача, так как необходимо ограничить возможные взаимозависимости между подсистемами:

- Необходимо свести такие взаимозависимости к минимуму; в противном случае разработка каждой из подсистем будет ограничиваться темпами работы над другими подсистемами.
- Эти взаимозависимости должны быть известны: если не удастся составить перечень всех связей между подсистемами, то после завершения разработки проекта будет получен набор элементов программы, которые, возможно, будут работать каждая в отдельности, но не смогут быть собраны вместе в завершенную систему, удовлетворяющую общим требованиям к исходной задаче.

Наиболее очевидным примером обсуждаемого метода, удовлетворяющим критерию декомпозиции, является метод нисходящего (сверху вниз) проектирования (top-down design). В соответствии с этим методом разработчик должен начать с наиболее абстрактного описания функции, выполняемой системой. Затем последовательными шагами детализировать это представление, разбивая на каждом шаге каждую подсистему на небольшое число более простых подсистем до тех пор, пока не будут получены элементы с настолько низким уровнем абстракции, что становится возможной их непосредственная реализация. Этот процесс можно представить в виде дерева.
Типичным контрпримером (counter-example) является любой метод, предусматривающий включение в разрабатываемую систему модуля глобальной инициализации. Многие модули системы нуждаются в инициализации - открытии файлов или инициализации переменных.
Каждый модуль должен произвести эту инициализацию до начала выполнения непосредственно возложенных на него операций. Могло бы показаться, что все такие действия для всех модулей системы неплохо сосредоточить в одном модуле, который проинициализирует сразу все для всех. Подобный модуль будет обладать хорошей "согласованностью во времени" (temporal cohesion) в том смысле, что все его действия выполняются на одном этапе работы системы. Однако для получения такой "согласованности во времени", придется нарушать автономию других модулей. Придется модулю инициализации дать право доступа ко многим структурам данных, принадлежащим различным модулям системы и требующим специфических действий по их инициализации. Это означает, что автор модуля инициализации должен будет постоянно следить за структурами данных других модулей и взаимодействовать с их авторами. А это несовместимо с критерием декомпозиции.

Отладка **ПС** - это деятельность, направленная на обнаружение и исправление ошибок в ПС с использованием процессов выполнения его программ.
Тестирование **ПС** - это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Указанный набор данных называется тестовым или просто тестом. Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПС ошибки, поиска места ошибки в программах и документации ПС и редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами:
Отладка = Тестирование + Поиск ошибок + Редактирование.

Успех отладки ПС в значительной степени предопределяет рациональная организация тестирования. При отладке ПС отыскиваются и устраняются, в основном, те ошибки, наличие которых в ПС устанавливается при тестировании. Тестирование не может доказать правильность ПС, в лучшем случае оно может продемонстрировать наличие в нем ошибки. Другими словами, нельзя гарантировать, что тестированием ПС практически выполнимым набором тестов можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникает две задачи. Первая задача: подготовить такой набор тестов и применить к ним ПС, чтобы обнаружить в нем по возможности большее число ошибок. Однако чем дольше продолжается процесс тестирования (и отладки в целом), тем большей становится стоимость ПС. Отсюда вторая задача: определить момент окончания отладки ПС (или отдельной его компоненты). Признаком возможности окончания отладки является полнота охвата пропущенными через ПС тестами (т.е. тестами, к которым применено ПС) множества различных ситуаций, возникающих при выполнении программ ПС, и относительно редкое проявление ошибок в ПС на последнем отрезке процесса тестирования. Последнее определяется в соответствии с требуемой степенью надежности ПС, указанной в спецификации его качества.

Для оптимизации набора тестов, т.е. для подготовки такого набора тестов, который позволял бы при заданном их числе обнаруживать большее число ошибок в ПС, необходимо, во-первых, заранее планировать этот набор и, во-вторых, использовать рациональную стратегию планирования тестов. Проектирование тестов можно начинать сразу же после завершения этапа внешнего описания ПС. Возможны разные подходы к выработке стратегии проектирования тестов, которые можно условно графически разместить между следующими двумя крайними подходами. Левый крайний подход заключается в том, что тесты проектируются только на основании изучения спецификаций ПС (внешнего описания, описания архитектуры и спецификации модулей). Строение модулей при этом никак не учитывается, т.е. они рассматриваются как черные ящики. Фактически такой подход требует полного перебора всех наборов входных данных, так как в противном случае некоторые участки программ ПС могут не работать при пропуске любого теста, а это значит, что содержащиеся в них ошибки не будут проявляться. Однако тестирование ПС полным множеством наборов входных данных практически неосуществимо. Правый крайний подход заключается в том, что тесты проектируются на основании изучения текстов программ с целью протестировать все пути выполнения каждой программ ПС. Если принять во внимание наличие в программах циклов с переменным числом повторений, то различных путей выполнения программ ПС может оказаться также чрезвычайно много, так что их тестирование также будет практически неосуществимо.

Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю. Она включает проектирование значительной части тестов по спецификациям, но она требует также проектирования некоторых тестов и по текстам программ. При этом в первом случае эта стратегия базируется на принципах:

- на каждую используемую функцию или возможность - хотя бы один тест,
- на каждую область и на каждую границу изменения какой-либо входной величины - хотя бы один тест,
- на каждую особую (исключительную) ситуацию, указанную в спецификациях, - хотя бы один тест.

Во втором случае эта стратегия базируется на принципе: каждая команда каждой программы ПС должна проработать хотя бы на одном тесте.

Оптимальную стратегию проектирования тестов можно конкретизировать на основании следующего принципа: для каждого программного документа (включая тексты программ), входящего в состав ПС, должны проектироваться свои тесты с целью выявления в нем ошибок.

Различаются два основных вида отладки (включая тестирование): автономную и комплексную отладку ПС. *Автономная* отладка ПС означает последовательное раздельное тестирование различных частей программ, входящих в ПС, с поиском и исправлением в них фиксируемых при тестировании ошибок. Она фактически включает отладку каждого программного модуля и отладку сопряжения модулей. *Комплексная* отладка означает тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех документах (включая тексты программ ПС), относящихся к ПС в целом. К таким документам относятся определение требований к ПС, спецификация качества ПС, функциональная спецификация ПС, описание архитектуры ПС и тексты программ ПС.

Тестирование архитектуры ПС. Целью тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС. К моменту начала тестирования архитектуры ПС должна быть уже закончена автономная отладка каждой подсистемы. Ошибки реализации архитектуры могут быть связаны, прежде всего, с взаимодействием этих подсистем, в частности, с реализацией архитектурных функций (если они есть). Поэтому хотелось бы проверить все пути взаимодействия между подсистемами ПС. При этом желательно хотя бы протестировать все цепочки выполнения подсистем без повторного вхождения последних. Если заданная архитектура представляет ПС в качестве малой системы из выделенных подсистем, то число таких цепочек будет вполне обозримо.

Тестирование внешних функций. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Несмотря на то, что все эти программы автономно уже отлажены, указанные расхождения могут быть, например, из-за несоответствия внутренних спецификаций программ и их модулей (на основании которых производилось автономное тестирование) функциональной спецификации ПС. Как правило, тестирование внешних функций производится так же, как и тестирование модулей на первом шаге, т.е. как черного ящика.

Тестирование качества ПС. Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС. Это наиболее трудный и наименее изученный вид тестирования. Завершенность ПС проверяется уже при тестировании внешних функций. На данном этапе тестирование этого примитива качества может быть продолжено, если требуется получить какую-либо вероятностную оценку степени надежности ПС. Однако, методика такого тестирования еще требует своей разработки. Могут тестироваться такие примитивы качества, как точность, устойчивость, защищенность, временная эффективность, в какой-то мере - эффективность по памяти, эффективность по устройствам, расширяемость и, частично, независимость от устройств. Каждый из этих видов тестирования имеет свою специфику и заслуживает отдельного рассмотрения.

Тестирование документации по применению ПС. Целью тестирования является поиск несогласованности документации по применению и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС (тестированию определения требований к ПС и аттестации ПС), поэтому весьма важно разработчикам сначала самим воспользоваться ПС так, как это будет делать пользователь. Все тесты на этом этапе готовятся исключительно на основании только документации по применению ПС. Прежде всего, должны тестироваться возможности ПС как это делалось при тестировании внешних функций, но только на основании документации по применению. Должны быть протестированы все неясные места в документации, а также все примеры, использованные в документации. Далее тестируются наиболее трудные случаи применения ПС с целью обнаружить нарушение требований относительности легкости применения ПС.

Тестирование определения требований к ПС. Целью тестирования является выяснение, в какой мере ПС не соответствует предъявленному определению требований к нему. Особенность этого вида тестирования

заключается в том, что его осуществляет организация-покупатель или организация-пользователь ПС как один из путей преодоления барьера между разработчиком и пользователе. Обычно это тестирование производится с помощью контрольных задач - типовых задач, для которых известен результат решения. В тех случаях, когда разрабатываемое ПС должно прийти на смену другой версии ПС, которая решает хотя бы часть задач разрабатываемого ПС, тестирование производится путем решения общих задач с помощью как старого, так и нового ПС (с последующим сопоставлением полученных результатов). Иногда в качестве формы такого тестирования используют опытную эксплуатацию ПС - ограниченное применение нового ПС с анализом использования результатов в практической деятельности. По существу, этот вид тестирования во многом перекликается с испытанием ПС при его аттестации, но выполняется до аттестации, а иногда и вместо аттестации.

13.Среды разработки; инструменты тестирования и отладки.

IDE (Integrated Development Environment) – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора либо интерпретатора, средств автоматизации сборки и отладчика. Помимо этого, IDE может содержать инструменты для интеграции с системами управления версиями и другие полезные утилиты. Есть IDE, которые предназначены для работы только с одним языком программирования, однако большинство современных IDE позволяет работать сразу с несколькими.
IDE представляет собой более сложный инструмент, чем обычный текстовый редактор. Несмотря на то, что в текстовых редакторах есть масса полезных функций вроде подсветки синтаксиса, единственная их задача – обеспечивать работу с кодом. То есть для полноценной разработки вам понадобится еще хотя бы компилятор и отладчик.
IDE уже содержит в себе все эти и другие полезные компоненты. По сути, термин IDE обозначает то, что у вас под рукой будет все, что необходимо для разработки приложений и программ.

Инструменты тестирования и отладки – это программные средства, которые помогают разработчикам и тестировщикам обнаружить и исправить ошибки и проблемы в программном обеспечении. Эти инструменты обеспечивают возможность автоматизированного тестирования, анализа кода, поиска утечек памяти, диагностики сетевых проблем и многих других задач, связанных с обнаружением и исправлением ошибок в программном обеспечении. Некоторые популярные инструменты тестирования и отладки включают в себя:

Инструменты автоматизированного тестирования: например, Selenium, JUnit, TestNG, Cucumber, Postman и другие.

Среды разработки интегрированного программного обеспечения (IDE): например, Visual Studio, Eclipse, IntelliJ IDEA, Xcode и другие, которые предоставляют инструменты для отладки кода.

Профилировщики кода: например, Java VisualVM, Xcode Instruments, Chrome DevTools и другие, которые помогают анализировать производительность и определять проблемные места в коде.

Инструменты статического анализа кода: например, SonarQube, findbugs, PMD, Checkstyle и другие, которые помогают выявлять потенциальные ошибки и улучшать качество кода.

Инструменты для отладки сетевых проблем: например, Wireshark, Fiddler, Charles Proxy и другие, которые позволяют отслеживать и анализировать сетевой трафик.

Инструменты для анализа исключений: например, Sentry, Bugsnag, Crashlytics и другие, которые помогают в обнаружении и отладке исключительных ситуаций в реальном времени.

Инструменты для тестирования безопасности: например, Burp Suite, OWASP ZAP, Nessus, Acunetix и другие, которые помогают выявлять уязвимости и обнаруживать проблемы с безопасностью в приложениях.

Это только некоторые примеры инструментов тестирования и отладки, и выбор конкретного инструмента зависит от конкретных требований и потребностей разработчиков и тестировщиков.

Факторы для выбора среды:

- язык разработки;
- простота использования;
- на каких платформах работает;
- стоимость.

PhpStorm

Платформы: Windows/Linux/macOS

Умная среда от известной компании JetBrains предназначена для разработки на PHP, JavaScript, HTML и CSS и идеально подходит для работы с различными CMS: Drupal, Wordpress, Symfony, Joomla и многими другими. Среда разработки глубоко анализирует структуру кода, помогая избегать ошибок, а также поддерживает базы данных и SQL.

Преимущества

- Автодополнение кода и качественная отладка.
- Удобная навигация.
- Безопасный рефакторинг – применить изменения во всем проекте можно за пару кликов.
- Функция Live Edit позволяет мгновенно посмотреть все изменения в браузере.
- Интерфейс будет понятен даже новичкам.

Недостатки

- Нет бесплатной версии. Но можно скачать триал.

CLion

Платформы: Windows/Linux/macOS

Продукт JetBrains. CLion – идеальное кроссплатформенное решение для тех, кто работает на C и C++ (и не только). Умный редактор, удобный генератор кода, статический и динамический анализ, безопасный рефакторинг. Особенности данной среды разработки можно перечислять бесконечно.

Преимущества

- Поддержка удаленной разработки по SSH.
- Просмотр значений переменных прямо в редакторе.
- Умная помощь при написании кода.
- Возможность кастомизировать редактор.
- Быстрый и безопасный рефакторинг.
- Широкий функционал. IDE можно использовать даже для программирования микроконтроллеров.

Недостатки

- Нет бесплатной версии. Но, как и в случае с PhpStorm, можно скачать пробную версию.

Microsoft Visual Studio

Платформы: Windows/macOS (для Linux есть только редактор кода)

Поддерживаемые языки: Ajax, ASP.NET, DHTML, ASP.NET, JavaScript, Visual Basic, Visual C#, Visual C++, Visual F#, XAML и другие.

Microsoft Visual Studio – это премиум IDE, стоимость которой зависит от редакции и типа подписки. Она позволяет создавать самые разные проекты, начиная с мобильных и веб-приложений и заканчивая видеоиграми. Microsoft Visual Studio включает в себя множество инструментов для тестирования совместимости – вы сможете проверить свое приложение на более чем 300 устройствах и браузерах. Благодаря своей гибкости эта IDE отлично подойдет как для студентов, так и для профессионалов.

Особенности:

- Огромная коллекция всевозможных расширений, которая постоянно пополняется.
- Технология автодополнения IntelliSense.
- Возможность кастомизировать рабочую панель.
- Поддержка разделенного экрана (split screen).

Из недостатков можно выделить тяжеловесность этой IDE. Для выполнения даже небольших правок могут потребоваться значительные ресурсы, поэтому если нужно выполнить какую-то простую и быструю задачу, удобнее использовать более легкий редактор.

PyCharm

Платформы: Windows/Linux/macOS

Поддерживаемые языки: Python, Jython, Cython, IronPython, PyPy, AngularJS, Coffee Script, HTML/CSS, Django/Jinja2 templates, Gql, LESS/SASS/SCSS/HAML, Mako, Puppet, RegExp, Rest, SQL, XML, YAML и т.д.

Это интегрированная среда разработки на языке Python, которая была разработана международной компанией JetBrains (да, и снова эти ребята). Эта IDE распространяется под несколькими лицензиями, в том числе как Community Edition, где чуть урезан функционал. Сами разработчики характеризуют свой продукт как «самую интеллектуальную Python IDE с полным набором средств для эффективной разработки на языке Python».

Преимущества

- Поддержка Google App Engine; IronPython, Jython, Cython, PyPy wxPython, PyQt, PyGTK и др.
- Поддержка Flask-фреймворка и языков Mako и Jinja2.
- Редактор Javascript, Coffescript, HTML/CSS, SASS, LESS, HAML.
- Интеграция с системами контроля версий (VCS).
- UML диаграммы классов, диаграммы моделей Django и Google App Engine.

Недостатки

- Иногда встречаются баги, которые, как правило, не вызывают сильных неудобств.

IntelliJ IDEA

Платформы: Windows/Linux/macOS

Поддерживаемые языки: Java, AngularJS, Scala, Groovy, AspectJ, CoffeeScript, HTML, Kotlin, JavaScript, LESS, Node JS, PHP, Python, Ruby, Sass,TypeScript, SQL и другие.

Стоимость: от 499$ в год. Бесплатная версия работает только с Java и Android.

Еще одна IDE, разработанная компанией Jet Brains. Здесь тоже есть возможность использовать бесплатную версию Community Edition, а у платной версии есть тестовый 30-дневный период. Изначально IntelliJ IDEA создавалась как среда разработки для Java, но сейчас разработчики определяют эту IDE как «самую умную и удобную среду разработки для Java, включающую поддержку всех последних технологий и фреймворков». Используя плагины, эту IDE можно использовать для работы с другими языками.

Преимущества

- Инструменты для анализа качества кода, удобная навигация, расширенные рефакторинги и форматирование для Java, Groovy, Scala, HTML, CSS, JavaScript, CoffeeScript, ActionScript, LESS, XML и многих других языков.
- Интеграция с серверами приложений, включая Tomcat, TomEE, GlassFish, JBoss, WebLogic, WebSphere, Geronimo, Resin, Jetty и Virgo.
- Инструменты для работы с базами данных и SQL файлами.
- Интеграция с коммерческими системами управления версиями Perforce, Team Foundation Server, ClearCase, Visual SourceSafe.
- Инструменты для запуска тестов и анализа покрытия кода, включая поддержку всех популярных фреймворков для тестирования.

Недостатки

Придется потратить время для того, чтобы разобраться в этой IDE, поэтому начинающим программистам она может показаться сложноватой.

Eclipse

Платформы: Windows/Linux/macOS

Поддерживаемые языки: C, C++, Java, Perl, PHP, Python, Ruby и другие.

Это бесплатная опенсорсная среда разработки, которая хорошо подойдет как новичкам, так и опытным разработчикам. Помимо инструментов отладки и поддержки Git/CVS, Eclipse поставляется с Java и инструментом для создания плагинов. Изначально Eclipse использовалась только для Java, но сейчас,

благодаря плагинам и расширениям, ее функции значительно расширились. Именно из-за возможности расширить Eclipse своими модулями эта платформа и завоевала свою популярность среди разработчиков. Функционал Eclipse не такой большой, как у IntelliJ IDEA, зато эта среда разработки распространяется с открытым исходным кодом.

Преимущества

- Возможность программировать на множестве языков.
- Значительная гибкость среды за счет модульности.
- Возможность интеграции JUnit.
- Удаленная отладка (при использовании JVM).

Недостатки

- Новичкам может быть сложно разобраться в многообразии возможностей.

Xcode

Платформы: macOS

Поддерживаемые языки: AppleScript, C, C++, Java, Objective-C, Swift.

Функциональная среда для создания приложений под продукты Apple – iPhone, iPad, Mac, Apple TV и Apple Watch. В IDE могут работать как индивидуальные, так и корпоративные разработчики. Чтобы разместить созданное приложение в App Store, необходимо купить лицензию разработчика.

Преимущества

- Фирменный компилятор Apple.
- Создание прототипов без написания кода.
- Умный анализатор кода.

Недостатки

- Работает только на Mac.

14.Социальный контекст компьютинга: история компьютинга и компьютеров; эволюция идей и компьютеров; социальный эффект компьютеров и Интернета; профессионализм, кодекс этики и ответственное поведение; авторские права, интеллектуальная собственность и компьютерное пиратство.

Арифмометр — это механическая вычислительная машина, предназначенная для выполнения алгебраических операций. Первая схема такого устройства датируется 1500 годом за авторством Леонардо да Винчи. Вокруг его схемы в 60-х годах 20 века возникло много споров. Доктор Роберто Гуателли, работавший в IBM с 1951 года по проекту воссоздания машин Леонардо да Винчи, в 1968 году создал копию счетной машины по эскизам 16-го века.

Расцвет арифмометров пришелся на 17 век. Первой построенной моделью стал арифмометр Вильгельма Шиккарда в 1623 году. Его машина была 6-разрядной и состояла из 3 блоков — множительного устройства, блока сложения-вычитания и блока записи промежуточных результатов.

Также 17 век отметился ещё несколькими арифмометрами: «паскалина» за авторством Блеза Паскаля, арифмометр Лейбница и машина Сэмюэля Морленда. В промышленных масштабах арифмометры начали производиться в начале 19 века, а распространены были практически до конца 20-го.

Аналитическая и разностная машины Бэббиджа

Идея о создании разностной машины не принадлежит Чарльзу Бэббиджу. Она впервые была описана немецким инженером Иоганном Мюллером в книге с очень сложным названием. До конца не ясно, повлияли ли на Бэббиджа идеи Мюллера при создании разностной машины, поскольку Чарльз ознакомился с его работой в переводе, дата создания которого неизвестна.

Считается, что основные идеи для создания разностной машины Бэббидж взял из работ Гаспара де Прони и его идей о декомпозиции математических работ. Его идея заключалась в следующем: есть 3 уровня, на каждом из которых математики занимаются решением определенных проблем. На верхнем уровне находятся самые крутые математики и их задача — вывод математических выражений, пригодных для расчетов. У математиков на втором уровне стояла задача вычислять значения функций, которые вывели на верхнем уровне, для аргументов, с определенным периодом. Эти значения становились опорными для третьего уровня, задачей которого являлись рутинные расчеты. От них требовалось делать только грамотные вычисления. Их так и называли — «вычислители». Эта идея навела Бэббиджа на мысль о создании машины, которая могла бы заменить «вычислителей». Машина Бэббиджа основывалась на методе аппроксимации функций многочленами и вычисления конечных разностей. Собственно, поэтому машина и называется разностной.

В 1822 году Бэббидж построил модель разностной машины и заручился государственной поддержкой в размере 1500 фунтов стерлингов. Он планировал, что закончит машину в течение 3 лет, но по итогу работа была не завершена и через 9 лет. За это время он получил ещё 15500 фунтов стерлингов в виде субсидий от государства. Но всё же часть машины функционировала и производила довольно точные (>18 знаков после запятой) расчеты.

Во время работы над разностной машиной у Чарльза Бэббиджа возникла идея о создании аналитической машины — универсальной вычислительной машины. Её называют прообразом современного цифрового компьютера. Она состояла из арифметического устройства, памяти и устройства ввода-вывода, реализованного с помощью перфокарт различного типа.

Табулятор

История электромеханических машин начинается в 1888 году, когда американский инженер Герман Холлерит, основатель компании CTR (будущая IBM), изобрел электромеханическую счетную машину — табулятор, который мог считывать и сортировать данные, закодированные на перфокартах. В аппарате использовались электромагнитные реле, известные еще с 1831 года и до Холлерита не применявшиеся в счетной технике. Управление механическими счетчиками и сортировкой осуществлялось электрическими импульсами, возникающими при замыкании электрической цепи при наличии отверстия в перфокарте. Импульсы использовались и для ввода чисел, и для управления работой машины. Поэтому табулятор Холлерита можно считать первой счетной электромеханической машиной с программным управлением. Машину полностью построили в 1890 году и использовали при переписи населения США в том же году. Впоследствии табуляторы использовались вплоть до 1960-х — 1970-х годов в бухгалтерии, учете, обработке данных переписей и подобных работах. И даже если в учреждении имелась полноценная ЭВМ, табуляторы все равно использовали, чтобы не нагружать ЭВМ мелкими задачами.

Поколения:

Первое:

Первой работающей машиной с архитектурой фон Неймана стала Манчестерская малая экспериментальная машина, созданная в Манчестерском университете в 1948 году; в 1949 году за ним последовал компьютер Манчестерский Марк I, который уже был полной системой, с трубками Уильямса и магнитным барабаном в качестве памяти, а также с индексными регистрами. Другим претендентом на звание «первый цифровой компьютер с хранимой программой» стал EDSAC, разработанный и сконструированный в Кембриджском университете. Заработавший менее чем через год после «Baby», он уже мог использоваться для решения реальных задач. На самом деле, EDSAC был создан на основе архитектуры компьютера EDVAC, наследника ENIAC. В отличие от ENIAC, использовавшего параллельную обработку, EDVAC располагал единственным обрабатывающим блоком. Такое решение было проще и надёжнее, поэтому такой вариант становился первым реализованным после каждой очередной волны миниатюризации. Многие считают, что Манчестерский Марк I / EDSAC / EDVAC стали «Евами», от которых ведут свою архитектуру почти все современные компьютеры. Первым универсальным программируемым компьютером в континентальной Европе был Z4 Конрада Цузе, завершённый в сентябре 1950 года. В ноябре того же года командой учёных под руководством Сергея Алексеевича Лебедева из Киевского института электротехники, УССР, была создана так называемая «малая электронная счётная машина» (МЭСМ). Она содержала около 6000 **электровакуумных ламп** и потребляла 15 кВт. Машина могла выполнять около 3000 операций в секунду. Другой машиной того времени была австралийская CSIRAC, которая выполнила свою первую тестовую программу в 1949 году.
В октябре 1947 года директора компании Lyons & Company британской компании, владевшей сетью магазинов и ресторанов, решили принять активное участие в развитии коммерческой разработки

компьютеров. Компьютер LEO I начал работать в 1951 году и впервые в мире стал регулярно использоваться для рутинной офисной работы.
Разработанная в 1950-1951 в СССР ЭВМ М-1 стала первой в мире ЭВМ, в которой все логические схемы были выполнены на полупроводниках.
Машина Манчестерского университета стала прототипом для Ferranti Mark I. Первая такая машина была доставлена в университет в феврале 1951 года, и, по крайней мере, девять других были проданы между 1951 и 1957 годами.
В июне 1951 года UNIVAC 1 был установлен в Бюро переписи населения США. Машина была разработана в компании Remington Rand, которая, в конечном итоге, продала 46 таких машин по цене более чем в $1 млн за каждую. UNIVAC был первым массово производившимся компьютером; все его предшественники изготовлялись в единичном экземпляре. Компьютер состоял из 5200 электровакуумных ламп и потреблял 125 кВт энергии. Использовались ртутные линии задержки, хранящие 1000 слов памяти, каждое по 11 десятичных цифр плюс знак (72-битные слова). В отличие от машин IBM, оснащавшихся устройством ввода с перфокарт, UNIVAC использовал ввод с металлизированной магнитной ленты стиля 1930-х, благодаря чему обеспечивалась совместимость с некоторыми существовавшими коммерческими системами хранения данных. Другими компьютерами того времени использовался высокоскоростной ввод с перфоленты и ввод-вывод с использованием более современных магнитных лент.
Первой советской серийной ЭВМ стала «Стрела», производившаяся с 1953 года на Московском заводе счётно-аналитических машин. «Стрела» относится к классу больших универсальных ЭВМ с трёхадресной системой команд. ЭВМ имела быстродействие 2000-3000 операций в секунду. В качестве внешней памяти использовались два накопителя на магнитной ленте ёмкостью 200 000 слов, объём оперативной памяти — 2048 ячеек по 43 разряда. Компьютер состоял из 6200 ламп, 60 000 полупроводниковых диодов и потреблял 150 кВт энергии.
В 1954 году IBM выпускает машину IBM 650, ставшую довольно популярной — всего было выпущено более 2000 машин. Она весит около 900 кг, и ещё 1350 кг весит блок питания; оба модуля имеют размер примерно 1,5 × 0,9 × 1,8 метров. Цена машины составляет $0,5 млн (около $4 млн в пересчёте на 2011 год) либо может быть взята в лизинг за $3 500 в месяц ($30 000 на 2011 год). Память на магнитном барабане хранит 2000 10-знаковых слов, позже память была увеличена до 4000 слов. По мере исполнения программы инструкции считывались прямо с барабана. В каждой инструкции был задан адрес следующей исполняемой инструкции. Использовался компилятор Symbolic Optimal Assembly Program (SOAP), который размещал инструкции по оптимальным адресам, так, чтобы следующая инструкция читалась сразу и не требовалось ждать, пока барабан повернётся до нужного ряда.
В 1955 году Морис Уилкс изобретает *микропрограммирование*, принцип, который позднее широко используется в микропроцессорах самых различных компьютеров. Микропрограммирование позволяет определять или расширять базовый набор команд с помощью встроенных программ (которые носят названия *микропрограмма* или *firmware*).
В 1956 году IBM впервые продаёт устройство для хранения информации на магнитных дисках — RAMAC (Random Access Method of Accounting and Control). Оно использует 50 металлических дисков диаметром 24 дюйма, по 100 дорожек с каждой стороны. Устройство хранило до 5 МБ данных и стоило по 10 000 $ за МБ. (В 2006 году подобные устройства хранения данных — жёсткие диски — стоят около 0,001 $ за Мб.)

1950-е — начало 1960-х: второе поколение

Следующим крупным шагом в истории компьютерной техники стало изобретение **транзистора** в 1947 году. Они стали заменой хрупким и энергоёмким лампам. О компьютерах на транзисторах обычно говорят как о «втором поколении», которое преобладало в 1950-х и начале 1960-х. Благодаря транзисторам и печатным платам было достигнуто значительное уменьшение размеров и объёмов потребляемой энергии, а также повышение надёжности. Например, IBM 1620 на транзисторах, ставшая заменой IBM 650 на лампах, была размером с письменный стол. Однако компьютеры второго поколения по-прежнему были довольно дороги и поэтому использовались только университетами, правительствами, крупными корпорациями.
Компьютеры второго поколения обычно состояли из большого количества печатных плат, каждая из которых содержала от одного до четырёх логических вентилей или триггеров. В частности, IBM Standard Modular System определяла стандарт на такие платы и разъёмы подключения для них. Первые полупроводниковые компьютеры строились на германиевых транзисторах, потом им на смену пришли более дешёвые кремниевые. Логика строилась на биполярных транзисторах и прошла эволюцию от РТЛ, ТТЛ до ЭСЛ-логики. Им на смену пришли полевые транзисторы, на основе которых строились простейшие микросхемы уже для компьютеров третьего поколения.
Концепция ЭВМ 1950-х годов предполагала наличие дорогостоящего вычислительного центра с собственным персоналом. Содержание таких ЭВМ могли себе позволить лишь крупные корпорации и государственные структуры (а также ряд крупных университетов). В общей сложности в 1958 году существовало только 1700 ЭВМ всех разновидностей в пользовании 1200 организаций. Однако в течение нескольких последующих лет были выпущены тысячи, а затем десятки тысяч компьютеров, и они впервые стали широко доступны для среднего бизнеса и научных работников.

Без рывка в сфере вычислительной техники, сделанного в 1940-е гг. и чётко сформулированного технического задания к разработчикам такого рода, вычислительная техника не только не развилась бы до современных компьютеров, но по всей вероятности осталась бы на уровне довоенного периода (что показали опыты Цузе, создавшего гениальные и революционные для своего времени образцы вычислительной техники, совершенно невостребованной ни государственными структурами, ни общественными институтами). Фактически, появлением первых компьютеров, а затем суперкомпьютеров и стремительному рывку в развитии вычислительной техники, началу серийного производства компьютеров, формированием компьютерной индустрии со всеми сопутствующими отраслями (индустрии программных продуктов, компьютерных игр и т. д.) человечество обязано опытам по автоматизации баллистических вычислений Второй мировой войны в Великобритании и в меньшей степени в США.

В 1959 году на основе транзисторов IBM выпустила IBM 7090 и машину среднего класса IBM 1401. Последняя использовала перфокарточный ввод и стала самым популярным компьютером общего назначения того времени: в период 1960—1964 гг. было выпущено более 100 тыс. экземпляров этой машины. В ней использовалась память на 4000 символов (позже увеличенная до 16 000 символов). Многие аспекты этого проекта были основаны на желании заменить перфокарточные машины, которые широко использовались начиная с 1920-х до самого начала 1970-х гг.

В 1960 году IBM выпустила транзисторную IBM 1620, изначально только перфоленточную, но вскоре обновлённую до перфокарт. Модель стала популярна в качестве научного компьютера, было выпущено около 2000 экземпляров. В машине использовалась память на магнитных сердечниках объёмом до 60 000 десятичных цифр.

В том же 1960 году DEC выпустила свою первую модель — PDP-1, предназначенную для использования техническим персоналом в лабораториях и для исследований. Этот относительно мощный по тем временам компьютер (100 тыс. операций в секунду) имел довольно компактные размеры (занимал пространство размером с бытовой холодильник)

В 1961 году Burroughs Corporation выпустила B5000, первый двухпроцессорный компьютер с виртуальной памятью на основе подкачки сегментов. Другими уникальными особенностями были стековая архитектура, адресация на основе дескрипторов, и отсутствие программирования напрямую на языке ассемблера.

В 1962 году совместно Манчестерским университетом Виктории и компаниями Ferranti и Plessey был создан компьютер Atlas с виртуальной памятью на основе подкачки страниц и конвейерным выполнением инструкций.

Компьютер второго поколения IBM 1401, выпускавшийся в начале 1960-х, занял около трети мирового рынка компьютеров, было продано более 10 000 таких машин.

Применение полупроводников позволило улучшить не только центральный процессор, но и периферийные устройства. Второе поколение устройств хранения данных позволяло сохранять уже десятки миллионов символов и цифр. Появилось разделение на жёстко закреплённые (*fixed*) устройства хранения, связанные с процессором высокоскоростным каналом передачи данных, и сменные (*removable*) устройства. Замена кассеты дисков в сменном устройстве требовала лишь несколько секунд. Хотя ёмкость сменных носителей была обычно ниже, но их заменяемость давала возможность сохранения практически неограниченного объёма данных. Магнитная лента обычно применялось для архивирования данных, поскольку предоставляла больший объём при меньшей стоимости.

Во многих машинах второго поколения функции общения с периферийными устройствами делегировались специализированным сопроцессорам. Например, в то время как периферийный процессор выполняет чтение или пробивку перфокарт, основной процессор выполняет вычисления или ветвления по программе. Одна шина данных переносит данные между памятью и процессором в ходе цикла выборки и исполнения инструкций, и обычно другие шины данных обслуживают периферийные устройства. На PDP-1 цикл обращения к памяти занимал 5 микросекунд; большинство инструкций требовали 10 микросекунд: 5 на выборку инструкции и ещё 5 на выборку операнда.

«Сетунь» была первой ЭВМ на основе троичной логики, разработана в 1958 году в Советском Союзе. Первыми советскими серийными полупроводниковыми ЭВМ стали «Весна» и «Снег», выпускаемые с 1964 по 1972 год. Пиковая производительность ЭВМ «Снег» составила 300 000 операций в секунду. Машины изготавливались на базе транзисторов с тактовой частотой 5 МГц. Всего было выпущено 39 ЭВМ.

Лучшей отечественной ЭВМ 2-го поколения считается БЭСМ-6, созданная в 1966 году.

1960-е: третье поколение

Бурный рост использования компьютеров начался с «третьего поколения» вычислительных машин. Начало этому положило изобретение **интегральной схемы**, которое стало возможным благодаря цепочке открытий, сделанных американскими инженерами в 1958—1959 годах. Они решили три фундаментальные проблемы, препятствовавшие созданию интегральной схемы; за сделанные открытия один из них получил Нобелевскую премию.

В 1964 году был представлен мейнфрейм IBM/360. Эти ЭВМ и её наследники на долгие годы стали фактическим промышленным стандартом для мощных ЭВМ общего назначения. В СССР аналогом IBM/360 были машины серии ЕС ЭВМ.

Параллельно с компьютерами третьего поколения продолжали выпускаться компьютеры второго поколения. Так, компьютеры «UNIVAC 494» выпускались до середины 1970-х годов.

1970-е: четвёртое поколение

В 1969 году сотрудник компании Intel Тэд Хофф предлагает создать центральный процессор на одном кристалле. То есть, вместо множества интегральных микросхем создать одну главную **интегральную микросхему**, которая должна будет выполнять все арифметические, логические операции и операции управления, записанные в машинном коде. Такое устройство получило название микропроцессор.

В 1971 году компания Intel по заказу фирмы Busicom выпускает первый микропроцессор «Intel 4004» для использования в калькуляторе (модель Busicom 141-PF). Появление микропроцессоров позволило создать микрокомпьютеры — небольшие недорогие компьютеры, которые могли себе позволить купить маленькие компании или отдельные люди. В 1980-х годах микрокомпьютеры стали повсеместным явлением.

Первый массовый домашний компьютер был разработан Стивом Возняком — одним из основателей компании Apple Computer. Позже Стив Возняк разработал первый массовый персональный компьютер.

Компьютеры на основе микрокомпьютерной архитектуры с возможностями, добавленными от их больших собратьев, сейчас доминируют в большинстве сегментов рынка.

Ассоциация вычислительной техники (ACM) - крупнейшее в мире образовательное и научное компьютерное общество. Он имеет свой собственный Кодекс этики и другой набор этических принципов, которые также были одобрены IEEE в качестве стандарта для преподавания и практики разработки программного обеспечения. Этими кодексами являются Кодекс этики и профессионального поведения и Кодекс этики и профессиональной практики программной инженерии соответственно, и некоторые из их руководящих принципов представлены ниже:

Из Кодекса этики и профессионального поведения (ACM):

Вносить вклад в общество и благосостояние людей.Программисты должны работать над созданием компьютерных систем, которые могут уменьшить негативные последствия для общества, такие как угрозы безопасности и здоровью, и которые могут облегчить повседневную деятельность и работу. Это “обязательство развиваться в соответствии с высокими стандартами” (Savage).

Избегайте причинения вреда другим. Компьютерные системы оказывают косвенное влияние на третьи стороны. Они могут привести к потере информации и ресурсов, что может нанести серьезный ущерб пользователям, широкой общественности или работодателям. Поэтому разработчики программного обеспечения должны минимизировать риск причинения вреда другим людям из-за ошибок в кодировании или проблем безопасности, следуя стандартам проектирования и тестирования систем (Кодекс этики и профессионального поведения).

Будьте честны и заслуживайте доверия. Этот принцип побуждает программистов быть честными и осознавать свои ограничения в знаниях и образовании при написании компьютерных систем. Кроме того, если программист знает, что с компьютерной системой что-то не так, он или она должны немедленно сообщить об этом, чтобы избежать нежелательных последствий.

Отдавайте должное интеллектуальной собственности. Для каждого разработчика программного обеспечения обязательным является никогда не использовать и не приписывать себе чужую работу, даже если она не защищена законом об авторском праве, патентом и т. Д. Они должны признавать и полностью доверять работам других людей, и они должны использовать свои собственные идеи для разработки программного обеспечения.

Уважайте частную жизнь других людей. Компьютерные системы неправильно используются некоторыми людьми для нарушения конфиденциальности других. Разработчики программного обеспечения должны писать программы, которые могут защитить личную информацию пользователей и которые могут предотвратить несанкционированный доступ к ней других нежелательных людей (Кодекс этики и профессионального поведения).

Соблюдайте конфиденциальность. Если этого не требует закон или какие-либо другие этические нормы, программист должен хранить в секрете любую дополнительную информацию, связанную с его или ее работодателем, которая возникает в результате работы в проекте.

Из Кодекса этики и профессиональной практики программной инженерии (IEEE, ACM):

Одобряйте программное обеспечение, только если у них есть обоснованное убеждение, что оно безопасно и соответствует спецификациям.Программисты не могут считать, что система готова к использованию только потому, что она выполняет необходимые задачи. Они должны убедиться, что эти системы также безопасны и соответствуют всем спецификациям, требуемым пользователем. Если программы небезопасны, пользователи остаются незащищенными от хакеров, которые могут украсть важную информацию или деньги. Поэтому для обеспечения безопасности системы перед ее утверждением следует выполнить несколько тестов.

Принять на себя полную ответственность за свою работу. Если в программе присутствуют ошибки, разработчик программного обеспечения должен принять на себя полную ответственность за свою работу и должен работать над ее пересмотром, исправлением, модификацией и тестированием.

Не использовать сознательно программное обеспечение, полученное или сохраненное незаконно или неэтично.Если компьютерная система будет использоваться в качестве основы для создания другой, то разрешение на это должно быть запрошено программистом. Этот принцип запрещает использование любого

другого программного обеспечения для любых целей, если способ его получения неясен или известен как незаконный или неэтичный.

Выявлять, определять и решать этические, экономические, культурные, правовые и экологические проблемы, связанные с рабочими проектами. Если программист замечает и определяет, что работа над проектом приведет к каким-либо проблемам, программист должен сообщить об этом своему работодателю, прежде чем продолжить.

Убедитесь, что спецификации программного обеспечения, над которым они работают, удовлетворяют требованиям пользователей и имеют соответствующие разрешения.Разработчики программного обеспечения должны обратиться к своим работодателям за соответствующим одобрением создаваемой ими системы, прежде чем продолжить работу над следующей частью. Если она не соответствует требованиям, то следует внести изменения в исходный код системы.

Обеспечить надлежащее тестирование, отладку и проверку программного обеспечения. Программисты должны выполнять соответствующие тесты для частей программного обеспечения, с которыми они работают, и должны проверять наличие ошибок и дыр в системе безопасности, чтобы убедиться, что программы хорошо реализованы.

Не участвовать в мошеннических финансовых действиях, таких как взяточничество, двойное выставление счетов или другие ненадлежащие финансовые действия. Программисты могут участвовать в незаконных действиях с целью получения денег. Они участвуют в них из-за угроз, экономических проблем или просто потому, что хотят получить легкие деньги, воспользовавшись своими знаниями о том, как работают компьютерные системы. Это руководство запрещает программисту участвовать в таких незаконных действиях.

Улучшают свою способность создавать безопасное, надежное и полезное качественное программное обеспечение. Поскольку технологии с каждым годом развиваются все быстрее, как и виртуальная преступность, потребность в хорошо структурированных и разработанных программах возрастает. Компьютерные системы стареют и ограничиваются новыми и новыми устройствами. Программисты должны “углублять свои знания о разработках в области анализа, спецификации, проектирования, разработки, обслуживания и тестирования программного обеспечения и связанных с ними документов” (Кодекс этики и профессиональной практики программной инженерии), чтобы создавать лучшие части программного обеспечения.

Пиратство

Нелегальное копирование и распространение программных продуктов (в том числе компьютерных игр) на дисках и в компьютерных сетях. Включает в себя снятие разнообразных программных защит. Для этого существует специальный класс программного обеспечения — так называемые «кряки» (от англ. *to crack* — взламывать), специальные патчи, готовые серийные номера или их генераторы для программного продукта, которые снимают с него ограничения, связанные со встроенной защитой от нелегального использования.

Кроме того, существуют инструменты программистов, которые могут использоваться для облегчения самого процесса взлома — отладчики, дизассемблеры, редакторы PE-заголовка, редакторы ресурсов, распаковщики, и т. п.

Использование взломанного программного обеспечения также несёт в себе набор рисков в плане компьютерной безопасности. Поскольку зачастую такие программы лишаются возможности обновляться, уязвимости не будут своевременно устраняться, что может повлечь взлом или заражение компьютера в будущем. Сами же взломанные программы или средства для их взлома также могут распространяться с внедрением вредоносного кода, то есть вместе с установкой нелицензионного программного обеспечения на компьютер можно загрузить программы-шипоны, вирусы, троянское программное обеспечения или же во взломанное программное обеспечение могут быть встроены бекдоры.

15.Объектно-ориентированное программирование: объектно-ориентированное проектирование, инкапсуляция и скрытие информации; разделение интерфейса и реализации; классы, наследники и наследование; полиморфизм; иерархии классов.

Объектно–ориентированное проектирование. Разработка объектно–ориентированной модели системы ПО (системной архитектуры) с учётом требований. В этой модели определение всех объектов подчинено решению конкретной задачи.

Инкапсуляция в объектно-ориентированном программировании - принцип, которым определяется способ организации данных и методов внутри классов. Инкапсуляция обеспечивает сокрытие внутренней реализации объекта от внешнего мира и предоставляет доступ к ним только через определенные методы. Это позволяет контролировать и управлять доступом к данным, обеспечивая их целостность и безопасность.

Под сокрытием данных подразумевается защита членов класса от незаконного или несанкционированного доступа. Основное различие между сокрытием данных и инкапсуляцией заключается в том, что сокрытие данных больше фокусируется на безопасности данных, а инкапсуляция - больше на скрытии сложности системы.

Принцип разделения интерфейса — один из пяти основных принципов объектно-ориентированного программирования и проектирования, сформулированных Робертом Мартином. Клиенты не должны зависеть от методов, которые они не используют. То есть если какой-то метод интерфейса не используется клиентом, то изменения этого метода не должны приводить к необходимости внесения изменений в клиентский код.

Принцип разделения интерфейса снижает сложность поддержки и развития приложения. Чем проще и минималистичнее используемый интерфейс, тем менее ресурсоёмкой является его реализация в новых классах, тем меньше причин его модифицировать, но и в случае модификации она будет значительно проще.

Наследование – это концепция, которая предполагает, что один класс может наследовать функции и данные другого класса. Класс, от которого производится наследование называется родительским или базовым классом, класс который наследует – наследником. Отношение между классом наследником и базовым классом можно определить словом “является”.

Полиморфизм

Говоря про полиморфизм в общем, можно сказать, что это возможность обработки данных разных типов одной и той же функцией. Различают параметрический полиморфизм и *ad–hoc* полиморфизм. Параметрический полиморфизм предполагает, что один и тот же код в функции может работать с данными разных типов. *Ad–hoc* полиморфизм предполагает создание различных реализаций функций в зависимости от типа аргумента(ов), при этом их сигнатура (без учета типов входных аргументов) остается одной и той же.

Класс объявляется с помощью ключевого слова *class* перед ним могут стоять несколько модификаторов, после располагается имя класса. Если предполагается, что класс является наследником другого класса или реализует один или несколько интерфейсов, то они отделяются двоеточием от имени класса и перечисляются через запятую.

Внутри себя, класс может содержать методы, поля и свойства. Методы похожи на функции из языков группы структурного программирования. Фактически они определяют то, как можно работать с данным классом или объектами класса. Поля – это переменные, связанные с данным классом, а свойства – это конструкции специального вида, которые упрощают работу с полями (в первом приближении такого понимания будет достаточно).

```
class DemoClass
{
        // Поле класса
        int field = 0;

        // Свойство класса
        public int Property {get;set;}

        // Метод класса
        public void Method()
        {
        Console.WriteLine("Method");
        }
}
```

16.Основные вычислительные алгоритмы: алгоритмы поиска и сортировки.

Поиск — обработка некоторого множества данных с целью выявления подмножества данных, соответствующего критериям поиска.

Все алгоритмы поиска делятся на

- поиск в неупорядоченном множестве данных;
- поиск в упорядоченном множестве данных.

Упорядоченность – наличие отсортированного ключевого поля.

Сортировка — упорядочение (перестановка) элементов в подмножестве данных по какому-либо критерию. Чаще всего в качестве критерия используется некоторое числовое поле, называемое ключевым. Упорядочение элементов по ключевому полю предполагает, что ключевое поле каждого следующего элемента не больше предыдущего (***сортировка по убыванию***). Если ключевое поле каждого последующего элемента не меньше предыдущего, то говорят о ***сортировке по возрастанию***.

Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве при обработке данных.

Все алгоритмы сортировки делятся на
- алгоритмы внутренней сортировки (сортировка массивов);
- алгоритмы внешней сортировки (сортировка файлов).

Последовательный Поиск
Последовательный поиск предполагает последовательный просмотр всех записей множества, организованного как массив.

Метод транспозиции

Метод транспозиции: каждый запрос к записи сопровождается сменой мест этой и предшествующий записи; в итоге наиболее часто используемые записи постепенно перемещаются в начало таблицы; и при последующем обращении к ним, эти записи находятся почти сразу.

Метод перемещения в начало

В этом методе каждый запрос к записи сопровождается её перемещением в начало таблицы. В итоге в начале таблицы оказывается запись, используемая в последний раз.

Индексно-последовательный поиск
Для индексно-последовательного поиска в дополнение к отсортированной таблице заводится вспомогательная таблица, называемая ***индексной***.

Каждый элемент индексной таблицы состоит из ключа и указателя на запись в основной таблице, соответствующей этому ключу. Элементы в индексной таблице, как элементы в основной таблице, должны быть отсортированы по этому ключу.

Если индекс имеет размер, составляющий 1/8 от размера основной таблицы, то каждая восьмая запись основной таблицы будет представлена в индексной таблице.
Если размер основной таблицы — **n**, то размер индексной таблицы — **ind_size = n/8**.

Достоинство алгоритма индексно-последовательного поиска заключается в том, что сокращается время поиска, так как последовательный поиск первоначально ведется в индексной таблице, имеющей меньший размер, чем основная таблица. Когда найден правильный индекс, второй последовательный поиск выполняется по небольшой части записей основной таблицы.

Бинарный поиск

Бинарный поиск производится в упорядоченном массиве.

При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в массиве. Если они равны, то поиск успешен. В противном случае поиск осуществляется аналогично в левой или правой частях массива.

Алгоритм может быть определен в рекурсивной и нерекурсивной формах.

Бинарный поиск также называют поиском методом деления отрезка пополам или дихотомии.
На каждом шаге осуществляется поиск середины отрезка по формуле
mid = (left + right)/2
Если искомый элемент равен элементу с индексом mid, поиск завершается.
В случае если искомый элемент меньше элемента с индексом mid, на место mid перемещается правая граница рассматриваемого отрезка, в противном случае — левая граница.

Чтобы уменьшить количество шагов поиска можно сразу смещать границы поиска на элемент, следующий за серединой отрезка:
left = mid + 1
right = mid — 1

Сортировка пузырьком / Bubble sort

Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован. Заметим, что после первой итерации самый большой элемент будет находиться в конце массива, на правильном месте. После двух итераций на правильном месте будут стоять два наибольших элемента, и так далее. Очевидно, не более чем после n итераций массив будет отсортирован. Таким образом, асимптотика в худшем и среднем случае – $O(n^2)$, в лучшем случае – $O(n)$.

Шейкерная сортировка / Shaker sort

(также известна как сортировка перемешиванием и коктейльная сортировка). Заметим, что сортировка пузырьком работает медленно на тестах, в которых маленькие элементы стоят в конце (их еще называют «черепахами»). Такой элемент на каждом шаге алгоритма будет сдвигаться всего на одну позицию влево. Поэтому будем идти не только слева направо, но и справа налево. Будем поддерживать два указателя begin и end, обозначающих, какой отрезок массива еще не отсортирован. На очередной итерации при достижении end вычитаем из него единицу и движемся справа налево, аналогично, при достижении begin прибавляем единицу и двигаемся слева направо. Асимптотика у алгоритма такая же, как и у сортировки пузырьком, однако реальное время работы лучше.

Сортировка вставками / Insertion sort

Создадим массив, в котором после завершения алгоритма будет лежать ответ. Будем поочередно вставлять элементы из исходного массива так, чтобы элементы в массиве-ответе всегда были отсортированы. Асимптотика в среднем и худшем случае – $O(n^2)$, в лучшем – $O(n)$. Реализовывать алгоритм удобнее по-другому (создавать новый массив и реально что-то вставлять в него относительно сложно): просто сделаем так, чтобы отсортирован был некоторый префикс исходного массива, вместо вставки будем менять текущий элемент с предыдущим, пока они стоят в неправильном порядке.

Сортировка Шелла / Shellsort

Используем ту же идею, что и сортировка с расческой, и применим к сортировке вставками. Зафиксируем некоторое расстояние. Тогда элементы массива разобьются на классы – в один класс попадают элементы, расстояние между которыми кратно зафиксированному расстоянию. Отсортируем сортировкой вставками каждый класс. В отличие от сортировки расческой, неизвестен оптимальный набор расстояний. Существует довольно много последовательностей с разными оценками. Последовательность Шелла – первый элемент равен длине массива, каждый следующий вдвое меньше предыдущего. Асимптотика в худшем случае – $O(n^2)$. Последовательность Хиббарда – $2^n — 1$, асимптотика в худшем случае – $O(n^{1.5})$, последовательность Седжвика (формула нетривиальна, можете ее посмотреть по ссылке ниже) — $O(n^{4/3})$, Пратта (все произведения степеней двойки и тройки) — $O(n\log^2 n)$. Отмечу, что все эти последовательности нужно рассчитать только до размера массива и запускать от большего от меньшему (иначе получится просто сортировка вставками). Также я провел дополнительное исследование и протестировал разные последовательности вида $s_i = a * s_{i-1} + k * s_{i-1}$ (отчасти это было навеяно эмпирической последовательностью Циура – одной из лучших последовательностей расстояний для небольшого количества элементов). Наилучшими оказались последовательности с коэффициентами a = 3, k = 1/3; a = 4, k = 1/4 и a = 4, k = -1/5.

Быстрая сортировка / Quicksort

Выберем некоторый опорный элемент. После этого перекинем все элементы, меньшие его, налево, а большие – направо. Рекурсивно вызовимся от каждой из частей. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотика: $O(n\log n)$ в среднем и лучшем случае, $O(n^2)$. Наихудшая оценка достигается при неудачном выборе опорного элемента. Моя реализация этого алгоритма совершенно стандартна, идем одновременно слева и справа, находим пару элементов, таких, что левый элемент больше опорного, а правый меньше, и меняем их местами. Помимо чистой быстрой сортировки, участвовала в сравнении и сортировка, переходящая при малом количестве элементов на сортировку вставками. Константа подобрана тестированием, а сортировка вставками — наилучшая сортировка, подходящая для этой задачи (хотя не стоит из-за этого думать, что она самая быстрая из квадратичных).

Сортировка слиянием / Merge sort

Сортировка, основанная на парадигме «разделяй и властвуй». Разделим массив пополам, рекурсивно отсортируем части, после чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем

минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Слияние работает за O(n), уровней всего logn, поэтому асимптотика O(nlogn). Эффективно заранее создать временный массив и передать его в качестве аргумента функции. Эта сортировка рекурсивна, как и быстрая, а потому возможен переход на квадратичную при небольшом числе элементов.

Сложность алгоритма - это количественная характеристика, которая говорит о том, сколько времени, либо какой объём памяти потребуется для выполнения алгоритма.
Big O показывает то, как сложность алгоритма растёт с увеличением входных данных. При этом она всегда показывает худший вариант развития событий - верхнюю границу.

17.Основы программирования, основанного на событиях.

Событи́йно-ориенти́рованное программи́рование — парадигма программирования, в которой выполнение программы определяется событиями — действиями пользователя (клавиатура, мышь, сенсорный экран), сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета).
СОП можно также определить как способ построения компьютерной программы, при котором в коде (как правило, в головной функции программы) явным образом выделяется *главный цикл приложения*, тело которого состоит из двух частей: *выборки события* и *обработки события*.
Как правило, в реальных задачах оказывается недопустимым длительное выполнение обработчика события, поскольку при этом программа не может реагировать на другие события. В связи с этим при написании событийно-ориентированных программ часто применяют автоматное программирование.
Событийно-ориентированное программирование, как правило, применяется в трёх случаях:

1. при построении пользовательских интерфейсов (в том числе графических);
2. при создании серверных приложений в случае, если по тем или иным причинам нежелательно порождение обслуживающих процессов;
3. при программировании игр, в которых осуществляется управление множеством объектов.

Событийно-ориентированное программирование применяется в серверных приложениях для решения проблемы масштабирования на 10000 одновременных соединений и более.
В серверах, построенных по модели «один поток на соединение», проблемы с масштабируемостью возникают по следующим причинам:

- слишком велики накладные расходы на структуры данных операционной системы, необходимые для описания одной задачи (сегмент состояния задачи, стек);
- слишком велики накладные расходы на переключение контекстов.

Серверное приложение при событийно-ориентированном программировании реализуется на системном вызове, получающем события одновременно от многих дескрипторов (мультиплексирование). При обработке событий используются исключительно неблокирующие операции ввода-вывода, чтобы ни один дескриптор не препятствовал обработке событий от других дескрипторов.
Применение в настольных приложениях
В современных языках программирования события и обработчики событий являются центральным звеном реализации графического интерфейса пользователя. Рассмотрим, к примеру, взаимодействие программы с событиями от мыши. Нажатие правой клавиши мыши вызывает системное прерывание, запускающее определённую процедуру внутри операционной системы. В этой процедуре происходит поиск окна, находящегося под курсором мыши. Если окно найдено, то данное событие посылается в очередь обработки сообщений этого окна. Далее, в зависимости от типа окна, могут генерироваться дополнительные события. Например, если окно является кнопкой (в Windows все графические элементы являются окнами), то дополнительно генерируется событие нажатия на кнопку. Отличие последнего события в том, что оно более абстрактно, а именно, не содержит координат курсора, а говорит просто о том, что было произведено нажатие на данную кнопку.
Здесь обработчик события представляет собой процедуру, в которую передается параметр sender, как правило содержащий указатель на источник события. Это позволяет использовать одну и ту же процедуру для обработки событий от нескольких кнопок, различая их по этому параметру.

18.Введение в компьютерную графику: использование простых графических API.

Графические API-интерфейсы - это не часть оборудования, а часть программного обеспечения, но они необходимы для рендеринга графики, представленной на нашем экране, и без них связи между приложениями и GPU / ГРАФИЧЕСКИЙ ПРОЦЕССОР было бы невозможно.

Графические API-интерфейсы - это то, что позволяет приложениям взаимодействовать с графическим процессором, чтобы отмечать, как он должен отрисовывать следующий кадр или его часть. Концепция основана на абстракции, которая представляет собой создание на языке программирования того, что такое графический процессор. Чтобы понять концепцию абстракции, мы предположим, что вместо графического процессора у нас подключен автомат с газировкой.
API машины обновления будет библиотекой со следующими функциями: подбрасывать монету, возвращать сдачу, выбирать обновление и доставлять Refreshment таким образом, чтобы программа могла взаимодействовать, эта часть API вызывается Front-End, и то, что мы делаем с ним, - это список инструкций, в графическом процессоре этот список называется DisplayList или списком экранов.

Какие графические API существуют сегодня?

Наиболее часто используемые сегодня графические API:

- **Вулкан:** Заменяет OpenGL, он используется во всех типах операционных систем, который, как говорят, не зависит от платформы, но является основным API Google.
- **Металл:** Графический API Apple, используемый в macOS и специально оптимизированный для архитектуры графического процессора.
- **DirectX:** API Microsoft для Windows и Xbox был разделен на несколько версий в зависимости от платформы, но недавно они объединили его в единую версию.
- **GHM/GHMX/GHM++:** Графический API для консолей Sony PlayStation 4 и PlayStation 5, GNMX - это высокоуровневый API, GNM - это низкоуровневый API версии на PS4 и GNM ++ на PS5.
- **HBH:** Графический API Nintendo Switch, который используется исключительно в этой серии консолей.
- **OpenGL:** API, с которого все началось, первоначально известный как IrisGL и предназначенный для рабочих станций Silicon Graphics, превратился в API для ПК, консолей и более поздних смартфонов. Я дохожу до версии 4, которая является Vulkan - это ребрендинг OpenGL 5.

Графические процессоры - это очень сложные процессоры, которые давно перестали быть просто игрушками для рендеринга видеоигр, сегодня они используются в таких областях, как искусственный интеллект или высокопроизводительные вычисления, что привело к эволюции графических API-интерфейсов и вышло за рамки графики.
В настоящее время приложения отправляют не один список, а несколько списков, один из которых является графикой, а остальные вычисления, где графический процессор используется для решения конкретных проблем, которые не имеют ничего общего с визуализацией графики, причем последние работают полностью асинхронно и, следовательно, не зависит от списка экранов.
Например, может случиться так, что приложение графического дизайна использует мощность графического процессора для создания специального эффекта на фотографии только потому, что графический процессор лучше оборудован для решения этой проблемы, чем графический процессор. Благодаря спискам вычислений вы можете сделать это, используя бесплатные ресурсы графического процессора для решения этих небольших проблем.

Графические API высокого и низкого уровня: чем они отличаются?

Когда мы говорим о низкоуровневом API, мы имеем в виду API, который работает близко к графическому процессору, который находится внизу стека, в то время как драйвер находится наверху, поэтому высокоуровневый API - это тот, который требует, чтобы драйвер генерировал дисплей. список. Поскольку определенные задачи драйвера не выполняются в высокоуровневом API, теоретически достигается то, что время выполнения списка экранов ЦП короче, это означает завершение кадра за меньшее количество миллисекунд или предоставление большего времени для улучшения графики тот самый кадр.
На самом деле неверно, что низкоуровневым API-интерфейсам не хватает драйвера, поскольку его можно читать и прослушивать в некоторых местах, но это намного проще и усложняет работу при выполнении некоторых важных задач в приложении, это позволит разработчикам максимально оптимизируйте синхронизацию каждого кадра, контролируя процесс создания списка экранов.
Однако во многих случаях для разработчиков может быть намного удобнее использовать высокоуровневый API из-за того, что дополнительное время разработки не окупается с финансовой точки зрения или просто потому, что выгода, которую можно получить за счет адаптации игры к API низкий уровень незаметен.

19.Обзор языков программирования: история языков программирования; краткий обзор парадигм программирования.

1843: Машинный алгоритм Ады Лавлейс

Ада Лавлейс изобретает первый в истории машинный алгоритм для разностной машины Чарльза Бэббиджа, который закладывает основу для всех языков программирования.

1944-45: Планкалкюль
Где-то между 1944-45 годами Конрад Цузе разработал первый «настоящий» язык программирования под названием Plankalkül (Расчет плана). Язык Zeus (помимо прочего) позволял создавать процедуры, в которых хранятся фрагменты кода, которые можно было вызывать снова и снова для выполнения рутинных операций.

1949: Язык Ассемблера
Ассемблер использовался в автоматическом калькуляторе с электронным запоминанием задержки (EDSAC). Ассемблер был разновидностью низкоуровневого языка программирования, который упростил язык машинного кода. Другими словами, конкретные инструкции, необходимые для работы с компьютером.

1952: Автокодирование
Автокод был общим термином, используемым для семейства языков программирования. Autocode, впервые разработанный Аликом Гленни для компьютера Mark 1 в Университете Манчестера, был первым в истории скомпилированным языком, который был реализован, что означает, что он может быть переведен непосредственно в машинный код с помощью программы, называемой компилятором. Автокод использовался на первых вычислительных машинах Ferranti Pegasus и Sirius в дополнение к Mark 1.

1957: Fortran
FORmula TRANslation или FORTRAN был создан Джоном Бэкусом и считается старейшим языком программирования, используемым сегодня. Язык программирования был создан для научных, математических и статистических вычислений высокого уровня. FORTRAN до сих пор используется в некоторых из самых передовых суперкомпьютеров в мире.

1958: ALGOL (Алгоритмический язык)
Алгоритмический язык или АЛГОЛ был создан совместным комитетом американских и европейских компьютерных ученых. Алгол послужил отправной точкой для разработки некоторых из наиболее важных языков программирования, включая Pascal, C, C ++ и Java.

1964: BASIC (универсальный символьный код инструкций для начинающих)
Универсальный код символических инструкций для начинающих или BASIC был разработан группой студентов Дартмутского колледжа. Этот язык был написан для студентов, которые плохо разбирались в математике или компьютерах. Этот язык был разработан основателями Microsoft Биллом Гейтсом и Полом Алленом и стал первым товарным продуктом компании.

1970: ПАСКАЛЬ
Названный в честь французского математика Блеза Паскаля, Никлаус Вирт разработал язык программирования в его честь. Он был разработан как средство обучения компьютерному программированию, что означало, что его легко освоить. Apple предпочитала его на заре своей деятельности из-за простоты использования и мощности.

1972: C (Си)
Разработан Деннисом Ричи из Bell Telephone Laboratories для использования с операционной системой Unix. Он был назван C, потому что был основан на более раннем языке под названием «B». Многие из ведущих в настоящее время языков являются

производными от C, включая; C #, Java, JavaScript, Perl, PHP и Python. Он также использовался / до сих пор используется такими крупными компаниями, как Google, Facebook и Apple.

1972: SQL (в то время SEQUEL)
SQL был впервые разработан исследователями IBM Рэймондом Бойсом и Дональдом Чемберленом. SEQUEL (как его тогда называли) используется для просмотра и изменения информации, хранящейся в базах данных. В настоящее время язык является аббревиатурой - SQL, что означает язык структурированных запросов. Существует множество компаний, использующих SQL, и некоторые из них включают Microsoft и Accenture.

1980/81: Ада
Изначально Ada была разработана командой во главе с Джин Ичбиа из CUU Honeywell Bull по контракту с Министерством обороны США. Названный в честь математика середины 19-го века Ады Лавлейс, Ada представляет собой структурированный, статически типизированный, императивный, объектно-ориентированный язык программирования высокого уровня с широким спектром возможностей. Ада была расширена из других популярных в то время языков программирования, таких как Паскаль. Ada используется в системах управления воздушным движением в таких странах, как Австралия, Бельгия и Германия, а также во многих других транспортных и космических проектах.

1983: C ++
Бьярн Страуструп модифицировал язык C в Bell Labs, C ++ - это расширение C с такими улучшениями, как классы, виртуальные функции и шаблоны. Он был включен в 10 лучших языков программирования с 1986 года и получил статус Зала славы в 2003 году. C ++ используется в MS Office, Adobe Photoshop, игровых движках и другом высокопроизводительном программном обеспечении.

1983: Objective-C
Objective-C, разработанный Брэдом Коксом и Томом Лавом, является основным языком программирования, используемым для написания программного обеспечения для операционных систем Apple macOS и iOS.

1987: Perl
Perl был создан Ларри Уоллом и представляет собой универсальный язык программирования высокого уровня. Первоначально он был разработан как язык сценариев, предназначенный для редактирования текста, но в настоящее время он широко используется для многих целей, таких как CGI, приложения баз данных , системное администрирование, сетевое программирование и графическое программирование.

1990: Haskell
Haskell - это язык программирования общего назначения, названный в честь американского логика и математика Хаскелла Брукса Карри. Это чисто функциональный язык программирования, то есть в первую очередь математический. Он используется во многих отраслях, особенно в тех, которые имеют дело со сложными вычислениями, записями и обработкой чисел. Как и многие другие языки программирования той эпохи, не так уж часто можно увидеть, что Haskell используется для хорошо известных приложений. С учетом сказанного, язык программирования был использован для написания ряда игр, одна из которых - Nikki and the Robots.

1991: Python
Названный в честь британской комедийной труппы «Монти Пайтон», Python был разработан Гвидо Ван Россумом. Это универсальный язык программирования высокого уровня, созданный для поддержки различных стилей программирования и приятный в использовании (ряд руководств, примеров и инструкций часто содержат ссылки на Monty Python). Python по сей день является одним из самых популярных языков программирования в мире, который используют такие компании, как Google, Yahoo и Spotify.

1991: Visual Basic
Visual Basic, разработанный Microsoft, позволяет программистам использовать стиль перетаскивания для выбора и изменения предварительно выбранных фрагментов кода через графический интерфейс пользователя (GUI). В наши дни этот язык не используется слишком часто, однако Microsoft частично использовала Visual Basic для ряда своих приложений, таких как Word, Excel и Access.

1993: Ruby
Ruby, созданный Юкихиро Мацумото, представляет собой интерпретируемый язык программирования высокого уровня. Язык обучения, на который повлияли Perl, Ada, Lisp и Smalltalk - среди прочих. В основном Ruby используется для разработки веб-приложений и Ruby on Rails. Twitter, Hulu и Groupon - известные примеры компаний, использующих Ruby.

1995: Java
Java - это универсальный язык высокого уровня, созданный Джеймсом Гослингом для проекта интерактивного телевидения. Он обладает кросс-платформенной функциональностью и неизменно входит в число самых популярных языков программирования в мире. Java можно найти везде, от компьютеров до смартфонов и парковочных счетчиков.

1995: PHP
Ранее известный как «Персональная домашняя страница», что теперь означает «Препроцессор гипертекста», PHP был разработан Расмусом Лердорфом. Его основное применение включает создание и поддержку динамических веб-страниц, а также разработку на стороне сервера . Некоторые из крупнейших компаний по всему миру используют PHP, включая Facebook, Wikipedia, Digg, WordPress и Joomla.

1995: JavaScript
JavaScript был создан Бренданом Эйхом, этот язык в основном используется для динамической веб-разработки, документов PDF, веб-браузеров и виджетов рабочего стола. Почти каждый крупный веб-сайт использует JavaScript. Gmail, Adobe Photoshop и Mozilla Firefox включают несколько хорошо известных примеров.

2000: C #
Разработанный в Microsoft с надеждой на объединение вычислительных возможностей C ++ с простотой Visual Basic, C # основан на C ++ и имеет много общего с Java. Этот язык используется почти во всех продуктах Microsoft и используется в основном при разработке настольных приложений.

2003: Scala
Scala, разработанная Мартином Одерски, объединяет математическое функциональное программирование и организованное объектно-ориентированное программирование.

Совместимость Scala с Java делает его полезным при разработке под Android. Linkedin, Twitter, Foursquare и Netflix - это всего лишь несколько примеров многих компаний, которые используют Scala в своих технических стеках.

2009: Golang (Go)
Go был разработан Google для решения проблем, возникающих из-за больших программных систем. Благодаря своей простой и современной структуре Go завоевал популярность среди некоторых крупнейших технологических компаний по всему миру, таких как Google, Uber, Twitch и Dropbox.

2014: Swift
Разработанный Apple в качестве замены C, C ++ и Objective-C, Swift был разработан с целью быть проще, чем вышеупомянутые языки, и оставлять меньше места для ошибок. Универсальность Swift означает, что его можно использовать для настольных, мобильных и облачных приложений. Ведущее языковое приложение Duolingo запустило новое приложение, написанное на Swift.

Парадигмы программирования
Существует три основных парадигмы: структурное, объектно-ориентированное и функциональное. Интересно, что сначала было открыто функциональное, потом объектно-ориентированное, и только потом структурное программирование, но применяться повсеместно на практике они стали в обратном порядке.
Структурное программирование было открыто Дейкстрой в 1968 году. Он понял, что goto – это зло, и программы должны строиться из трёх базовых структур: последовательности, ветвления и цикла.
Объектно-ориентированное программирование было открыто в 1966 году.
Функциональное программирование открыто в 1936 году, когда Чёрч придумал лямбда-исчисление. Первый функциональный язык LISP был создан в 1958 году Джоном МакКарти.
Каждая из этих парадигм убирает возможности у программиста, а не добавляет. Они говорят нам скорее, что нам не нужно делать, чем то, что нам нужно делать.
Все эти парадигмы очень связаны с архитектурой. Полиморфизм в ООП нужен, чтобы наладить связь через границы модулей. Функциональное программирование диктует нам, где хранить данные и как к ним получить доступ. Структурное программирование помогает в реализации алгоритмов внутри модулей.
Структурное программирование
Дейкстра понял, что программирование – это сложно. Большие программы имеют слишком большую сложность, которую человеческий мозг не способен контролировать.
Чтобы решить эту проблему, Дейкстра решил сделать написание программ подобно математическим доказательствам, которые также организованы в иерархии. Он понял, что если в программах использовать только if, do, while, то тогда такие программы можно легко рекурсивно разделять на более мелкие единицы, которые в свою очередь уже легко доказуемы.
С тех пор оператора goto не стало практически ни в одном языке программирования.
Таким образом, структурное программирование позволяет делать функциональную декомпозицию.
Однако на практике мало кто реально применял аналогию с теоремами для доказательства корректности программ, потому что это слишком накладно. В реальном программировании стал популярным более «лёгкий» вариант: тесты. Тесты не могут доказать корректности программ, но могут доказать их некорректность. Однако на практике, если использовать достаточно большое количество тестов, этого может быть вполне достаточно.
Объектно-ориентированное программирование
ООП – это парадигма, которая характеризуется наличием инкапсуляции, наследования и полиморфизма.
Инкапсуляция обеспечивает сокрытие внутренней реализации объекта от внешнего мира и предоставляет доступ к ним только через определенные методы. Это позволяет контролировать и управлять доступом к данным, обеспечивая их целостность и безопасность.
Однако в современных языках инкапсуляция, наоборот, слабее, чем была даже в C. В Java, например, вообще нельзя разделить объявление класса и его определение. Поэтому сказать, что современные объектно-ориентированные языки предоставляют инкапсуляцию можно с очень большой натяжкой.
Наследование позволяет делать производные структуры на основе базовых, тем самым давая возможность осуществлять повторное использование этих структур. Наследование было реально сделать в языках до ООП, но в объектно-ориентированных языках оно стало значительно удобнее.
Наконец, полиморфизм позволяет программировать на основе интерфейсов, у которых могут быть множество реализаций. Полиморфизм осуществляется в ОО-языках путём использования виртуальных методов, что является очень удобным и безопасным.

Полиморфизм – это ключевое свойство ООП для построения грамотной архитектуры. Он позволяет сделать модуль независимым от конкретной реализации (реализаций) интерфейса. Этот принцип называется инверсией зависимостей, на котором основаны все плагинные системы.

Инверсия зависимостей так называется, что она позволяет изменить направление зависимостей. Сначала мы начинаем писать в простом стиле, когда высокоуровневые функции зависят от низкоуровневых. Однако, когда программа начинает становиться слишком сложной, мы инвертируем эти зависимости в противоположную сторону: высокоуровневые функции теперь зависят не от конкретных реализаций, а от интерфейсов, а реализации теперь лежат в своих модулях.

Любая зависимость всегда может быть инвертирована. В этом и есть мощь ООП.

Таким образом, между различными компонентами становится меньше точек соприкосновения, и их легче разрабатывать. Мы даже можем не перекомпилировать базовые модули, потому что мы меняем только свой компонент.

Функциональное программирование

В основе функционального программирования лежит запрет на изменение переменных. Если переменная однажды проинициализирована, её значение так и остаётся неизменным.

Какой профит это имеет для архитектуры? Неизменяемые данные исключают гонки, дедлоки и прочие проблемы конкурентных программ. Однако это может потребовать больших ресурсов процессора и памяти.

Применяя функциональный подход, мы разделяем компоненты на изменяемые и неизменяемые. Причём как можно больше функциональности нужно положить именно в неизменяемые компоненты и как можно меньше в изменяемые. В изменяемых же компонентах приходится работать с изменяемыми данными, которые можно защитить с помощью транзакционной памяти.

Интересным подходом для уменьшения изменяемых данных является Event Sourcing. В нём мы храним не сами данные, а историю событий, которые привели к изменениям этих данных. Так как в лог событий можно только дописывать, это означает, что все старые события уже нельзя изменить. Чтобы получить текущее состояние данных, нужно просто воспроизвести весь лог. Для оптимизации можно использовать снапшоты, которые делаются, допустим, раз в день.

Заключение

Таким образом, каждая из трёх парадигм ограничивает нас в чём-то:

- Структурное отнимает у нас возможность вставить goto где угодно.
- ООП не позволяет нам получить доступ до скрытых членов классов и навязывает нам инверсию зависимостей.
- ФП запрещает изменять переменные.