Ответы на вопросы к зачёту по программированию в ЗРЛ.

**1. Процесс сборки программы**

1. Препроцессинг

Вход — исходные файлы (\*.с)

Выход — обработанные файлы (\*.ii)

удаление из кода комментариев,

преобразование кода в соответствии с макросами,

выполнение директив, начинающихся с символа «#»

2. Компиляция

Вход — обработанные файлы (\*.ii)

Выход — код на ассемблере (\*.s) / объектные файлы (\*.o)

трансляция программных модулей, написанных на языке программирования высокого уровня, в эквивалентные программные модули на низкоуровневом языке (язык ассемблера/машинный код/объектные файлы)

(2.1. Ассемблирование)

Вход — код на ассемблере (\*.s)

Выход — объектные файлы (\*.o)

трансляция ассемблерного кода в машинный код

3. Линковка (компоновка)

Вход — объектные файлы (\*.o)

Выход — исполняемый файл

создание исполняемого файла из объектных файлов с помощью таблиц символов

Исполнитель алгоритма сборки программы — компилятор. Алгоритм — последовательность действий, переводящих целевой объект из начального состояния в требуемое. Создается на понятном исполнителю языке.

**2. Утилиты (команды) Linux: что такое утилита, как выполнить, примеры.**

Утилита — компьютерная программа узкого (специального) назначения в составе ПО ОС. Чтобы выполнить утилиту, её надо вызвать.

Вызов утилит:

<cmd> (простой вызов утилиты) [pwd]

<cmd> <p1> <p2> (вызов утилиты с параметрами) [mv parser.cpp.../parser.cpp]

<cmd> -<f> (сигнальный однобуквенный флаг) [ls -l]

<cmd> --<flag> (сигнальный полнобуквенный флаг) [ls --human-readable]

<cmd> -<f> <p> (однобуквенный флаг с параметром) [gcc parser.c -o parser]

<cmd> -<f>=<p> (однобуквенный флаг с параметром) [gcc parser.c -o parser -std=c99]

Возможно комбинирование ключей:

<cmd> -<f1><f2> (комбинация сигнальных ключей) [ls -lh]  
Список наиболее часто встречающихся утилит:

man — вывод справки;

cd - сменить директорию

ls - вывести список файлов и каталогов

mkdir - создать директорию

rm - удалить файл

mv - переименовать или переместить file1 в file2

cp - скопировать file1 в file2

cat - отображение содержимого file на стандартный вывод

whereis - отображение расположения файла file

**3. Представление целых чисел в памяти компьютера. Битовые операции.**

В памяти компьютера целые числа хранятся в двоичной системе счисления.

Целые числа со знаком — множество положительных и отрицательных чисел в диапазоне от [-2^(k-1) ; 2^(k-1) – 1].

Прямой код – это представление числа в двоичной системе счисления, при котором старший разряд отводится под знак числа. Если число положительное, то в левый разряд записывается 0; если число отрицательное, то записывается 1.

Была попытка сделать так, что все нули - это нижняя граница, все единицы - верхняя граница переменной. Например, для char -128 = 00000000,

127 = 11111111.

Самым удобным при вычислениях оказался дополнительный код. Дополнительный код следует аксиоме а + (-а) = 0.

Дополнительный код:

* Дополнительный код от положительного числа = самому числу в двоичной записи
* Для отрицательных чисел:
  + Модуль числа переводится в двоичный код
  + Обратный код (инвертируем все разряды двоичного числа)
  + + 12

При сложении дает в сумме значение 2^k.

Побитовые И, ИЛИ, НЕ, и ИСКЛЮЧАЮЩЕЕ ИЛИ:  
И (a & b) ИЛИ (a | b) НЕ (~a) ИСКЛ. ИЛИ (XOR) (a ^ b)

Побитовый сдвиг вправо >> (влево <<) — принимает два операнда: число, над которым необходимо совершить операцию сдвига, и число бит, на которое необходимо совершить сдвиг.

Приемы с битовыми операциями:

1. Получить бит числа

char num = 5; // 00000101

size\_t i = 3; // i находится в пределах размера битовой сетки

bool bit = (num & (1 << i)) ? 1 : 0; // i-ый бит числа

2. Выставить бит числа

char num = 5; // 00000101

size\_t i = 3;

size\_t j = 2;

num |= (1 << i); // Запись 1 на i-ую позицию

num &= ~(1 << j); // Запись 0 на 2 позицию

3. Битовые маски

Битовая переменная, используется:

-для "фильтрации" битов;

-для оптимального по памяти хранения булевых переменных.

Примеры:

{

const unsigned short olderBitMask = 0x8000; // битовая маска, используемая для операций со старшим битом, эквивалентно 32768 в 10-ой системе

unsigned short num = 65535; // 0xFFFF

std::cout << (num & olderBitMask) ? 1 : 0;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#define BIT\_0 0b00000001 // или 0x01 или 1

int main()

{

for (int i = 0; i < 10; ++i)

{

unsigned int value = rand() % 1000;

if (value & BIT\_0)

printf("%u нечетный\n", value);

else

printf("%u четный\n", value);

}

return 0;

}

4. Целочисленное деление и умножение на 2 в степени

char num = 5; // 00000101 - 5

num <<= 2; // 00010100 - 2^4 + 2^2 = 20 = 5 \* 4

num >>= 2; // 00000001

**4. Представление вещественных чисел в памяти компьютера.**

Формат представления вещественных чисел в памяти компьютера называется форматом с плавающей точкой. Вещественное число представляется в виде произведения: ***(-1) в степени s*** *- знака числа (0/1)* на ***мантиссу m*** *- часть числа, которая несет значение* на ***основание системы счисления e* в *степени* *p****,* которую называют порядком:

A = (-1)s \* m \* ep

Мантисса в нормализованном представлении ∈ [1, 10). В памяти компьютера мантисса представляется как целое число, содержащее только значащие цифры (0 и запятая не хранятся).

В ячейке хранится следующая информация: знак числа, порядок, мантисса.



Рассмотрим формулу на примере double и float: s – знак числа(1 бит всегда), m - мантисса(float - 23 бита, double - 52 бита), р – порядок(float - 8 бит, double - 11 бит).

Порядок байтов для хранения чисел:

* big-endian:  
  | 7 … 0 | 7 … 0 |  
  15 8 7 0
* little-endian (Intel x86):  
  | 7 … 0 | 7 … 0 |  
   7 0 15 8

**5. Адресное пространство. Адрес, указатель.**

Адресное пространство — диапазон всех возможных адресов, обозначающих определенное место в памяти. Делятся на три вида: логическое, линейное, физическое.

Из пространств с числовым представлением адреса наиболее известно пространство оперативной памяти. С точки зрения процессора массив запоминающих устройств состоит из элементарных ячеек длиной 1 байт.

Адрес переменной — путь, по которому находится значение самой переменной (сдвиг ячейки памяти от начала).

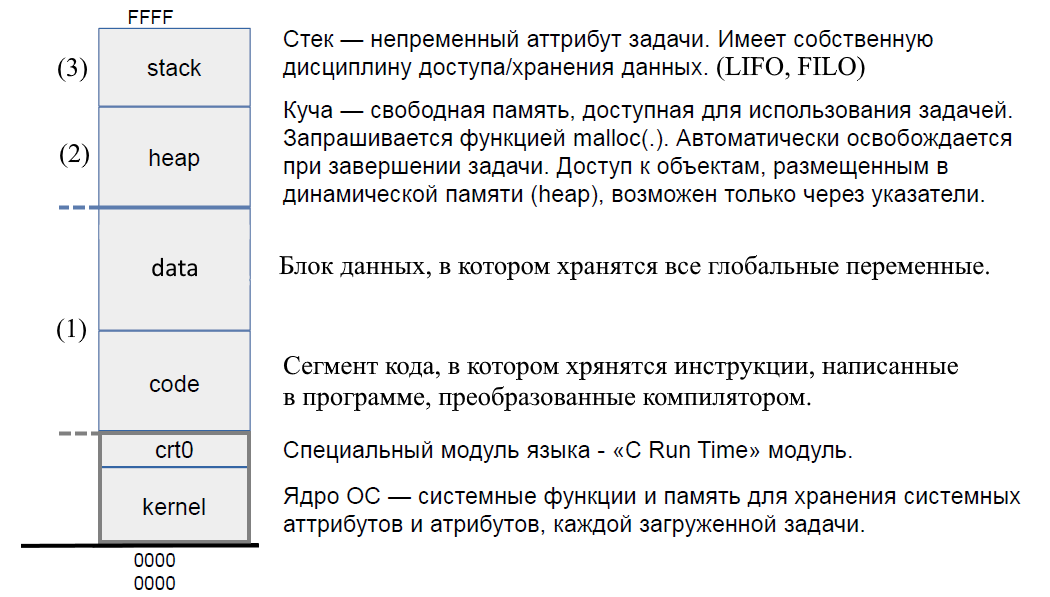
Указатель — тип данных, которая хранит номер – адрес ячейки памяти и, если указан, тип переменной.

Создание указателя: \*<имя переменной> = &<имя другой переменной>

Длина адреса зависит от разрядности машины (32/64 бита).

Только ОС может получить всю физическую память. Для пользовательских программ создается виртуальное адресное пространство, все адреса внутри программы считаются в рамках этого пространства.

**6. Модель памяти в OC Linux.**

****

**7. Время жизни объектов программы на языке С. Статическая, автоматическая, динамическая память.**

Статические глобальные объекты создаются после инициализации в программе и уничтожаются во время завершения.

Статические локальные объекты создаются после инициализации в программе и уничтожаются после выхода за границы фигурных скобок.

Динамические объекты создаются после выделения памяти под них программистом функциями malloc, calloc, realloc. Существуют, пока не будет вызвана функция free, в аргументы которой передан объект.

В программе существует 3 области памяти:

1. Статическая (блок кода и блок данных)
2. Динамическая (куча)
3. Автоматическая (стек: локальные переменные, аргументы функций (адреса возврата))

Статическая память - находится в code и data, выделяется один раз в момент запуска программы и в течение работы программы не изменяется.

Динамическая память – область памяти, которой можно управлять (решать, когда выделяется/освобождается/используется). Данные в куче, в отличие от стека и статической памяти, могут храниться непоследовательно.

Автоматическая память - изменяется в течение работы программы, но нельзя повлиять на то, как она выделяется и освобождается (система делает это автоматически, например, при объявлении новой переменной int в main). На стеке выделяется блок памяти длиной, соответствующей длине переменной, после чего Stack Pointer (SP) смещается вниз, там, где начинается свободная память.

Пример:

int main() {

char ch;

a(int b, int c);

…

}

int a(int b, int c) {

float fl;

f(int f);

…

}

|  |  |  |
| --- | --- | --- |
| ch | → | ch |
| c | c |
| b | b |
| Адрес возврата a | Адрес возврата a |
| fl | fl |
| f | return d() |
| Адрес возврата d |  |

Функции для работы с динамической памятью: malloc(), calloc(), realloc(), free().

Утечка памяти – ситуация, когда не освобождается память и теряется адрес, с которого начинается выделенная память (остается “висеть” в системе).

Концепции предотвращения утечек памяти:

1) Умные указатели – объекты, которые помимо всего прочего хранят информацию о владельце. При исчезновении объекта, память освобождается.

2) Сборщик мусора

(1-2 менее эффективные по быстродействию)

3) Ручное управление памятью

**8. Утечка памяти. Висячий указатель.**

Утечка памяти – ситуация, когда не освобождается память и теряется адрес, с которого начинается выделенная память (остается “висеть” в системе).

Пример:

int \*ptr;

ptr = (int \*) malloc(sizeof(int)); // выделение памяти

… // используем ее

ptr = (int \*) malloc(sizeof(int)); // повторное выделение

Перед вторым выделением памяти не было освобождение предыдущей - произошла утечка памяти.

Висячий указатель - это указатель, указывающий на недопустимые данные или данные, которые больше недействительны. Висячие указатели появляются при перемещении или удалении объектов, если при этом значение указателя остается без изменений. Даже если система перераспределяет освобожденную память под другие данные, указатель по-прежнему указывает на ячейку освобожденной памяти. Это может привести к непредсказуемому поведению программы. Если программа попытается записывать данные в память, используя такой указатель, данные могут повредиться, что может привести к трудноотслеживаемым ошибкам.

Примеры:

int \*ptr = (int \*) malloc(sizeof(int)) // динамически выделяем целочисленную переменную

\*ptr = 8; // помещаем значение в выделенную ячейку памяти

free(ptr); // освобождаем память

printf("%d\n", \*ptr); // разыменование висячего указателя приведет к неожиданным результатам

free(ptr); // попытка освободить память снова приведет к неожиданным результатам также

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

{

char \*ptr = NULL;

{

char c;

ptr = &c;

}

// c вне области видимости

// ptr висячий указатель

}

Один из распространенных методов выявления ошибки висячего указателя - установить указатель на NULL или на недопустимый адрес после того, как хранилище, на которое они указывают, было освобождено. При разыменовании нулевого указателя программа немедленно завершает свою работу, соответственно не будет возможности для повреждения данных или непредсказуемого поведения. Этот метод не помогает при наличии нескольких копий указателя.

**9. Выравнивание структур.**

Выравнивание - адрес переменной должен нацело делится на размер переменной в памяти.

В памяти поля структуры выравниваются по границе, кратной своему же размеру, т.е. однобайтовые поля не выравниваются, двухбайтовые выравниваются на четные позиции, четырехбайтные - на позиции кратные 4 и т.д.  
Пример:

1)

typedef struct {

char ch; // 1 байт занимает char, потом 3 пустых

int value; // 4 байта

} Foo; // sizeof(Foo) = 8

1 байт: ch

2 байт: пусто

3 байт: пусто

4 байт: пусто

5 байт: value[0]

6 байт: value[1]

7 байт: value[2]

8 байт: value[3]

2)

typedef struct {

char ch; // 1 байт под char, второй пустой

short id; // 2 байта

int value; // 4 байта

} Foo2; // sizeof(Foo2) = 8

1 байт: ch

2 байт: пусто

3 байт: id[0]

4 байт: id[1]

5 байт: value[0]

6 байт: value[1]

7 байт: value[2]

8 байт: value[3]

3)

typedef struct {

char ch; // 1 байт, второй пустой

short id; // 2 байта

short opt; // 2 байта, потом 2 пустых байта

int value; // 4 байта

} Foo3; // sizeof(Foo3) = 12

1 байт: ch

2 байт: пусто

3 байт: id[0]

4 байт: id[1]

5 байт: opt[0]

6 байт: opt[1]

7 байт: пусто

8 байт: пусто

9 байт: value[0]

10 байт: value[1]

11 байт: value[2]

12 байт: value[3]

Для компактного хранения структур и оптимизации кода лучше хранить поля, упорядочив их **по убыванию размера** в памяти.

Иногда можно дать указание процессору не использовать выравнивание, например, с помощью директивы #pragma pack (управляет выравниванием байт).

#pragma pack(push, 1) // установили размер выравнивания в 1 байт

struct Foo // описание структуры

{

// ...

};

#pragma pack(pop) // возврат к предыдущей настройке

Пример:

#pragma pack(1) // pack(n) - значение в байтах, используемое для упаковки

typedef struct {

char ch; // 1 байт

int value; // 4 байта

} Foo;

#pragma pack(push, 2) // ключевое слово push указывает, что укладываем в стек компилятора кол-во байт, на которое нужно выравнивать поля

typedef struct {

char ch; // занимает 2 байта

short sh; // занимает 2 байта

int i; // занимает 4 байта

} Foo2;

#pragma pack(pop) // удаление значение из стека компилятора

Таким образом.:

sizeof(Foo) = 5;

sizeof(Foo2) = 8;

#pragma pack(pop, n) - снять старое значение выравнивания и назначить n новым значением.

Инструкция #pragma pack (pop, r1, 2) эквивалентна:

#pragma pack (pop, r1) за которой следует

#pragma pack(2)

**10. Объявление и определение объектов.**

Для любого программного объекта существует 3 действия, связанные с его созданием:

* Объявление — «сообщение» компилятору, что он может использовать данный объект, который будет определен позже. Предоставляет компилятору тип и название объекта (происходит связывание имени с объектом);
* Определение - «сообщение» компилятору о том, что появится необходимость выделить память под объект;
* Инициализация - объекту присвоено первоначальное значение.

В случае с переменными, объявление является в том числе определением. Чтобы сделать только объявление переменной и не выделять под нее память, используется ключевое слово extern. (Т.е. int a; - это не объявление, а сразу определение вне зависимости, было ли присвоено ему значение; int a = 5; - это объявление (связывание имени a с объектом), затем определение (резервируется память под int) и инициализация (записывается значение 5)).

**Для каждого объявления должно быть ровно одно определение, иначе будет ошибка линковки.**Пример:

(good)

int f();

int f();

int f() {...}

(bad)

int f() {...};

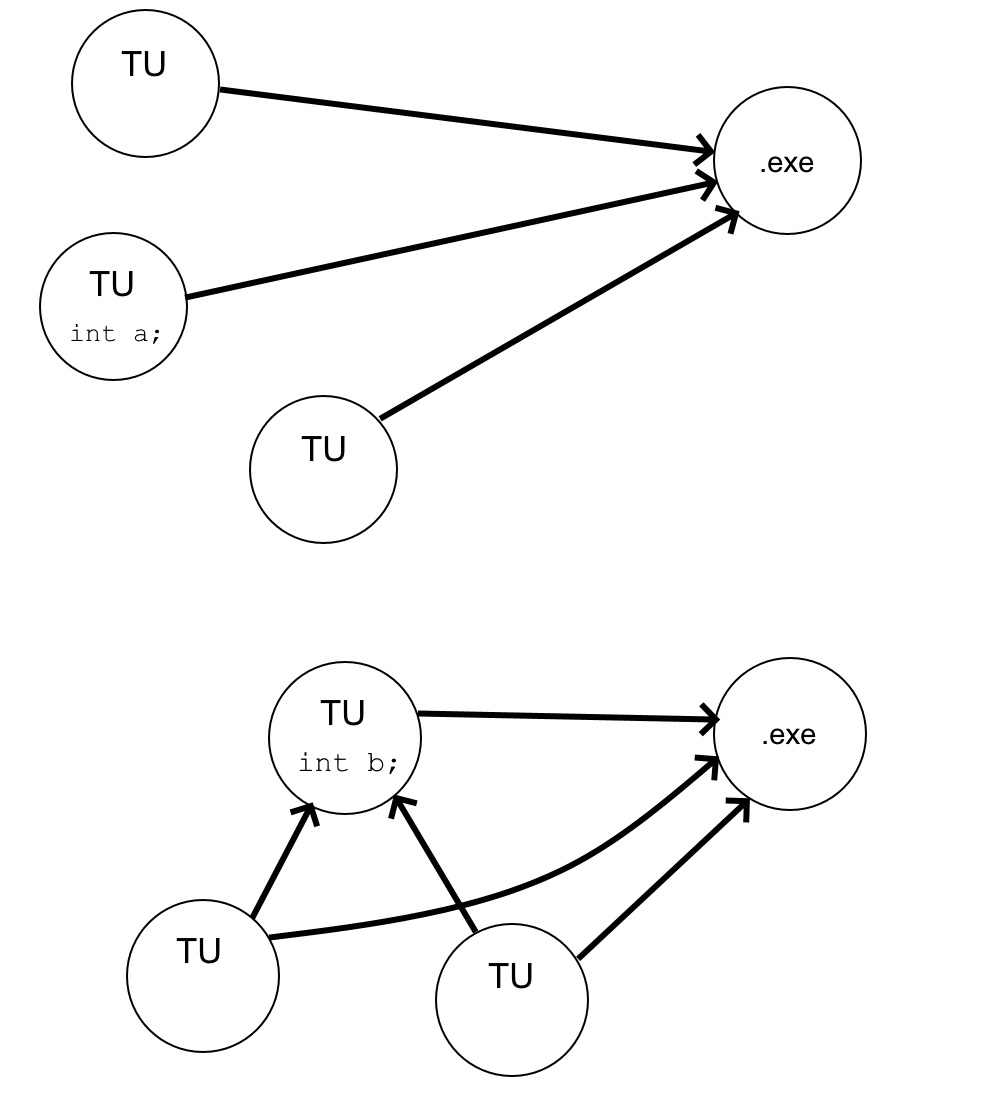
int f();

int f() {...}

**11. Внешнее и внутреннее связывание (external/internal linkage)**

Связывание бывает двух видов:

* internal - внутренняя (объект виден только внутри одного TU)
* external - внешняя (объект внутри одного TU виден другим TU)



Единица трансляции (Translation Unit, TU) - подаваемый на вход компилятора исходный файл .c со всеми включенными в него .h файлами.

Если данный символ (прим.: переменная или функция) является внутренним, то он виден исключительно в данном TU, и его определение не будет отнесено к какому-либо другому объявлению с таким же именем. Если же символ является внешним с точки зрения линковки, то такой символ может быть найден линковщиком в другом файле (например, если объявление в одном файле, определение - в другом).

|  |  |
| --- | --- |
| Объект | Linkage |
| Локальная переменная | internal |
| Глобальная переменная | external |
| Функции | external |

Ключевое слово static означает, что глобальная переменная или функция будут видны только в том файле, где определены. extern же означает, что глобальная переменная, определенная в другом файле, также будет известна в этом файле и также используется для доступа к функциям, определенным в других файлах.

**extern:** internal ⟶ external

**static:** external ⟶ internal

**12. Файловая декомпозиция. Заголовочные и исходные файлы. Защита от повторного включения заголовочных файлов.**

Заголовочные файлы — файлы с расширением .h, целью которых является удобное хранение набора объявлений объектов для их последующего использования в других программах. Как правило, заголовочные файлы не включают определения. Объявления в .h файлах свяжутся с определениями в .c файлах на этапе линковки.

Программы, как правило, включают в себя много исходных файлов, в каждом из которых находятся описания типов, переменных и т.д.. Чтобы имя можно было применять в разных исходных файлах для ссылки на определенный объект, оно должно быть описано как внешнее. Согласованность исходных файлов можно обеспечить помещением таких же описаний в заголовочные файлы (хедеры), после чего включить эти файлы везде, где требуются эти описания.

Единица трансляции (Translation Unit, TU) - подаваемый на вход компилятора исходный файл .c со всеми включенными в него .h файлами.

Если заголовочный файл содержит определения, то его повторное включение может привести к ошибке компоновки. Для предотвращения повторного включения существует директива #ifdef, которая указывает, что участок кода до #endif следует компилировать только в том случае, если объявления заголовочного файла не было. Директива #define же указывает, что заголовочный файл следует объявить. Альтернатива - #pragma once (в стандарт не вошла; помещается в начало заголовочного файла).

Ошибки линковщика:

* redefinition (переопределение) - 2 или больше определения относятся к одному и тому же объявлению:  
  int a(); int a() {...}  
   int a() {...}
* undefined reference (неопределенная ссылка) - функция объявлена, но для нее нет определения:  
  int a(); Ø

**13. Статические и динамические библиотеки функций.**

Статические (\*.а, \*.lib) — код функций вставляется в исполняемый файл.

**-** увеличение объема программы

**+** простое перемещение

Динамические (\*.so, \*.dll) — в исполняемый файл вставляется имя функции и ее адрес в библиотеке.

**-** необходим файл библиотеки при переносе

**+** не увеличивает объем программы

Отличие библиотек от программ — нет точки входа int main().

Пример подключения библиотек: gcc bar.c -lfoo, где foo имя библиотеки libfoo.a.

**14. Системы автоматизации сборки программ (СMake, qmake, nmake)**

Существует большое количество систем сборок, которые решают проблему кроссплатформенности make. Рассмотрим CMake, qmake, nmake.

CMake: Cross-platform make - не собирает(!) файлы, а генерирует Makefile под необходимую ОС. При вызове cmake нужно дополнительно вызвать make. Входной файл для CMake - CMakeLists.txt, в котором содержатся определенные команды.

add\_executable(project main.c) - указание, что собираем исполняемый файл

add\_library() - указание, что собираем библиотеку

set(VARIABLE The variable's value) - запись в переменную

include\_directories("headers/" "more\_headers/") - добавить папку с хедерами

Пример:

PROJECT(project)

set(SRC main.c lib.c)

set(HEADERS include.h)

add\_executable(project ${SRC} ${HEADERS})

qmake - кроссплатформенная система сборки, разработанная Qt. Аналогично CMake не собирает файлы. Используется в Qt Creator. Генерирует файл .pro, имеет собственный синтаксис.

nmake - кроссплатформенная система сборки, разработанная Microsoft. Используется в Microsoft Visual Studio.

**15. Утилита make. Makefile.**

make – утилита, предназначенная для автоматизации преобразования файлов из одной формы в другую. Правила преобразования находятся в скрипте с именем Makefile, который должен находиться в корне рабочей директории проекта. Сам скрипт состоит из набора правил:

1) цели (то, что данное правило делает)

2) реквизиты (то, что необходимо для выполнения правил и получения целей)

3) командами (выполняющими данные преобразования)

Если запустить  
make  
то программа попытается найти файл с именем по умолчанию Makefile в текущем каталоге и выполнить инструкции из него. Если в текущем каталоге есть несколько мейкфайлов, то можно указать на нужный вот таким образом:  
make -f <имя\_файла>

Инкрементная компиляция:

Разделение компиляции на два этапа: трансляция и линковка. Например, после изменения одного из исходных файлов достаточно провести его трансляцию и линковку всех объектный файлов. Пропускается трансляция не затронутых реквизитов, что сокращает время компиляции.

Фиктивные цели .PHONY: all clean install uninstall

В качестве make целей могут выступать не только реальные файлы. Две часто встречающиеся UNIX-команды:

1. $make // этой командой производят компиляцию программы

2. $make install // установка программы

Переменные:

Переменные в make представляют собой именованные строки и определяются так:

<VAR\_NAME> = <value string>

Пример:

CC=gcc # указываем компилятор

CFLAGS=-c -Wall # флаги компилятора

SOURCES=main.c lib.c # список исходников

OBJECTS=$(SOURCES:.c=.o)

TARGET=project # имя бинарника

all: $(TARGET)

$(TARGET): $(OBJECTS)

$(CC) $(OBJECTS) -o $@

%.o: %.c

$(CC) $(CFLAGS) $<

clean:

rm \*.o $(TARGET)

Автоматические переменные:

$@ Имя цели обрабатываемого правила

$< Имя первой зависимости обрабатываемого правила

$^ Список всех зависимостей обрабатываемого правила

**16. IDE: назначение, основные составные части, примеры.**

IDE – интегрированная среда разработки. Включает в себя:

1. Текстовый редактор

2. Транслятор (компилятор или интерпретатор)

3. Средства автоматизации сборки

4. Отладчик

5. Иногда присутствует git.

Были созданы для того, чтобы максимизировать производительность программиста благодаря тесно связанным компонентам, а именно:

А) Удобнее собирать проект - вместо множества команд нужно нажать пару кнопок.

Б) Есть статический анализ текста - ide показывает ошибки в синтаксисе или пропущенные знаки, а также сам может предполагать, какую именно переменную или функцию вы хотите написать, и дописать ее за вас.

Примеры — Microsoft Visual Studio, Code::bloks, Xcode.

**17. Процесс отладки ПО. Работа с отладчиком.**

Отладка — этап разработки компьютерной программы, на котором обнаруживают и устраняют ошибки.

Отладчик — программный инструмент, позволяющий программисту наблюдать за процессом выполнения программы. Отладчик позволяет запустить программу в режиме отладки, который используется для поиска ошибок в коде и позволяет просмотреть данные в коде на определенной строке. Для остановки на строке используются точки остановки или брейкпоинты. При этом сама строка не выполняется. В ide показывается как красная точка. Отладчик после остановки на брейкпоинте может перейти на следующую строчку, дойти до следующего брейкпоинта, войти или выйти из функции.

Отладка = тестирование + поиск ошибок + редактирование

**18. Системы контроля версий. Назначение, виды СКВ. СКВ Git. Основные понятия, операции. Ветвление.**

Система контроля версий позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям и т.д. Также среди базовых возможностей есть отслеживание изменений и контроль прав доступа. Такие системы наиболее широко используются при разработке ПО для хранения исходных кодов разрабатываемой программы.

Слияние версий - изменение из одной версии сливаются с изменениями другой таким образом, чтобы не нарушалась логическая связность проекта и не терялись данные.

Конфликт - ситуация, когда при слиянии нескольких версий сделанные в них изменения пересекаются между собой.

Виды СКВ:

* локальные (проект находится на одной локальной машине)
* централизованные (архитектура «клиент - сервер»)
* распределенные (проект хранится на сервере, другие машины хранят полную копию)

Система СКВ Git:

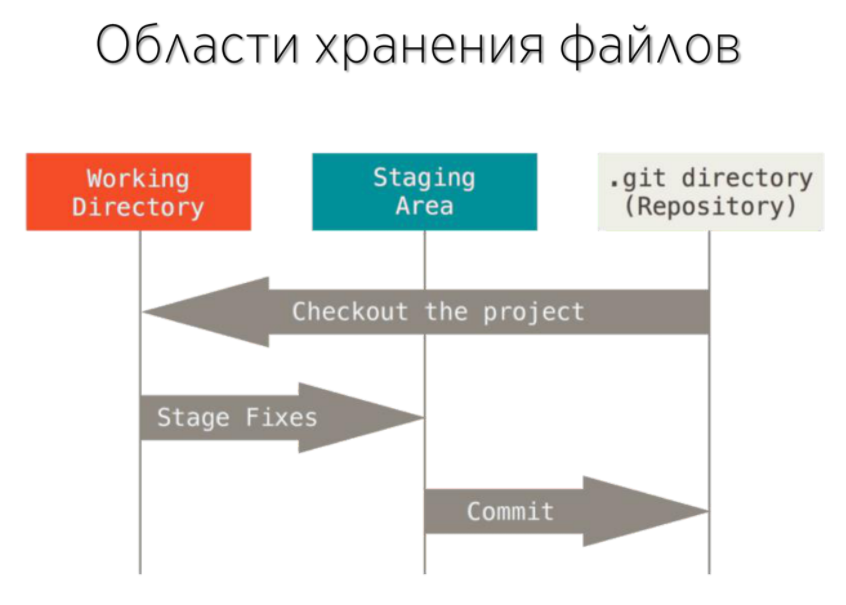
Git — это гибкая, распределенная (без единого сервера) СКВ. Предоставляет каждому разработчику локальный репозиторий, позволяет обмениваться сохраненными данными с другими пользователями. Работа над версиями проекта в Git может вестись в нескольких ветках, которые можно объединять, частично пересекать, уничтожать и т.д.

* в отличие от других СКВ, которые хранят версии в виде различий между файлами (diff), в Git используются снимки (snapshot)
* локальность операций
* целостность данных
* в основном только добавление
* можно работать как через консоль, так и через сторонние программы

Недостатки :

1. Unix — ориентированность. Отсутствует зрелая реализация, совместимая с другими ОС.

2. Не отслеживается изменение конкретных файлов, только изменение проекта целиком (может доставлять неудобства при работе над большим проектом)

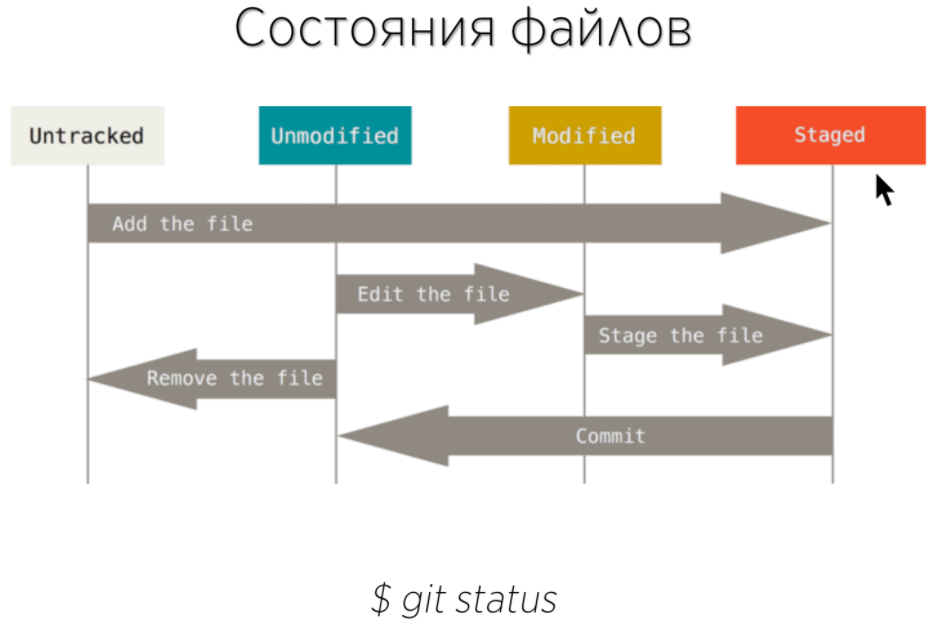


прим.: Staging area - область подготовленных файлов

checkout the project - обновление рабочей версии

Создание нового репозитория:

* git init
* git clone <repository>



Untracked - не отслеживаемые  
Unmodified - не измененные  
Modified - измененные  
Staged - подготовленные

После создания репозитория, все файлы в директории являются Untracked (при клонировании - unmodified)

Добавление файлов в индекс (Staged area):

* git add file
* git add file1 file2 file3
* git add .
* git add \*.c

Запись изменений: git commit (или git commit -m <message>)

Удаление файлов: git rm <files> но стоит учитывать, что файл полностью удалить из git практически невозможно, т.к. его предыдущие состояния были сохранены в ранних коммитах.

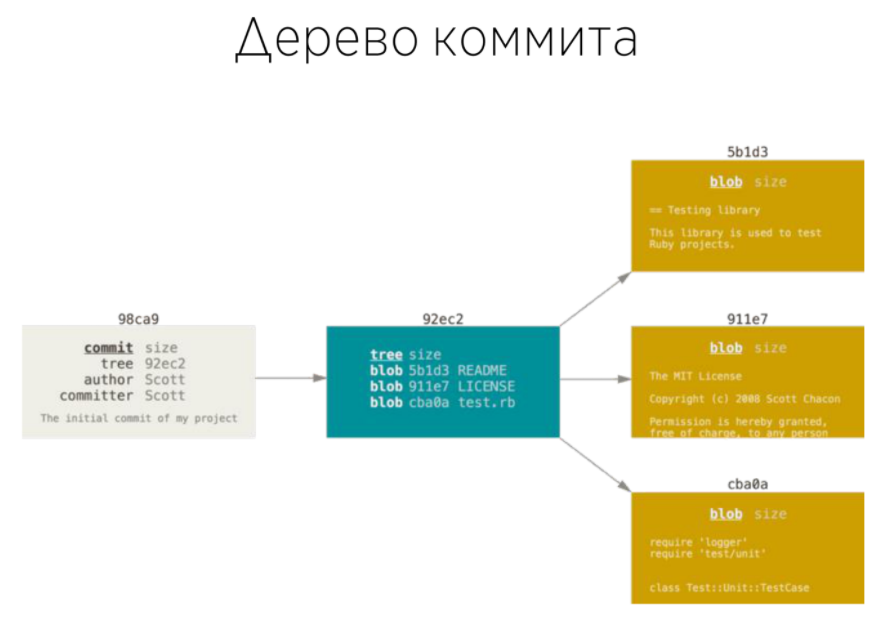
Перемещение файлов (переименование): git mv file\_from file\_to

Отмена изменений:

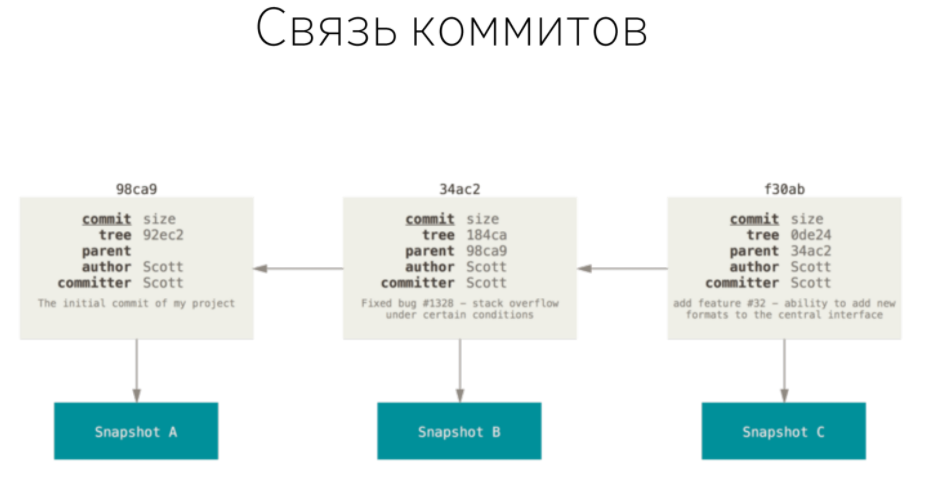
* удаление файла из индекса: git reset HEAD <files>
* отмена изменений в файле: git checkout --<files>
* изменить предыдущий коммит: git commit --amend

Работа с удаленными репозиториями:

* получение данных:  
  с перезаписью: git pull <remote> <branch>  
  получение снимка, на файлы не влияют: git fetch <remote> <branch>
* отправка данных: git push <remote> <branch>



Коммит - снимок текущего этапа проекта. Коммит содержит в себе: размер, указатель на корень дерева файловой структуры (файлы типа blob - массив двоичных данных) и метаданные (автор, коммитер и сообщение), а также указатель(-ли) на предыдущий коммит (получаем по результату хэш-функции к коммиту)



Ветвления:

Ветвь - указатель на один из коммитов, создается отдельной командой git branch <name>. Разработка ведется параллельно с изменениями в основной версии. Имя ветки по умолчанию - master. Указатель master, когда создаются первоначальные коммиты, указывает на последний коммит и движется вперед автоматически. Веток может быть много.

HEAD - специальный указатель на текущую ветку. Переключение веток производится по команде git checkout <branch> (создание ветки и переключение на нее - git checkout -b <branch>). Коммит происходит в ту ветку, на которую указывает HEAD. HEAD всегда только один.

Три вида операций могут приводить к необходимости объединения изменений:

1. Обновление рабочей копии — синхронизация рабочей копии до некоторого заданного состояния хранилища

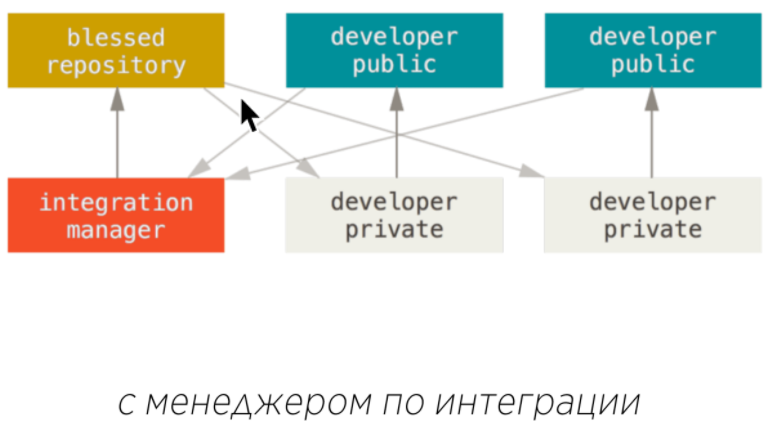
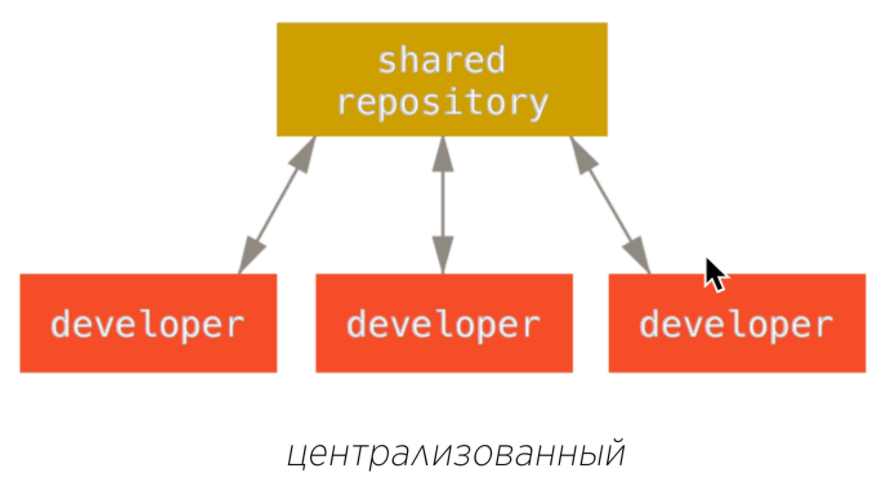
2. Фиксация изменений (коммит) — локальные изменения сливаются с изменениями, уже зафиксированными в основной версии

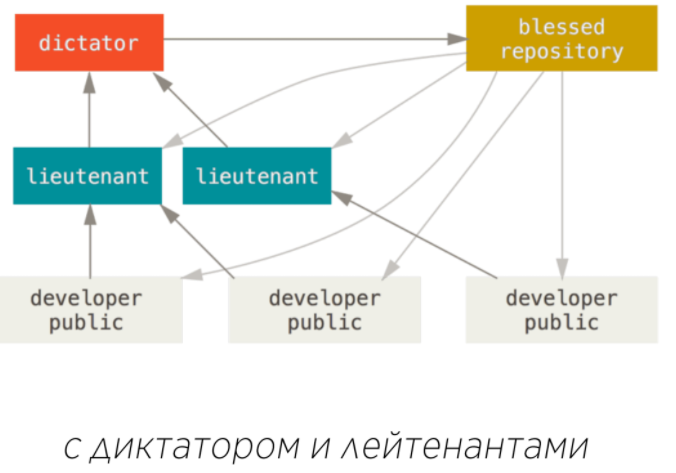
3. Слияние ветвей — объединение независимых изменений в единую версию документа (git merge <branch>). Для вливания ветки testing в master, необходимо переключится в master и выполнить команду merge.

Возможные случаи слияния веток:

* fast-merge (последний коммит текущей ветки является прямым продолжением целевой ветки. В этом случае для сливания Git не выполняет полноценное слияние, а просто переносит последний коммит текущей ветки в конец целевой ветки)
* merge commit (прямой связи коммитов нет, произошло расхождение веток. В таком случае создается отдельный коммит, который основан на 3 коммитах: предыдущий коммит целевой ветки, предыдущий коммит текущей ветки и коммит до расхождения веток)

Конфликт слияния - ситуация, когда в обоих сливаемых коммитах одновременно находится один (или несколько) и тот же файл(-ы).

**19. Распределенный процесс разработки. GitFlow.**

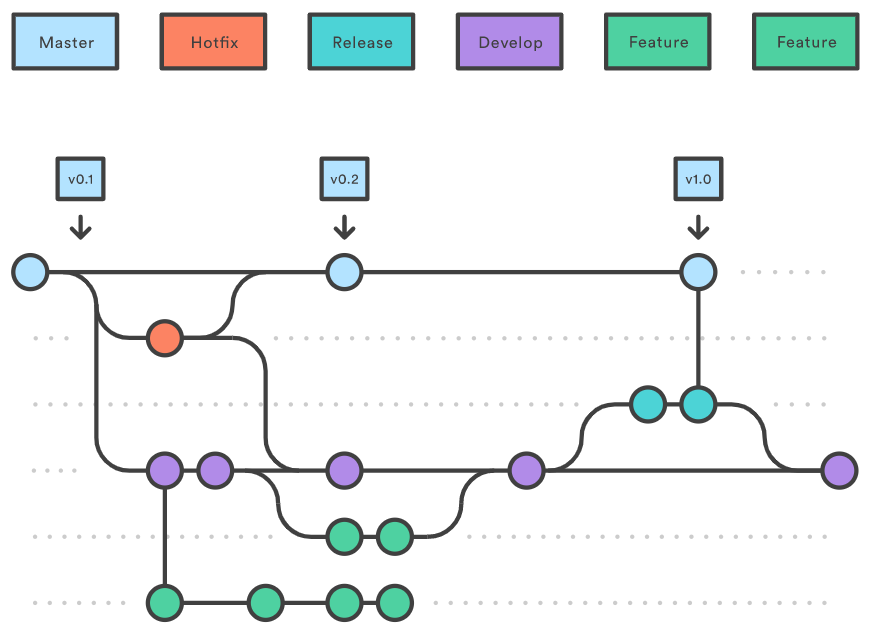
****

**• Централизованный: коммиты делаются в общий репозиторий; пользователи с равными правами.**

**• С менеджером по интеграции: есть общий репозиторий с финальными версиями, из которого программисты получают проект; программисты не могут делать коммиты в общий репозиторий; программисты делают коммит в личный публичный репозиторий; менеджер копирует из публичных репозиториев проект и при успешном тестировании добавляет в общий репозиторий**

**• С диктатором и лейтенантами: лейтенанты проверяют коммиты рядовых программистов, если коммит ценный для проекта, он посылается диктатору, который добавляет коммит в общий репозиторий**

**Git-Flow**

****

Git-Flow - методология, которая говорит о том, как организовать крупный проект так, чтобы контролировать его состояние в любой момент времени. В ее принципы входит:

• Master всегда собирается и не имеет не протестированных функций. Master хранит текущую версию проекта.

• Следующая версия проект разрабатывается в develop. В этой ветке постоянно проходит тестирование проекта.

• Новые функциональности разрабатываются в ветках feature, которые имеют начало и конец в develop.

• Для исправления багов, найденных в master, создается ветка hotfix, которая в конце концов сливается и в master, и в develop.

• Для слияния новой версии с master из develop выходит новая ветка release, которая впоследствии сливается с master.

**20. Форматирование исходного кода.**

Смотри гайдлайн: [КМБО-02/05-20 | РПО и ПЗР (vk.com)](https://vk.com/rpo_pzr_20?w=wall-198452929_13%2Fall)

Правила форматирования исходного кода нужны для того, чтобы обеспечить максимальную читаемость кода для других людей и свести код к единой стилистике.