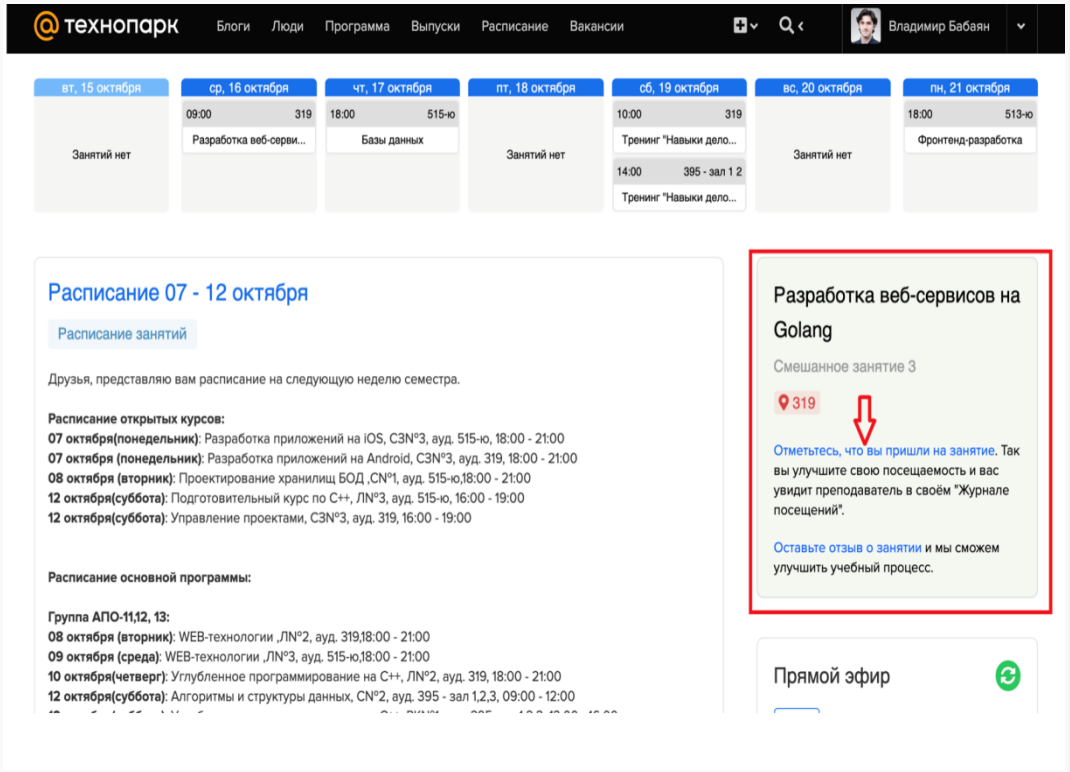
Подготовительная программа по программированию на языках C / C++

Лекция № 1

Отметьтесь на портале

- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения



Преподаватели (1 / 3)

Алексей Халайджи

Выпускник кафедры ИУ6 МГТУ им. Баумана, Технопарка
и Школы Анализа Данных (трек «Анализ данных в прикладных науках»)

Более 5 лет занимаюсь преподавательской деятельностью для школьников и студентов (Сириус, Технопарк, MADE, кафедра ИУ6 МГТУ им. Баумана и др.)

К команде @ mail.ru group присоединился в 2015 г.

В настоящий момент выхожу на защиту кандидатской диссертации

Преподаватели (2 / 3)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Михаил Кириченко

Руководитель группы в департаменте рекламных технологий

С 2012 г. занимается разработкой ядра рекламной системы Mail.Ru

Ведёт лекции по языку Си

Преподаватели (3 / 3)

МГТУ им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Игорь Анфёров

Старший программист группы серверной разработки B2B-приложений

Работает над MyTeam, ICQ, Агентом Mail.Ru

Ведёт лекции по ООП и системному программированию на C++

Цель и структура курса

Цель курса – **познакомить** слушателя с языками Си и C++, основными **парадигмами разработки** на этих языках, **инструментарием**, используемым при разработке современных приложений на C/C++, научить **работать в команде** и разрабатывать **с нуля проекты** средней сложности **на C/C++**.

Структура курса:

- 8 лекций
- 3 мастер-класса
- 2 хакатона
- 4 синкапа

В конце курса – открытая защита командных проектов

Чему научимся?

- **работать** с **памятью** на языке Си
- **разрабатывать контейнеры** на языке Си
- **взаимодействовать** с **операционной системой**
- реализовать **сетевое взаимодействие**
- применять **объектно-ориентированное программирование**
- писать **безопасный код**
- **разрабатывать качественный** и **переиспользуемый** код
- **использовать стандартную библиотеку** C++ (STL)
- **применять** современные системы **непрерывной интеграции**
- **работать** в **команде**
- **планировать** своё **время**
- **представлять результаты** своей работы в составе **проектной группы**

Модульно-рейтинговая система (1 / 4)

- **Модуль № 1. Язык Си, разбиение на команды и начало проектирования командных проектов**

4 практические задачи (индивидуально на языке С):

- ДЗ № 1 – до 5 баллов. Занятие выдачи – Л1. Дедлайн – Л3;
- ДЗ № 2 – до 8 баллов. Занятие выдачи – Л2. Дедлайн – С2;
- ДЗ № 3 – до 8 баллов. Занятие выдачи – Л3. Дедлайн – РК1;
- ДЗ № 4 – до 10 баллов. Занятие выдачи – С2. Дедлайн – Л6.

+ подготовка к первому синкапу в команде: РК № 1 – до 5 баллов

Модуль № 2. Объектно-ориентированное и сетевое программирование на языке С++, подготовка к первому хакатону

2 практических задания (индивидуально на языке С++):

- ДЗ № 5 – до 8 баллов. Занятие выдачи – Л4. Дедлайн – С3;
- ДЗ № 6 – до 13 баллов. Занятие выдачи – Л6. Дедлайн – Л8;

+ подготовка ко второму синкапу в команде: РК №2 – до 10 баллов

Модульно-рейтинговая система (2 / 4)

- **Модуль № 3. Обобщённое программирование на C++. Библиотека STL. Доработка MVP и подготовка к защите проекта**

1 практическое задание (индидуально на языке C++)

- ДЗ № 7 – до 8 баллов;

+ подготовка к третьему синкапу: РК № 3 – до 10 баллов;

+ подготовка к предзащите: РК № 4 – получение допуска к защите

+ открытая защита командных проектов – до 15 баллов

- полезность итогового продукта – до 5 баллов
- технологическая нетривиальность и обоснованность – до 5 баллов
- мастерство выступления – до 5 баллов

Модульно-рейтинговая система (3 / 4)

Действует система дедлайнов!

Проверяются только те задания**,** которые **сданы строго до дедлайна**.

Общая схема **сдачи домашних заданий**:

- вы выполняете ДЗ;
- до дедлайна вы должны прислать его своему ментору на проверку. Если не успели – получаете 0 баллов;
- можно отправить ментору более одного раза;
- после проверки работы ментором она отправляется преподавателю;
- преподаватель выставляет баллы и может оставить замечания
- (опционально) можно исправить замечания в течение ровно 1 недели после оценки работы преподавателем. Если всё устраивает – можно ничего не исправлять

Модульно-рейтинговая система (4 / 4)

Итоговая оценка выставляется в соответствии со следующей шкалой перевода:

- **от 0 до 69** – «неудовлетворительно»
- **от 70 до 79** – «удовлетворительно»
- **от 80 до 89** – «хорошо»
- **от 90 и более** – «отлично»

Организационные положения (1 / 2)

Блог дисциплины размещён по адресу:

https://park.mail.ru/blog/view/13/

Крайне рекомендуется:

- подписаться на него;
- задавать вопросы;
- участвовать в опросах и обсуждениях.

Все материалы, касающиеся курса, будут своевременно публиковаться в нём и дублироваться в чате в Telegram:

https://t.me/joinchat/jUD3YSNYJ5Q4Zjky

Организационные положения (2 / 2)

- длительность занятия – 4 часа с 1 перерывом продолжительностью до 10 минут;
- место проведения занятия следует проверять заранее на портале;
- в начале каждого занятия необходимо отмечаться на портале в электронном журнале;
- вопросы во время лекции стоит задавать по мере их поступления или индивидуально на перерыве или после занятия;
- вопросы вне аудиторных занятий лучше задавать в блоге, Telegram-чате или через личные сообщения на портале;
- после каждого занятия следует уделять пару минут для того, чтобы оставить отзыв по занятию на портале

Web-ресурсы и онлайн-книги

- C Programming — http://en.wikibooks.org/wiki/C_Programming
- Google C++ Code Style — https://google.github.io/styleguide/cppguide.html
- Code Style Guidelines | WebKit — https://webkit.org/code-style-guidelines/
- LLVM Coding Standards — LLVM5 documentation: http://llvm.org/docs/CodingStandards.html
- Справка по языкам C / C++ — http://en.cppreference.com/w/
- The C++ Resources Network — http://www.cplusplus.com/
- Standard C++ — http://isocpp.org/
- Cpp Core Guidelines — https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- More C++ Idioms — http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms
- Stack Overflow — http://stackoverflow.com/
- YouTube-канал конференций CppCon — https://www.youtube.com/user/CppCon
- YouTube-канал рабочей группы C++ — https://www.youtube.com/channel/UCJ9v015sPgEi0jJXe_zanjA

Модуль № 1. Язык Си

- Лекция № 1. Вводная лекция. Структура программы на Си. Основные конструкции языка
- Семинар № 1. Мастер-класс по работе с git и GitHub, знакомство с менторами и разбиение на команды
- Лекция № 2. Структуры, массивы, строки, работа с файлами. Рефакторинг
- Лекция № 3. Библиотеки, память, контейнеры в языке Си
- Семинар № 2. Мастер-класс «JSON-парсер»

- Синкап 1 (РК № 1). Обсуждение командных проектов

Рекомендуемая литература: модуль № 1 (1 / 2)

- Керниган Б., Ритчи Д. Язык программирования С. — Вильямс, 2012. — 304 с.
- Прата С. Язык программирования С. Лекции и упражнения. — Вильямс, 2013. — 960 с.
- Шилдт Г. Полный справочник по С. — Вильямс, 2009. — 704 с.
- 2011 CWE / SANS Top 25 Most Dangerous Software Errors:: http://cwe.mitre.org/top25/
- CERT Secure Coding Standards: https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards
- Common Weakness Enumeration: https://cwe.mitre.org/.

Рекомендуемая литература: модуль № 1 (2 / 2)

- King, K. C Programming: A Modern Approach, 2nd ed. (W. W. Norton & Co., 2008).
- MISRA C:2012 (MISRA C3). Guidelines for the Use of the C Language in Critical Systems (March 2013).
- Open Web Application Security Project: https://www.owasp.org/index.php/Main_Page
- Seacord, R. Secure Coding in C and C++, 2nd ed. (Addison-Wesley Professional, 2013).
- Seacord, R. The CERT® C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems (Addison-Wesley Professional, 2014).
- Sonar CFamily C Rules: https://www.sonarsource.com/why-us/products/codeanalyzers/sonarcfamilyforcpp/rules-c.html
- SonarQube SQALE Plugin: http://qualilogy.com/en/legacy-application-refactoring-sqale-plugin-2/

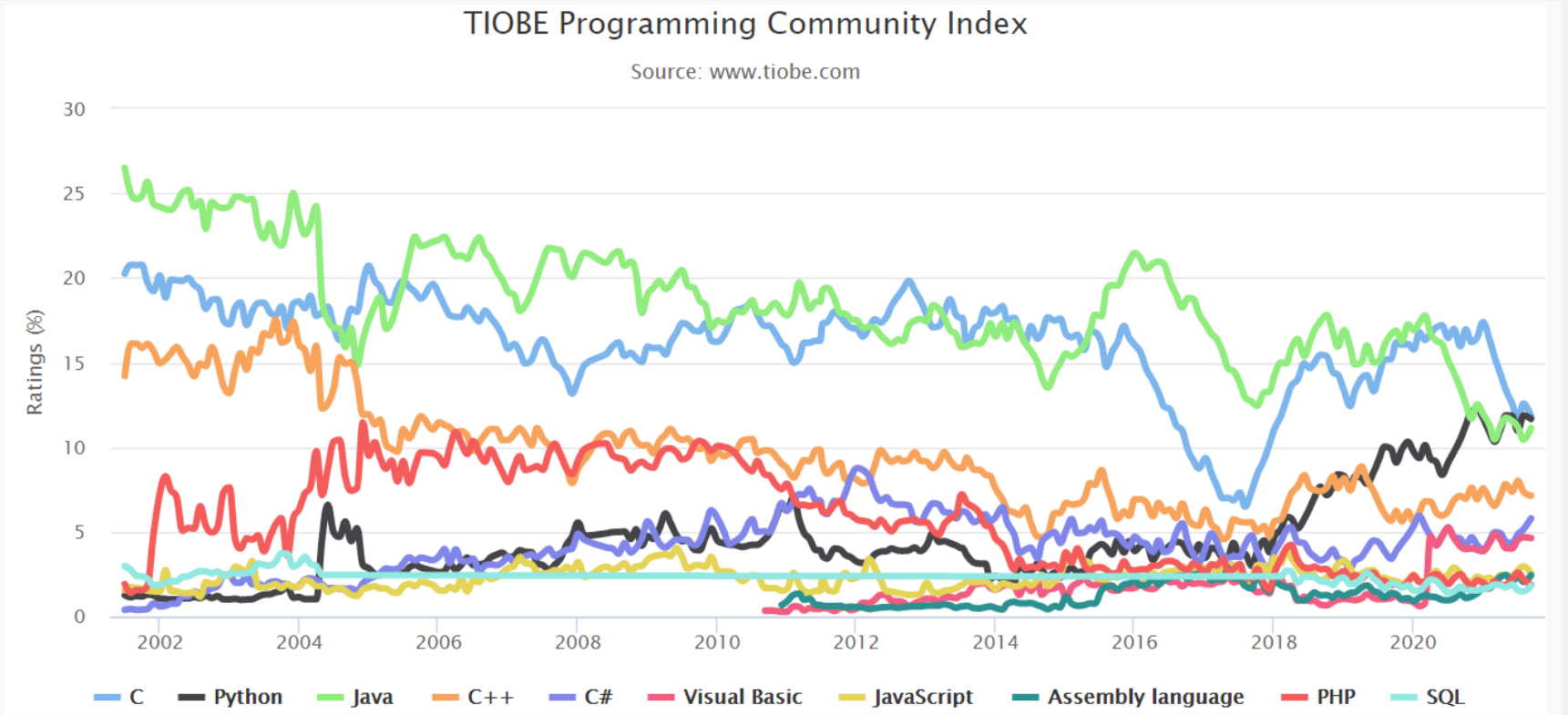
Лекция № 1. Структура программы на Си. Основные конструкции языка

1. Основные понятия теории программирования
2. Примитивные типы данных и операции над ними на языке Си
3. Описание и определение переменных и функций
4. Обзор базовых функций ввода/вывода на языке Си
5. Структура программы на языке Си
6. Основные конструкции языка Си
7. Этапы выполнения программы на языке Си
8. Структура памяти программы
9. Рекурсивные функции
10. Постановка домашнего задания № 1

Язык Си сегодня

- **2021** — активное применение языка в практике программирования задач общего и специального назначения:
  - ядра ОС:   
  - инструментальные средства:    
  - системы управления БД и Web-серверы и пр.   
  - проекты Mail.Ru Group: tarantool, Почта@Mail.Ru и пр.
- В сентябре 2021 г. TIOBE Programming Community Index языка Си составляет 11,83% (1-е место), а сам язы
  в нем занимает 1 – 2-е места с 1985 г.
  (конкурируя, главным образом, с Java).

Рейтинг языков согласно TIOBE за период с 2002 года до наших дней



Язык **Си** является **языком 2017 и 2019 годов** по версии **TIOBE** в связи с мощным подъёмом **мобильных устройств** и ростом популярности **низкоуровневого ПО** в **автомобильной промышленности**.

Основные понятия теории программирования

- **Программа** – упорядоченный набор инструкций, позволяющий по заданному алгоритму преобразовать входные данные в выходные
- **Операционная система** – сложная программа, управляющая всеми процессами, работой с памятью и файловой системой
- **Данные** – информация в структурированном виде
- **Тип данных** – подмножество данных с чёткой интерпретацией и с ограниченным набором допустимых операций
- **Переменные** – изменяемые данные, которым присвоено имя
- **Константы** – неизменяемые данные
- **Функции** – подпрограмма, имеющая своё имя, принимающая на вход аргументы и возвращающая результат их обработки
- **Трансляция** – перевод программы с одного языка программирования на другой
- **Компиляция** – трансляция кода на языке высокого уровня в язык ассемблера или машинных команд

Примитивные типы данных и операции над ними в языке Си

- Базовые типы данных языка Си:
  - **char** – целое число, занимающее 1 байт (ASCII-символ);
  - **int** – целое число, обычно занимающее 4 байта;
  - **float** – вещественное число одинарной точности (4 байта);
  - **double** – вещественное число двойной точности (8 байт).
- Есть и другие типы данных, размер которых может отличаться от платформы к платформе:
  - **char** <= **short** <= **int** <= **long** <= **long long**
- Тип может быть знаковым и беззнаковым
  - [**signed**] **char** принимает значения от -128 до +127;
  - **unsigned char** – от 0 до 255
- Логический тип отсутствует. Значение «0» (и приводимые к нему) считается ложью, всё остальное – истина.

Устройство знаковых и беззнаковых типов

- Компьютер хранит все значения в двоичном коде
  - каждый символ двоичного кода занимает ровно один бит и может принимать только одно из двух возможных значений: 0 или 1
- 00000000 – 8-разрядный «ноль»
- 00000001 – 8-разрядная «единица»
- 01111111 – 8-рязрядное число, равное 2^7 – 1 = 127
- 10000000 может быть воспринято как 8-разрядное число 128, а может – как -128, т.к. первый бит – знаковый, а остальные – в дополнительном коде
- Дополнительный код отрицательного числа – все информационные биты инвертируются, и к полученному числу прибавляется 1
- -1 в дополнительном коде – это: 00000001 -> 11111110 –> 11111111 что также можно интерпретировать как беззнаковое число 255

Двойственность знаковых и беззнаковых чисел. Дополнение и переполнение

Типы данных с фиксированным размером

- Размер типов данных short, int, long, long long не фиксируется стандартом и зависит от компилятора, операционной системы и пр.
- Существуют специальные типы данных с **фиксированным размером**:
  - **знаковые:целочисленные:** int8_t, int16_t, int32_t, int64_t
  - **беззнаковые целочисленные**: uint8_t, uint16_t, uint32_t, uint64_t
  - **знаковые целочисленные «как минимум»:** int_least8_t, int_least16_t, int_least32_t, int_least64_t
  - **беззнаковые целочисленные «как минимум»:** uint_least8_t, uint_least16_t, uint_least32_t, uint_least64_t
- **Достоинство** – уверенность в том, что необходимый диапазон значений поместится в типе данных
- **Недостаток** – доступно не на всех операционных системах и компиляторах (например, 64-разрядные типы на 32-разрядных системах или 8-битных микроконтроллерах). **Без явной необходимости – не использовать!**
- Определены в заголовочном файле <stdint.h>
- Границы допустимых значений каждого типа описаны в <limits.h>

Операция sizeof и тип size_t

- **Унарная операция** sizeof:
  - допускает скобочную и бесскобочную (только для переменных) нотацию: sizeof a или sizeof(T);
  - возвращает объем памяти, выделенной под объект простого или составного типа, в байтах как значение переносимого типа size_t, являющегося псевдонимом одного из базовых беззнаковых целых типов (ср. int32_t и пр.);
  - не учитывает возможного выравнивания объекта.
- Использование вычисляемых компилятором конструкций вида sizeof(T) не влияет на производительность кода, но **повышает переносимость.**

Операции над типами данных (1 / 2)

```
/* арифметические: +, -, *, /. Тип результата деления зависит от
Типов операндов - если хотя бы один из них вещественного типа,
то и результат - вещественного. Если оба целые - результат целый,
% - остаток от деления целых чисел */

/* логические: ! (отрицание), && (И), || (ИЛИ)
!!a - нормализация истинного выражения к 1, а ложного - к 0 */

/* битовые: ~ (НЕ), & (И), | (ИЛИ), ^ (ИСКЛЮЧАЮЩЕЕ ИЛИ, XOR)
Например: 00110010 & 01101001 = 00100000;
Битовые сдвиги: <<, >>
Например, 1 << 2 преобразует 00000001 в 00000100,
                 а 10 >> 3 - 00001010 в 00000001 */
```

Операции над типами данных (2 / 2)


```
/* инкремента/декремента: ++a, a++, --a, a--
префиксная форма - сначала изменяет, потом возвращает,
постфиксная форма - сначала возвращает текущее, потом изменяет */

/* присваивания: =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=
Разворачиваются одинаково: a += b -> a = a + b */

/* сравнения: <, >, <=, >=, ==, !=
(* операция присваивания «=» возвращает результат присваивания -
следите в условных выражениях за использованием ==) */

/* тернарный оператор:
<Условие> ? <Выражение, выполняемое если условие истинно>
          : <Выражение, выполняемое если условие ложно> */
```

Приоритет операций

Precedence	Operator	Description	Associativity
1	++ -- ( ) [ ] . -> (*type*) { *list* }	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement[note 1] Unary plus and minus Logical NOT and bitwise NOT Cast Indirection (dereference) Address-of Size-of[note 2] Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-left
14[note 4]	= += -= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

https://en.cppreference.com/w/c/language/operator_precedence

Определение переменной на языке Си


```
// это однострочный комментарий

/* это многострочный комментарий

Синтаксис определения переменной:
<Тип> <Имя> [= <Инициализатор>];
*/

int a = 10;
double b = 5.3;

// можно (но не нужно!!!) определять несколько переменных сразу
int c, d = 5;
```

Как нельзя определять переменные


```
int for = 10; // for - ключевое слово языка

double 5t; // имя переменной должно начинаться с буквы или _

char small letter = 10; // имя переменной может содержать только буквы,
                        // цифры и знак _

int a = 10;
double a = 20; // так нельзя, т.к. тип переменной задаётся при её
               // определении и не меняется на протяжении всей работы

int a = 10;
double A = 20; // так можно, т.к. регистр имеет значение
```

Как не стоит определять переменные


```
int uninitialized; // без инициализации

int first = 10, second = 50; // несколько сразу

int a = 10; // что такое «a»?

int a = 10;
// 1000 строк позже
// сделать что-то с a -> определяйте переменные
// как можно ближе к месту их реального использования!

int arg1;
int arg2; // не используйте 1, 2 и др. суффиксы
```

Синтаксис определения функции


```
// Функция – подпрограмма, обладающая именем, принимающая на
// вход аргументы и возвращающая результат
/* Общий синтаксис определения функции:
<возвращаемый тип или void> <имя>([<список аргументов> или void]) {
    <тело функции> - содержательная часть функции
}
<список аргументов> := <тип аргумента> <имя аргумента>
<список аргументов> := <тип аргумента> <имя аргумента>,
                       <список аргументов>
*/
// Примеры:
int power(double base, int n) { /* … */ }
void print_hello(void) { printf(“Hello!\n”); }
```

Стандартные функции ввода/вывода


```
/* Функция ввода с клавиатуры
Возвращает количество считанных значений или -1 */
int scanf(<Форматная строка>, <Список адресов переменных>);
/* Функция вывода на экран
Возвращает количество успешно выведенных символов */
int printf(<Форматная строка>, <Список выражений>);

<Форматная строка> имеет вид:
%[-][<Целое 1>].[<Целое 2>]<Формат>,
где «-» - выравнивание по левой границе,
    <Целое 1> - ширина поля вывода
    <Целое 2> - количество цифр дробной части вещ. числа
```

Некоторые допустимые форматы

d, i – целое десятичное число (int). I – автоматически определяет **0x**A и **0**6 как числа в 16- и 8-ричной системах счисления (например, **0x**A -> 10, **0**12 -> 10, **0**79 -> 7);

u – целое десятичное число без знака (unsigned int);

o – целое число в восьмеричной системе счисления;

x,X – целое число в 16-ричной системе счисления, %4x – без гашения незначащих нулей, X – буквы верхнего регистра;

f – вещественное число (float) в форме с фиксированное точкой;

e,E – вещественное число в форме с плавающей точкой (например, 5.7e-317 или 6.2E+13);

c – ASCII/ANSI-символ (задаваемый кодом в один байт, типом char);

s – символьная строка;

\n – переход на следующую строку, \t – табуляция,

\n hhh – вставка символа с 16-ричным ANSI-кодом hhh

%[<Множество допустимых символов>] – только указанные (например, %[A-Z]);

%[^[Множество недопустимых символов]] – любые кроме указанных (например, %[^ \n]);

и др. (zu, lu, llu, lf, … - см. документацию)

Примеры использования функций ввода/вывода

```
char a = ‘A’;
printf(“%4X - %c\n”, a, a); // вывод 16-ричного ASCII-кода
                            // символа и самого символа
int a = 0;
int b = 0;
scanf(“%d %d”, &a, &b);// нужно передавать адреса переменных,
                       // а не их значения, для чего
                       // используется оператор взятия адреса &

scanf(“%*[^\n]”); // считывание из буфера ввода всех мусорных
                  // символов до перевода строки
                  // например, ожидали число, а ввели строку
                  // символ * просто пропускает сканируемый эл-т
```

Константы


```
// Константы - неизменяемые данные

// большими буквами через _
#define MY_MAIN_CONSTANT 500 // #define не проверяет тип!

// через квалификатор const
const int GLOBAL_CONSTANT = 10;
// или через «east const» нотацию (тип читается справа налево)
int const GLOBAL_CONSTANT = 10;

int a = MY_MAIN_CONSTANT;
a = a + GLOBAL_CONSTANT; // поместить в a результат выражения
GLOBAL_CONSTANT = 20; // НЕЛЬЗЯ!
```

Именованные перечисления


```
// Именованные перечисления (enum - от слова enumeration)
// позволяют задать именованную группу констант

enum Day { MON = 1, TUE, WED, THU, FRI, SAT, SUN };

enum Day day = TUE;

printf("Today is %d.” // можно таким образом разрывать строки
       “Tomorrow it will be %d.”
       ”In a week it will be %d\n", day, day + 1, day + 7);

/*
Выведет
Today is 2, tomorrow it will be 3. In a week it will be 9
*/
```

Первая программа на языке Си


```
#include <stdio.h>
int main(void) { // можно просто int main() {
    printf(“Hello, world!”);
    return 0;
}
```

/* Функция main – точка входа в программу (вызывается операционной системой после загрузки программы в память)

return 0 – возврат значения 0 из функции (для main – статус выполнения программы: 0 – успех, не 0 – ошибка)

#include – директива препроцессора, включающая в текущий файл содержимое файла в скобках <>

stdio.h содержит **описание** функции вывода printf */

Описание и определение функции


```
int main(void); // описание функции (declaration)
// нужно для компилятора, чтобы ввести идентификатор
// и для человека, чтобы понять интерфейс

int main(void) { // определение функции (definition)
        /* определение подразумевает наличие реализации */
}

/*
Описаний может быть сколько угодно (0, 1, 2, …)
Определение д.б. ровно одно - One Definition Rule (ODR)
*/
```

Ошибки описания и определения (1 / 3)


```
// program.c
int main() {
    int a = b;
    int c = foo(a);
    return c;
}
// компилируем программу – возникли ошибки описания
$ gcc program.c
```

```
program.c: In function ‘main’:
program.c:2:13: error: ‘b’ undeclared (first use in this function)
    int a = b;
            ^
program.c:2:13: note: each undeclared identifier is reported only once for each function it appears in
program.c:3:13: warning: implicit declaration of function ‘foo’ [-Wimplicit-function-declaration]
    int c = foo(a);
            ^~~
```

Ошибки описания и определения (2 / 3)


```
// program.c
int foo(int); // добавили описание функции
int main() {
    int b = 10; // добавили определение переменной
    int a = b;
    int c = foo(a);
    return c;
}
// компилируем программу - возникла ошибка определения
$ gcc program.c
/tmp/ccLS4uJC.o: In function `main':
program.c:(.text+0x1b): undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

Ошибки описания и определения (3 / 3)


```
// program.c
int foo(int a) { return a; } // определили функцию
int main() {
    int b = 10;
    int a = b;
    int c = foo(a);
    return c;
}
// компилируем программу - ошибок нет
$ gcc program.c
```

Нарушения ODR


```
// program.c
int foo(int a) { return a; } // определили функцию
int main() {
    int b = 10;
    int a = b;
    int b = foo(a); // переопределили
                    // переменную b
    return b;
}
// переопределили функцию foo
int foo(int a) { return a * a; }
// компилируем программу - снова ошибки
// определения (redifinition)
$ gcc program.c
```

```
program.c: In function ‘main’:
program.c:5:9: error: redefinition of ‘b’
     int b = foo(a); // переопределили переменную b
         ^
program.c:3:9: note: previous definition of ‘b’ was here
     int b = 10;
         ^
program.c: At top level:
program.c:8:5: error: redefinition of ‘foo’
 int foo(int a) { return a * a; } // переопределили функцию
     ^~~
program.c:1:5: note: previous definition of ‘foo’ was here
 int foo(int a) { return a; } // определили функцию
     ^~~
```

Как стоит размещать код в программе


```
// раздел описаний (используемый и предоставляемый интерфейс файла программы)
extern int a; // описание глобальной переменной a, которая определена
              // где-то ещё (обычно для использования внешней переменной)
              // не создавайте глобальные переменные!
void print_to_n(int n); // описание функции, которая определена ниже

int main(void) {
    /*
        Реализация функции main, использующая интерфейс
        (без интерфейса компилятор не понял бы, что это за символы)
    */
}

// раздел определений (реализации функций интерфейса)
void print_to_n(int n) { /* … */ }
```

Принцип единственной ответственности (SRP)

- Старайтесь **проектировать функции** так, чтобы они выполняли **ровно одну задачу** (имели **одну зону ответственности**)
- Если задача **слишком сложная** – её необходимо **декомпозировать**, и тогда **зоной ответственности** исходной функции является **вызов дочерних функций** в **правильном порядке** (**диспетчеризация потоков управления**)
- **Ответственность функции main** – в **передаче управления** от **операционной системы программе**
- **Логика не должна содержаться в функции main**!
- Но **функция main** может выполнять **роль диспетчера**, **передавая управление другим функциям** и **возвращая полученный в итоге результат**
- Функция **main** также передаёт **входные аргументы программы**

Работа с аргументами командной строки: пример


```
int main(int argc, char *argv[]) {
    if (argc == 1) {
        printf("Usage: %s <args>", argv[0]);
        return 1;
    }
    for (int i = 1; i < argc; ++i) {
        printf("%s ", argv[i]);
    }
    return 0;
}
// Примеры работы
$ ./a.out              -> Usage: ./a.out <args>
$ ./a.out Hello world -> Hello world
```

Основные конструкции языка Си (1 / 6)

- Выражение – валидный набор операторов на языке Си, заканчивающийся точкой с запятой:
  - **int** a = 10;
  - a = b = 5 – эквивалентно b = 5; a = 5;
  - c = (a = 5, b = a * a) – эквивалентно a = 5; b = a * a; c = b;
  - a = (b = s / k) + n – эквивалентно b = s / k; a = b + n;
  - c = (a > b) ? a : b – эквивалентно «если a > b, то c = a, иначе c = b»
- Блок операторов – образует свою область видимости

```
{ <Оператор 1>; ... <Оператор n>; }
int res = 0;
{ int a = 5; res += 5; }
{ int a = 10; res -= a; } // ошибки переопределения нет, т.к. другая
                          // область видимости
```

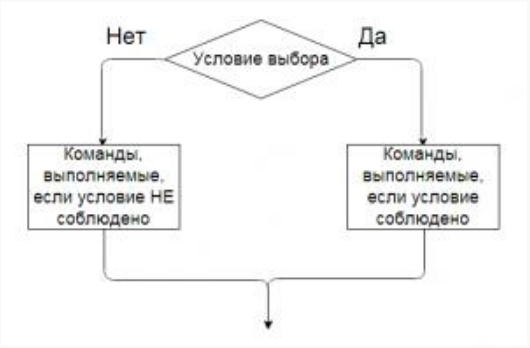
Основные конструкции языка Си (2 / 6)

- Условный оператор (оператор ветвления)

if (<Условное выражение>) <Оператор, если верно;>

[**else** <Оператор, если неверно;>]

- В качестве <оператора, если верно> может использоваться блок операторов, если выражений больше одного




```
int a = 0;
if (a > 10)
    a += 5;
    a += 5;
```

printf("%d\n", a); // выведет **5**, т.к. **отступы** ни на что **не влияют**

- В качестве <оператора, если неверно> может использоваться другой условный оператор
- В качестве условного выражения может использоваться составное выражение

Основные конструкции языка Си (3 / 6)

- Оператор выбора

switch (<выражение>) {

case <возможное значение 1> : <оператор; (или блок)>

case <возможное значение 2> : <оператор; (или блок)>

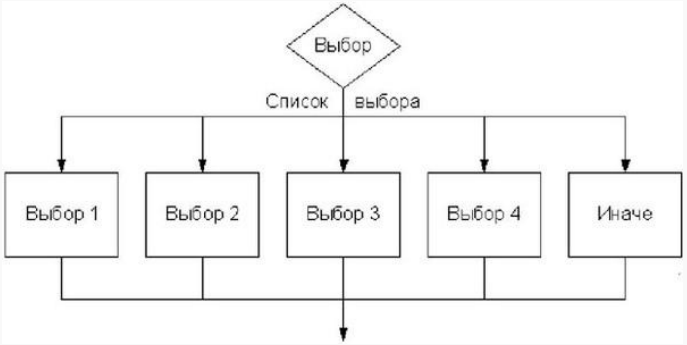
….

[**default** : <оператор; (или блок)>]

}

- После выхода в каком-то case, «проваливается дальше», выполняя все следующие операторы либо до конца switch, либо до нахождения break

```
int n = 0;
switch (n) {
    case 0: printf("n=0 ");
    case 1: printf("n=1 "); break;
    default: printf("n is unknown");
} // Выведет n = 0 n = 1
```



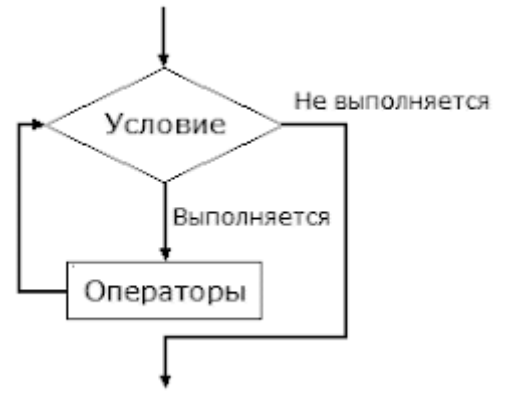
Основные конструкции языка Си (4 / 6)

- Цикл с предусловием

while (<Выражение>) <Оператор (или блок)>

- Если выражение сразу же ложно, то тело цикла не выполнится ни разу
- Часто используется для реализации так называемых поисковых циклов

```
srand(time(NULL)); // инициализация генератора случайных чисел текущим временем
int found = 0;
int secret = rand() % 10; // загадывание случайного числа от 0 до 9
int i = rand() % 10;
int tries = 5; // количество попыток
while (!found && tries > 0) {
    if (i == secret) { found = 1; }
    else { --tries; i = rand() % 10; }
}
printf("%s %d\n", found ? "Отгадал число " : "Не отгадал число", secret);
```



Основные конструкции языка Си (5 / 6)

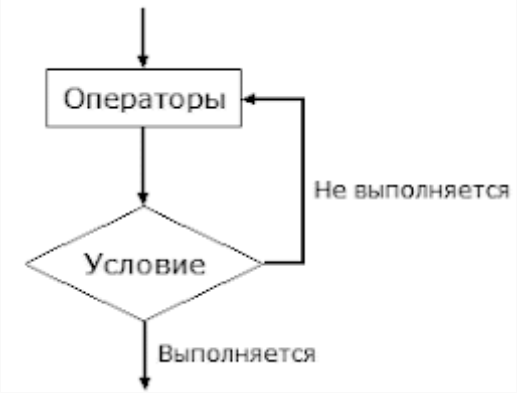
- Цикл с постусловием

do <Оператор (или блок)> **while** (<Выражение>);

- Тело цикла выполнится как минимум один раз;

```
srand(time(NULL)); // инициализация генератора случайных чисел текущим временем
int found = 0;
int secret = rand() % 10; // загадывание случайного числа от 0 до 9
int tries = 5; // количество попыток
do {
    int i = rand() % 10;
    if (i == secret) { found = 1; }
    else { ++i; --tries; }
} while (!found && tries > 0);

printf("%s %d\n", found ? "Отгадал число " : "Не отгадал число", secret);
```



Основные конструкции языка Си (6 / 6)

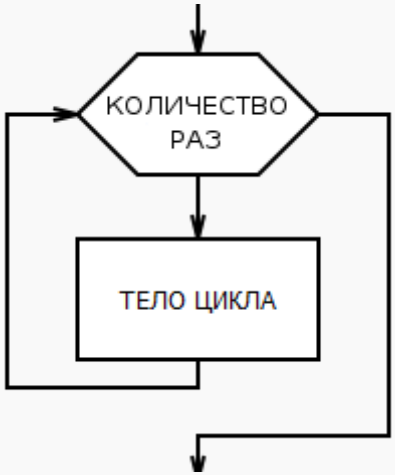
- Счётный (универсальный) цикл

```
for (<Выражение 1>; <Выражение 2>; <Выражение 3>)
    <Оператор>;
```

- Эквивалентно:

```
<Выражение 1>;
while (<Выражение 2>) {
    <Оператор>;
    <Выражение 3>;
}
```

```
int sum = 0;
for (int i = 0 ; i < 100; ++i) {
    sum += i;
}
```

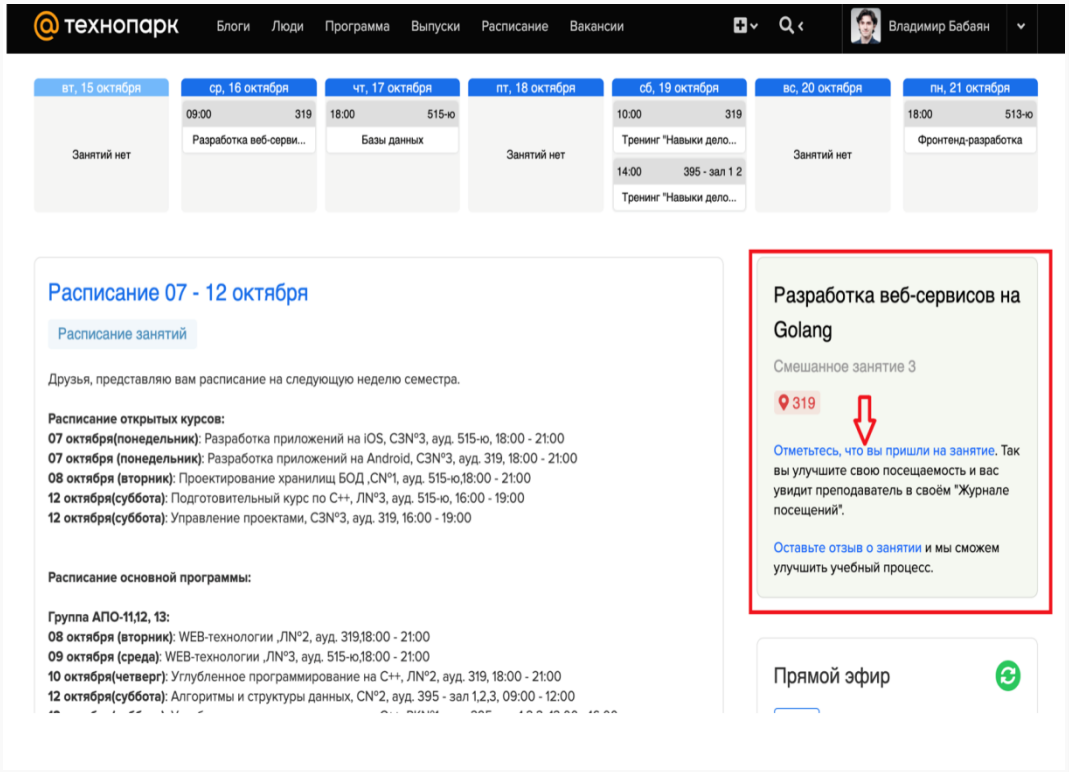



```
srand(time(NULL));
int found = 0;
int secret = rand() % 10;

for  (int i = rand() % 10, tries = 5;
       !found && tries > 0;
       /* ничего не делать */ ) {
   if (i == secret) { found = 1; }
   else { i = rand() % 10; --tries; }
}
printf("%s %d\n", found ?
          "Отгадал число " :
          "Не отгадал число",
          secret);
```

Отметьтесь на портале

- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения



Этапы выполнения программы на языке Си

- **Перед** непосредственным **выполнением** программа проходит несколько стадий обработки:
  - **Обработка препроцессором**
    - обработка программы как текста;
    - выполнение директив программиста для преобразования текста в код, необходимый для компиляции
  - **Компиляция**
    - полноценный **разбор кода** программы (включая **синтаксический** анализ);
    - **извлечение** основных программных **сущностей** (переменные, функции и т.п.);
    - **трансляция** программного кода в язык **машинных команд**.
    - происходит для **каждого** файла **раздельно**
    - на выходе для **каждого файла** с **исходным кодом** образует **объектный файл**
  - **Линковка**
    - **связывание** найденных **сущностей** из **разных объектных** файлов
    - **объединение объектных** файлов в **единый исполняемый файл**

Основы взаимодействия с препроцессором

- Препроцессор анализирует текст программы **до компиляции**
- Для **взаимодействия** с препроцессором существуют **директивы**, начинающиеся с символа #
- Директивы распространяют своё действие от **точки вхождения до конца файла**
- Примеры директив:
  - #include – замечается содержимым файла с исходным кодом, указанным в качестве параметра
  - #define – вводит новые символические константы и макроопределения (обратная директива - #undef)
  - #if, #ifdef, #ifndef, #else, #elif, #endif – позволяют организовывать включение/исключение фрагментов кода в зависимости от условия
  - #error – возбуждение ошибки
  - #pragma – выполнение операций, зависящих от реализации компилятора

Примеры использования директив препроцессора (1 / 2)


```
#define N 100

#ifdef SQUARE
#undef SQUARE
#endif

#define SQUARE(N) N * N
#if 4 * N * N != SQUARE(N + N)
#error Incorrect SQUARE macro
#endif
```

Примеры использования директив препроцессора (2 / 2)


```
#define PRINT_ARR(ARR) do { \
    for (size_t i = 0; i < N; ++i) { \
        printf("%d ", ARR[i]); \
    } \
} while (0);

#pragma mark - Main program
int a[N];
#ifdef DEBUG
PRINT_ARR(a)
#endif

gcc -DDEBUG test.c // Установка DEBUG-флага при компиляции
```

Модели управления памятью и области видимости объектов данных

- Предлагаемые языком C модели управления объектами данных (переменными) закреплены в понятии класса памяти, которое охватывает:
  - **время жизни** — продолжительность хранения объекта в памяти;
  - **область видимости** — части исходного кода программы, из которых можно получить доступ к объекту по идентификатору;
  - **связывание** — части исходного кода, способные обращаться к объекту по его имени.
- Для языка C характерны **три области видимости**:
  - **блок** — фрагмент кода, ограниченный фигурными скобками (напр. составной оператор), либо заголовок функции, либо заголовок оператора for, while, do while и if;
  - **прототип функции**;
  - **файл**.

Классы памяти в языке C

Класс памяти	Время жизни	Область видимости	Тип связывания	Точка определения
Автоматический	Автоматическое	Блок	Отсутствует	В пределах блока, опционально `auto`
Регистровый	Автоматическое	Блок	Отсутствует	В пределах блока, `register`
Статический, без связывания	Статическое	Блок	Отсутствует	В пределах блока, `static`
Статические, с внешним связыванием	Статическое	Файл	Внешнее	Вне функций
Статические, с внутренним связыванием	Статическое	Файл	Внутреннее	Вне функций, `static`

Системные аспекты выделения и освобождения памяти

- Взаимодействие программы с ОС в плане работы с памятью состоит **в выделении и освобождении участков оперативной памяти разной природы и различной длины**, понимание механизмов которого:
  - упрощает создание и использование структур данных произвольной длины;
  - дает возможность избегать «утечек» оперативной памяти;
  - позволяет разрабатывать высокопроизводительный код.
- **В частности, необходимо знать о существовании 4-частной структуры памяти данных:**
  - область данных, сегмент BSS и куча (входят в сегмент данных);
  - программный стек (не входит в сегмент данных).

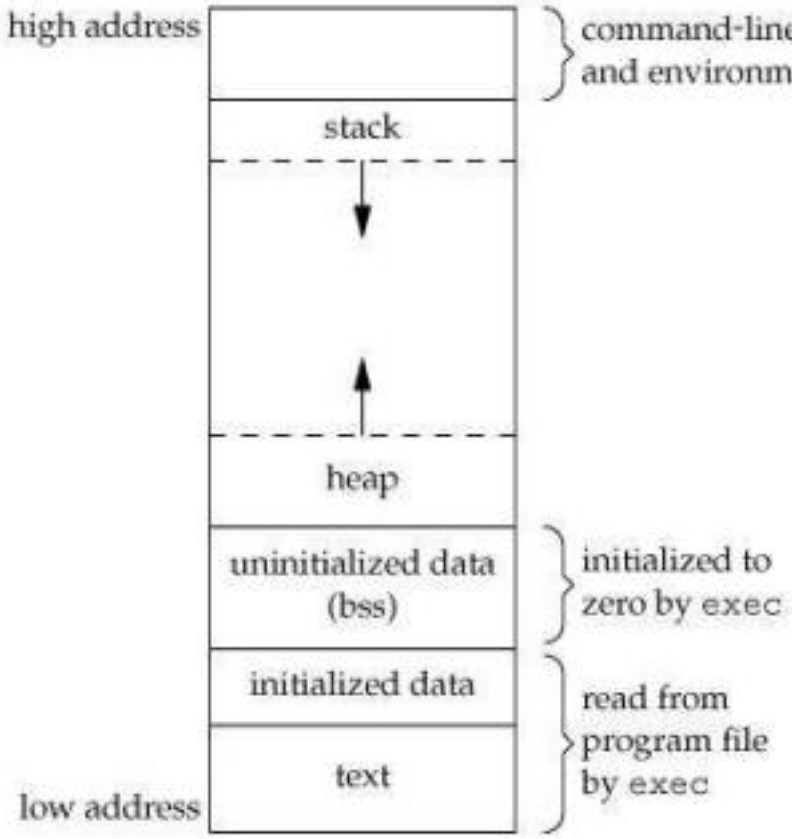
Область данных и сегмент BSS

- **Область данных** (data area) — используется для переменных со статической продолжительностью хранения, которые явно получили значение при инициализации:
  - делится на область констант (read-only area) и область чтения-записи (read-write area);
  - инициализируется при загрузке программы, но до входа в функцию main на основании образа соответствующих переменных в объектном файле.
- **Сегмент BSS** (Block Starting Symbol segment, .bss) — для статических переменных, не получивших значение при инициализации (инициализированы нулевыми битами):
  - располагается «выше» области данных (занимает более старшие адреса);
  - по требованию загрузчика ОС до входа в main() может эффективно заполнить BSS нулями в блочном режиме (zero-fill-on-demand).

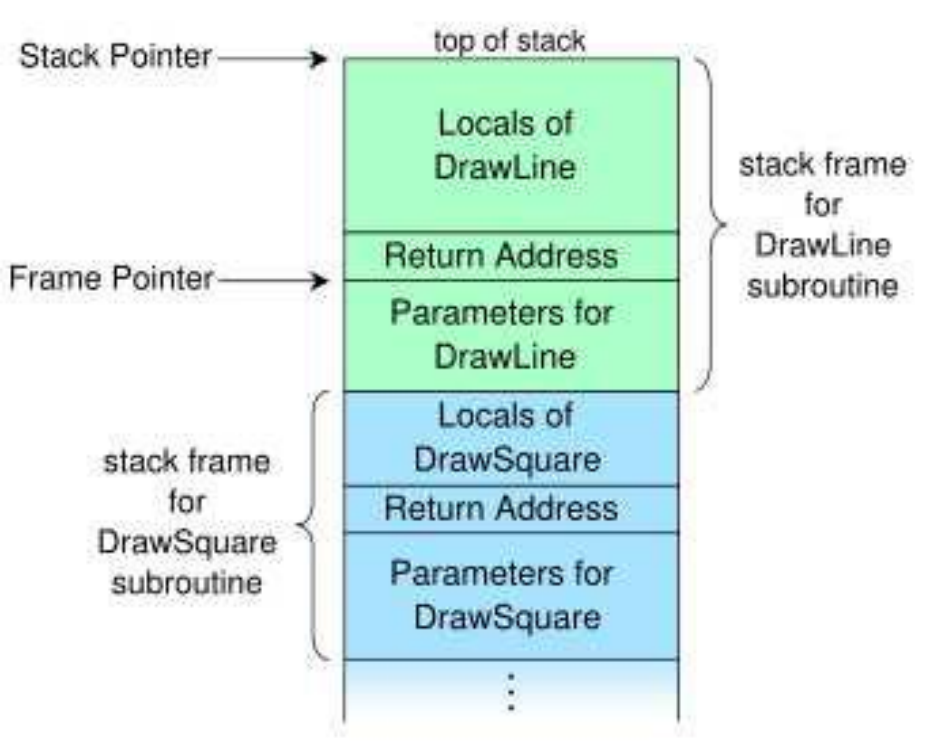
Куча и программный стек

- **Куча** (heap) — контролируется программистом посредством вызова, в том числе, функций malloc / free:
  - располагается «выше» сегмента BSS;
  - является общей для всех разделяемых библиотек и динамически загружаемых объектов (DLO, dynamically loaded objects) процесса.
- **Программный стек** (stack) — содержит значения, передаваемые функции при ее вызове (stack frame), и автоматические переменные:
  - следует дисциплине LIFO;
  - растет «вниз» (противоположно куче);
  - обычно занимает самые верхние (максимальные) адреса виртуального адресного пространства.

Типичная организация памяти программы



Стек (1/3). Кадр стека



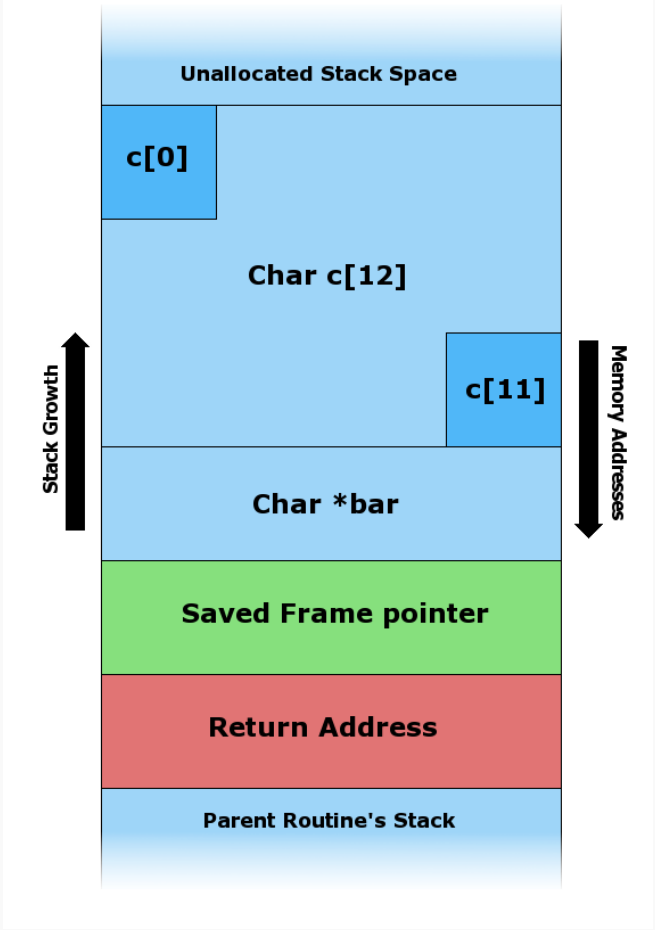
Стек (2/3). Переполнение стека и рекурсия.

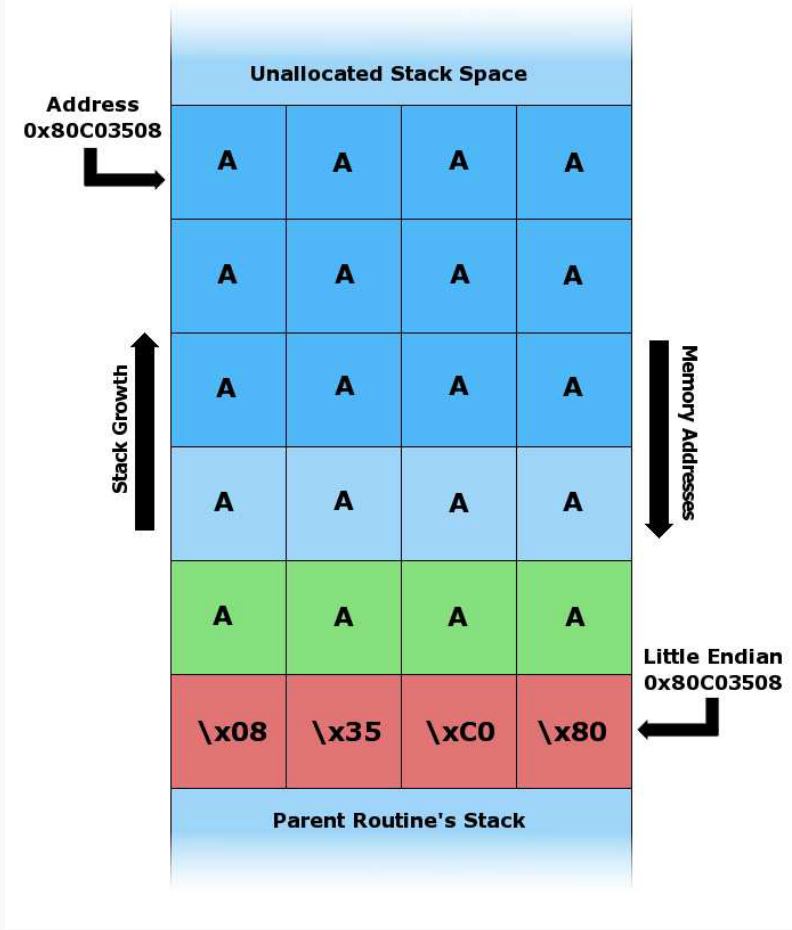

```
int main() {
      main();
}
int add(int n) {
      return n + add(n + 1);
}

// У рекурсии всегда должна быть базовое условие выхода!!!

int main (int argc, char **argv) {
      char c[12];
      strcpy(c, argv[1]); // no bounds checking...
}
```

Стек (3/3). Переполнение стека.





Ещё немного о рекурсии (1 / 2)

Рекурсия – ситуация, когда **одновременно в стеке** существует **несколько фреймов активации одной** и той же **функции**.

Проще говоря – когда **функция так или иначе вызвала внутри** себя **саму же себя**.

Сделать она это могла **напрямую** (**прямая рекурсия**) или **через другие функции** (**косвенная рекурсия**).

У любой рекурсии должно быть **2 составляющие**:

- **рекурсивное правило** – **правило продолжения** рекурсии;
- **базовое условие** – правило **остановки, прекращения** рекурсии

Ещё немного о рекурсии (2 / 2)


```
/* У рекурсии различают прямой и обратный ход */
void foo(int n) {
  if (n == 0) { return; } // базовое условие
  /* сделать что-то ДО рекурсивного вызова - прямой ход */
  foo(n / 2);
  /* сделать что-то ПОСЛЕ рекурсивного вызова - обратный ход */
}

// может быть более одного рекурсивного вызова
int Fibbo(n) {
    if (n <= 0) { return 0; }
    return Fibbo(n - 1) + Fibbo(n - 2); // else не нужен
} // рекурсия - не всегда эффективно!
```

Антишаблоны структурного программирования (1 / 2)

- **«Загадочный» код** (cryptic code) — выбор малоинформативных, часто однобуквенных идентификаторов.
- **«Жесткий» код** (hard code) — запись конфигурационных параметров как строковых, логических и числовых литералов, затрудняющих настройку и сопровождение системы.
- **Спагетти-код** (spaghetti code) — несоблюдение правил выравнивания, расстановки декоративных пробельных символов, а также превышение порога сложности одной процедуры (функции).
- **Магические числа** (magic numbers) — неготовность определять как символические константы все числовые литералы за исключением, может быть, 0, 1 и -1.

Антишаблоны структурного программирования (2 / 2)

- **Применение функций как процедур** (functions as procedures) — неготовность анализировать возвращаемый результат системных и пользовательских функций.
- **«Божественные» функции** (God functions) — функции, берущие на себя ввод данных, вычисления и вывод результатов или иные задачи, каждая из который следует оформить самостоятельно.
- **Неиспользование переносимых типов** — size_t, ptrdiff_t и др.
- **«Утечки» памяти** (memory leaks) и внезапное **завершение процесса** вместо аварийного выхода из функции.
- Использование ветвлений с **условиями, статистически смещенными** не к истинному, а **к ложному результату**.
- **Недостижимый код** (unreachable code).

Использование оператора continue для неструктурной передачи управления как пример антишаблона (1 / 2)


```
// не рекомендуется
// см. SonarQube C Quality Profile [Sonar Way]
// MINOR: ‘continue’ should not be used
int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < 10; i++) {
        if(i == 5) {
            continue;          /* Non-Compliant */
        }
        printf(“i = %d\n”, i);
    }
    return -1;
}
```

Использование оператора continue для неструктурной передачи управления как пример антишаблона (2 / 2)


```
// рекомендуется
// см. SonarQube C Quality Profile [Sonar Way]
// MINOR: ‘continue’ should not be used
int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < 10; i++) {
        if(i != 5) {          /* Compliant */
            printf(“i = %d\n”, i);
        }
    }
    return -1;
}
```

Статическое качество исходного кода на языке C

- Сложившийся на сегодня ландшафт инструментальных средств статического анализа кода (SonarQube и др.) в первую очередь оперирует **правилами** организации и оформления кодовой базы (rules), среди которых выделяются **уязвимости** (vulnerabilities), **неполадки** (bugs) и **видимые проблемы** (code smells).
- Источниками нормы в ходе проверок являются стандарты, среди которых для языка C наиболее известны следующие:
  - **Common Weakness Enumeration** (CWE) — составленный сообществом перечень часто встречающихся «слабых мест» в безопасности ПО;
  - **MISRA C** — стандарт разработки ПО на языке C, созданный и поддерживаемый Motor Industry Software Reliability Association;
  - **SANS TOP 25** — родственный CWE список из 25 самых опасных, распространенных и критических ошибок ПО;
  - **SEI CERT C Coding Standard** — стандарт разработки безопасного кода, предложенный Software Engineering Institute (SEI).

Уроки безопасности: неполадки в коде на языке C

MAJOR [CERT, CWE]: Dead stores SHOULD be removed.

```
// NON-COMPLIANT
int pow(int a, int b) {
    if(b == 0) { return 0; }
    int x = a;
    for (int i = 1; i < b; i++) {
        x = x * a;     // Dead store because the last return
                       // statement should return x instead
                       // of returning a
    }
    return a;
}
```

Уроки безопасности: неполадки в коде на языке C

MAJOR [CERT, CWE]: Dead stores SHOULD be removed.

```
// COMPLIANT
int pow(int a, int b) {
    if(b == 0) { return 0; }
    int x = a;
    for (int i = 1; i < b; i++) {
        x = x * a;
    }
    return x;
}
```

Уроки безопасности: неполадки в коде на языке C

[]: Doubled prefix operators !! and ~~ SHOULD NOT be used. (EXCEPTIONS: Boolean normalization !! is ignored.)

MINOR []: Values of different enum types SHOULD NOT be compared.

```
// NON-COMPLIANT
enum apple {BRAEBURN, FUJI, GRANNY_SMITH, RED_DELICIOUS};
enum orange {BLOOD, NAVEL, BITTER, BERGAMOT, MANDARIN};
bool fun(enum apple v1, enum orange v2) {
    return v1 == v2; // Non-compliant
}
```

Что еще изучить?

- Дополнительную информацию о распространенных **атаках** (attack) на ПО, уязвимостях, **средствах управления рисками** (жарг. «контролях», controls) и пр. можно почерпнуть из материалов The Open Web Application Security Project (OWASP).
  - Известный пример атаки: [Classical] Buffer Overflow
  - Известные уязвимости: Double Free, Memory Leak, Unchecked Return Value: Missing Check Against Null.

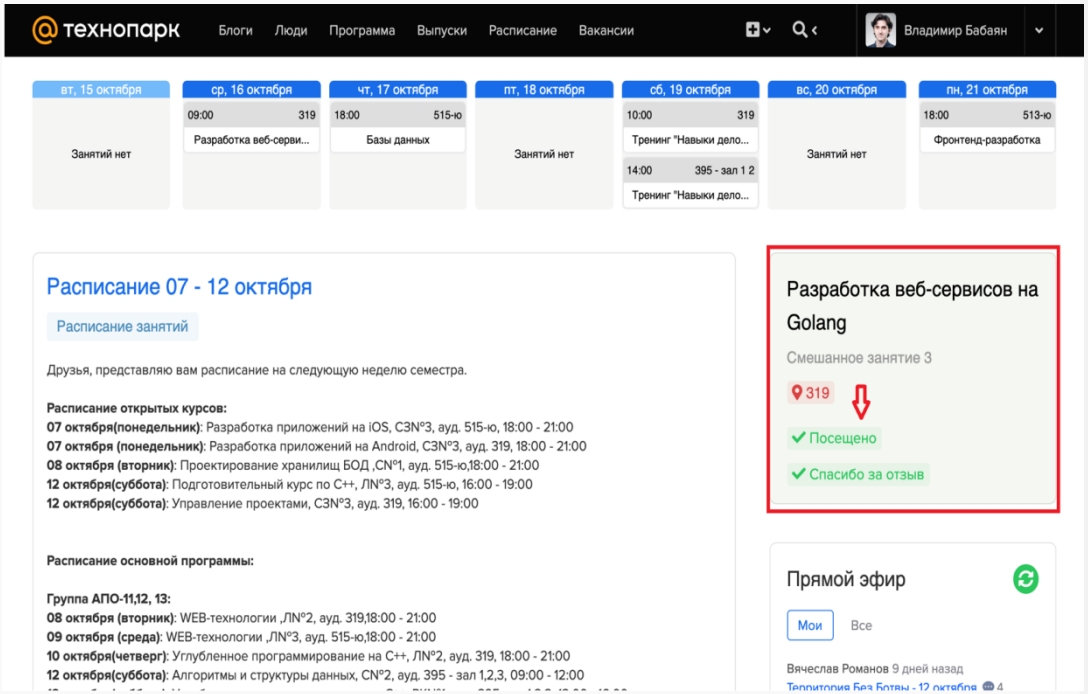
- Авторитетным изданием на тему безопасной разработки ПО на языках C и C++ является также книга Р. Сикорда (Robert C. Seacord) *Secure Coding in C and C++* (2005, 2013).

Домашнее задание №1

- Объединиться в команды для выполнения проекта на языке C++, выбрать ментора и оставить заявку в комментариях к посту на портале
- Настроить окружение для программирования на языке C (желательно, Unix-подобное – с помощью виртуальной машины/Docker/...)
- Ознакомиться с процедурой решения задач, описанной в GitLab https://github.com/leshiy1295/prep-2021-02/tree/workflow
- Решить ДЗ № 1, расположенное в ветке hw-1 в GitLab https://github.com/leshiy1295/prep-2021-02/tree/hw-1
- На семинаре № 1 мы научимся работать с GitHub и git – пока можно делать само задание, скачав архив с кодом
- **Дедлайн**: Лекция №3 (18.10.2021)

Оставьте обратную связь

Обратная связь позволяет нам улучшать курс **сразу**, не дожидаясь его завершения.



Умение оставлять конструктивную обратную связь является одним из **важнейших** при работе в **команде**, поэтому **воспользуйтесь** такой возможностью с целью **дополнительного обучения.**

Это занимает **не более 5 минут** и абсолютно **анонимно**!

Есть вопросы?

Спасибо за внимание!