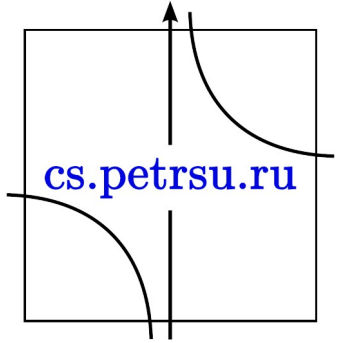


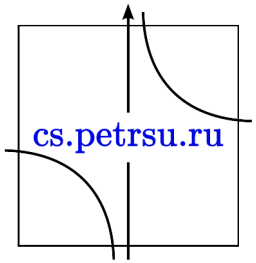
Основы информатики и программирования

Лекция №1

Введение

Структура курса

- Лекции (15 занятий 30 часов)
  - материалы выкладываются каждую неделю
- Практические (15 часов)
  - Разбор решения задач, обсуждения сложных вещей



Структура курса

- Лабораторные (15 занятий 30 часов)
  - практика программирования
  - требования
    - результат соответствует требованиям задачи
    - соблюдены сроки
    - код соответствует установленным стандартам
  - технические задания для каждой задачи сформулированы на странице курса
  - для каждой задачи определена стоимость в баллах
    - за каждую неделю просрочки стоимость уменьшается в два раза
  - успешная защита работы, если разработанная программа соответствует всем установленным требованиям

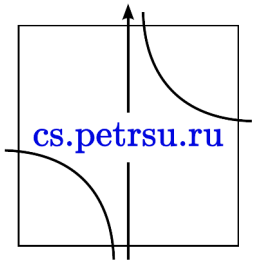
Структура курса

Набор из семи заданий по разработке прототипов программных продуктов

- практические навыки программирования
- способность к организации деятельности

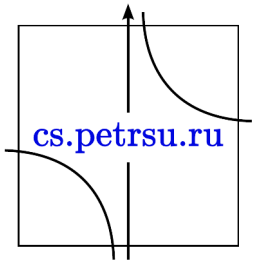
Схема «заказчик-подрядчик-исполнитель»

- результат соответствует требованиям задачи
- соблюдены оговоренные сроки
- код продукта соответствует установлен



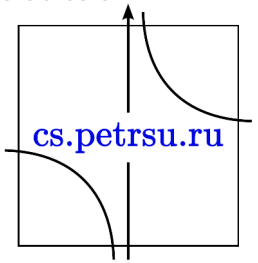
Структура курса

- Самостоятельная работа
  - изучение материалов, документации, чтение дополнительной литературы
  - удаленная работа над заданиями
- Зачет/Экзамен
  - Математика/ПМиИ – зачет
  - ИСиТ, ПрИН - экзамен



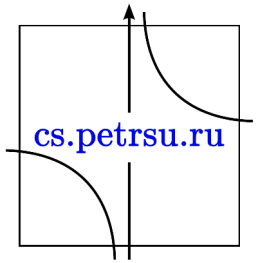
Контроль посещаемости

- Посещение всех форм занятий обязательно
- Пропуски фиксируются преподавателями
  - сведения представляются в деканат
- Документальное обоснование пропусков
  - представлять лектору
  - медсправки, эпикризы, ... — продление сроков
  - командировки, мероприятия — без продления



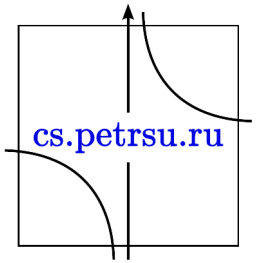
- Чистяков Дмитрий Борисович

- Богоявленская Ольга Юрьевна
- Димитров Вячеслав Михайлович
- Ермаков Владислав Александрович
- Рыбин Егор Ильич



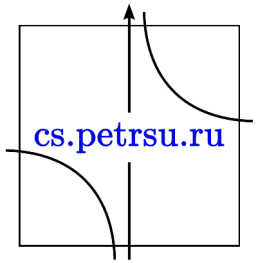
Вычислительная система ИМИТ

- Система GNU/Linux, свободное ПО
- Дисплейные классы
- Терминальный сервер kappa (удаленная работа)
- Домашние каталоги
- Почта, чат, видеоконференции, веб-сервер, gitlab и др.



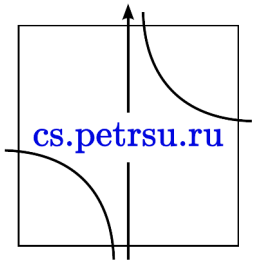
Техническая поддержка

- support@cs.petrsu.ru
- Вадим Анатольевич Пономарев (vadim@cs.petrsu.ru)
- Рыбин Егор Ильич (rybin@cs.petrsu.ru)

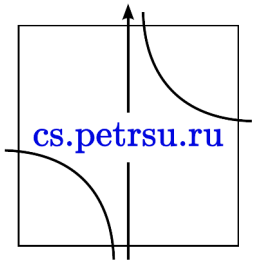


Информатика и программирование

- Информатика — наука о методах и процессах сбора, хранения, обработки, передачи, анализа и оценки информации, обеспечивающих возможность её использования для принятия решений.
- В английском языке: "Computer Science" или "Computing Science" — наука о вычислениях (в общем абстрактном смысле, а не просто о компьютерах).



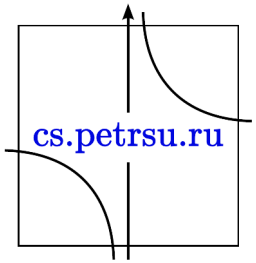
- Программирование — создание компьютерной программы для выполнения определенных вычислений.
- Парадигма программирования — совокупность идей и понятий, определяющих стиль написания компьютерных программ.



Парадигма программирования

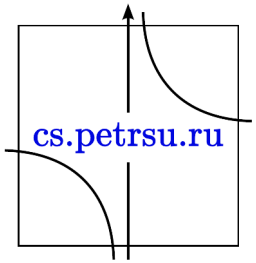
Парадигма определяется:

- вычислительной моделью
- базовой программной единицей(-ами)
- методами разделения абстракций



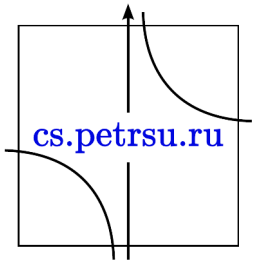
Основные вычислительные модели

- машина Тьюринга (императивное программирование)
- λ-исчисление (функциональное программирование)
- Резолюции над Хорновскими дизъюнктами (логическое программирование)



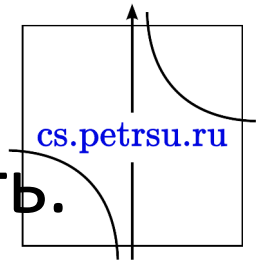
Парадигмы

- Императивное (процедурное) программирование
  - Структурное программирование
  - Объектно-ориентированное программирование
  - Аспектно-ориентированное программирование
- Декларативное программирование
  - Функциональное программирование
  - Логическое программирование
    - Лисп
- Метапрограммирование, ориентированное на языки программирование/предметную область
    - Tex, SQL, Html, Prolog, XML, UML



Императивное программирование

- в исходном коде программы записываются инструкции (команды);
- инструкции должны выполняться последовательно;
- данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями;
- данные, полученные при выполнении инструкции, могут записываться в память.

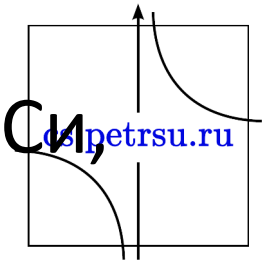


Императивное программирование

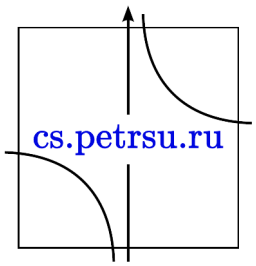
Основные черты:

- использование именованных переменных;
- использование оператора присваивания;
- использование составных выражений;
- использование подпрограмм;
- и др.

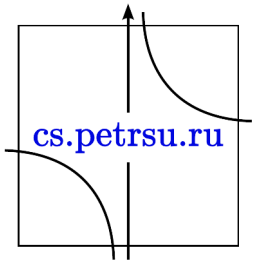
Машинные коды, ассемблер, fortran, basic, Си, C++, Python, Java, Php, Ruby, C#



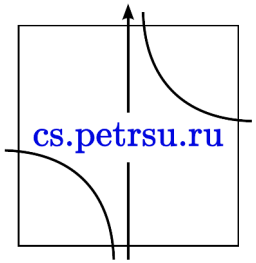
- Изменение состояния программы посредством исполнения инструкций
- Вычислительная модель: машина Тьюринга
- Основные механизмы управления:
  - Последовательное исполнение команд
  - Ветвление
  - Цикл
  - Безусловный переход
  - Вызов подпрограммы
  - Лексический контекст (область видимости)



Мы будем работать в императивной структурной парадигме программирование и использовать язык программирования Си.



- Си (от лат. буквы C) — компилируемый статически типизированный язык программирования общего назначения, разработанный в 1969—1973 годах сотрудником Bell Labs Деннисом Ритчи.



The C Programming Language

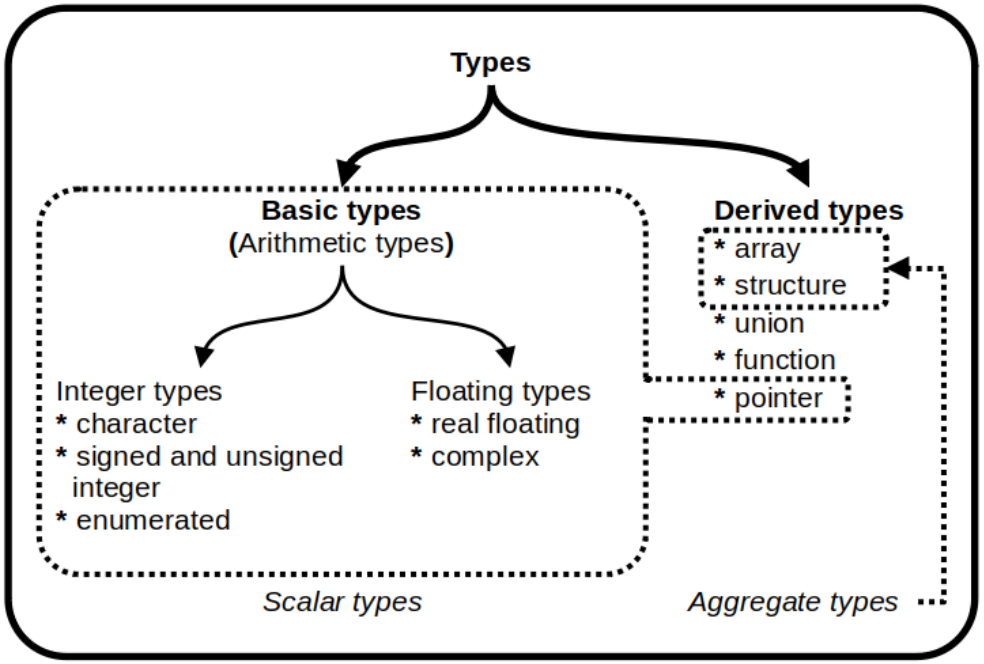
Concepts

Scopes of identifiers
* function
* file
* block
* function prototype

Linkages of identifiers
* external
* internal
* none

Name spaces of identifiers
* label names
* the tags of structures, unions and enumerations
* the members of structures or unions
* ordinary identifiers

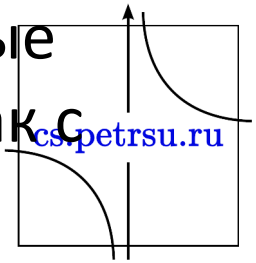
Storage durations of objects
* static
* automatic
* allocated



Язык программирования

— формальный **язык**, предназначенный для записи компьютерных программ.

- элементарные выражения, представляющие минимальные сущности, с которыми язык имеет дело;
- средства комбинирования, с помощью которых из простых объектов составляются сложные;
- средства абстракции, с помощью которых сложные объекты можно называть и обращаться с ними как с единым целым.

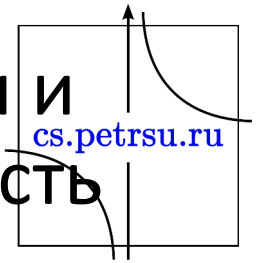


Язык Си

```c
/**
 * main.c -- программа "Hello, students!"
 *
 * Copyright (c) 2022, Mikhail Kryshen <kryshen@cs.petrsu.ru>
 *
 * This code is licensed under MIT license.
 */

#include <stdio.h>

int main()
{
    /* Выводим приветствие */
    fprintf(stdout, "Hello, students!\n");

    return 0;
}
```

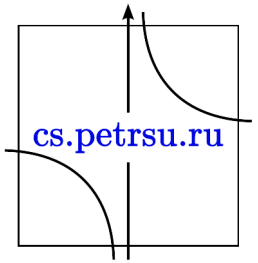
Язык Си

- Небольшой, быстрый в освоении язык (функциональность вынесена в библиотеки)
- Один из первых в истории ЯВУ, сохранивший широкое распространение и в наши дни
  - Компоненты ядра и оболочек современных ОС
  - Реализация сетевых протоколов
  - Платформы и приложения для встроенных систем
- Минимум проверок на этапах трансляции и выполнения, свобода и производительность



Язык Си

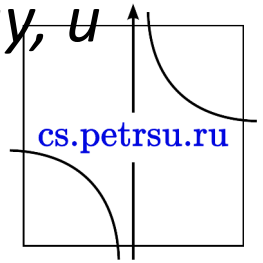
- Большинство «промышленных» технологий разработки наследуют синтаксис и библиотеки функций языка Си
  - C++, D
  - Perl, PHP, JavaScript
  - Java, C# (платформа .NET)
  - …
- Язык стандартизован на международном уровне (ISO/IEC 9899:TC2)



Язык Си

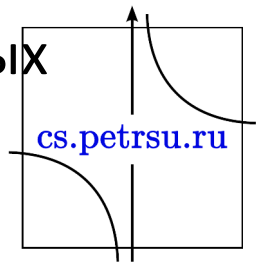
- Ответственность за корректное использование указателей, индексации массивов, списков аргументов переменной длины и возможные утечки памяти возлагается на разработчика
- Автоматические и динамические объекты не инициализируются

«Си — инструмент, острый, как бритва: с его помощью можно создать и элегантную программу, и кровавое месиво». Б. Керниган



Этапы сборки программы

- Препроцессор (gcc -E … / cpp)
  - извлечение и выполнение директив (#...)
- Компиляция (gcc -c … / cc1)
  - генерация машинного (объектного) кода
  - возможно промежуточное построение ассемблерного кода
  - оптимизация
- Компоновка/сборка (gcc / ld)
  - построение исполняемого файла из кода объектных файлов и библиотек



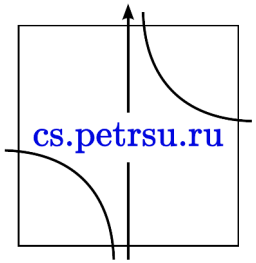
Этапы сборки программы

- Компиляция и редактирование связей вручную:

```
gcc -g -O0 -c -o hello.o hello.c
gcc -g -o hello hello.o
```

- Запуск программы:

```
./hello
```



Получаем объектный код программы

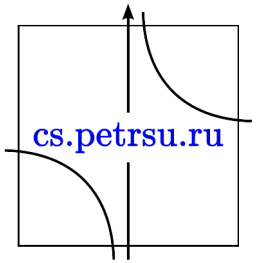
- -g генерирует информацию для отладчика
- -O0 отключает оптимизацию (для отладчика)
- -c получить объектный код и остановиться
- -o hello.o задает путь и имя объектного файла

```
gcc -g -O0 -c -o hello.o hello.c
```

Получаем исполняемый файл

- -g -O0 см. выше
- -o hello задает путь и имя исполняемого файла

```
gcc -g -o hello hello.o
```



Использование make

```
# цель по умолчанию (при вызове make или make task1)
# собираем программу task1 из объектного файла task1.o
task1: main.o
	gcc -g -O0 -o task1 main.o

main.o: main.c
	gcc -g -O0 -c main.c

# цель clean (при вызове make clean)
# удаляем программу и объектные файлы
clean:
	rm task1 *.o
```

Использование make

Набор макроопределений, подстановок и правил сборки

- Макроопределение

  CC = gcc
- Макроподстановка

  $(CC) -g -O0 -c hello.c
- Правило

  hello : hello.o

  $(CC) -g -o hello hello.o

  hello.o : hello.c

  $(CC) -g -O0 -c -o hello.o hello.c

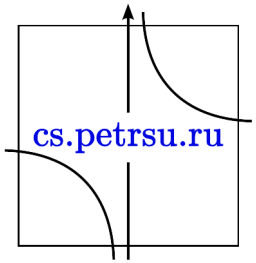
- Макроопределение

  имя = подстановка
- Макроподстановка

  $(имя)
- Правило

  цель : зависимости

  <TAB>действие



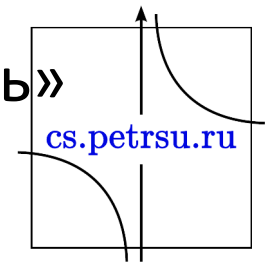
Использование make

Цель — чаще всего файл, который необходимо получить на выходе

- Зависимости — чаще всего файлы, которые необходимы для получения целевого
- Действия — конкретные команды достижения цели

Особенности синтаксиса Makefile

- Действие всегда выделено одним символом табуляции
- Пробелы => ошибка типа «Пропущен разделитель»

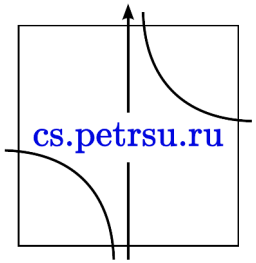


Использование make

Как работает make

- Анализируются зависимости заданной цели (или цели по умолчанию — первой в Makefile)
- Зависимости могут выступать в роли целей в других правилах
- Выполняются только действия тех правил, в которых целевой файл не существует или временная отметка модификации старше отметок зависимостей

Применение make позволяет сократить время повторной сборки!



Анатомия функции

Имени функции предшествует тип возвращаемого значения. Результат main имеет целый тип (int) и передается в ОС как признак успешного или ошибочного завершения

Каждая функция имеет имя, основная функция имеет имя main, имена в Си чувствительны к регистру

Скобки указывают на то, что main является функцией, кроме того, позволяют определить передаваемые функции параметры. В этом примере список параметров пуст

```
int main ()
{
    fprintf (stdout, "Hello, world!\n");
    return 0;
}
```

Тело функции состоит из инструкций, каждая завершается ;

В фигурные скобки заключены инструкции, составляющие тело функции. Выполнение программы начинается с первой инструкции функции main

Ключевое слово return обеспечивает выход из функции и передачу возвращаемого значения

Стандартная библиотека

С символа # (sharp) начинаются директивы препроцессора Си, который обрабатывает текст программы до компилятора. Директива include приводит к включению текста указанного файла в текст программы

Файл заголовков имеет расширение .h и содержит объявления используемых функций. Имя файла в угловых скобках означает его поиск в стандартных каталогах

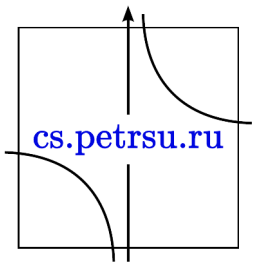
```
#include <stdio.h>

int main ()
{
    fprintf (stdout, "Hello, world!\n");
    return 0;
}
```

Файл stdio.h – один из файлов заголовков стандартной библиотеки

Функция fprintf, обеспечивающая форматный вывод данных, как и многие другие функции не является частью языка. Она доступна в стандартной библиотеке. Для использования функции из библиотеки необходимо подключение заголовочного файла, в котором она определена – stdio.h (сокращение от **st**andar**d** **i**nput **o**utput)

Структура программы

- Тело функции
  - операторы определения и описания переменных
  - операторы-выражения
    - непосредственные константы
    - переменные
    - операции
    - вызовы функций
  - операторы потока управления (структурное программирование)
    - ветвления
    - циклы
    - блочный (составной) оператор



Типы данных и ввод-вывод

```c
#include <stdio.h>

#define PI 3.14159265358979323846

int main()
{
    /* Радиус круга */
    float r;

    /* Запрашиваем у пользователя радиус круга */
    fprintf(stdout, "Введите радиус: ");
    fscanf(stdin, "%f", &r);

    /* Вычисляем и печатаем площадь круга */
    fprintf(stdout, "Площаль круга: %.15f\n", PI * r * r);

    return 0;
}
```

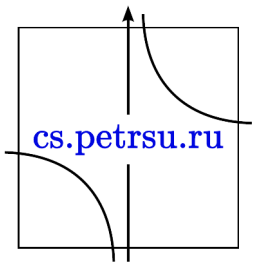
Переменные

Переменные в программе:

- модифицируемые объекты данных
- идентифицируются именем
- должны быть явно отнесены к некоторому типу

Имя переменной:

- начинается с латинской буквы или символа подчеркивания
- содержит латинские буквы, цифры или символ подчеркивания



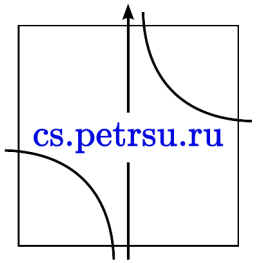
Переменные

Тип данных определяет:

- формат представления значений объекта данных
- диапазон допустимых значений объекта данных
- диапазон допустимых операций

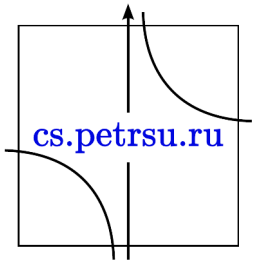
Определение переменной:

- тип имя1, имя2…; тип имя = значение;



Переменные

- Описание переменной вводит имя (идентификатор) переменной и сообщает ее тип: компилятор получает возможность проверить корректность использования.
- Неинициализированная переменная может иметь неопределенное значение.
- Область видимости переменной — от описания до конца блока, в котором описана.



Целочисленные типы	Формат scanf
char	%hhd, %c (как символ)
short	%hd
int	%d
long	%ld

С плавающей точкой	Формат scanf
float	%f
double	%lf

```c
#include <stdio.h>

#define PI 3.14159265358979323846

int main()
{
    /* Радиус круга */
    float r;
    /* Площадь */
    float area;

    /* Запрашиваем у пользователя радиус круга */
    fprintf(stdout, "Введите радиус: ");
    fscanf(stdin, "%f", &r);

    /* Вычисляем и печатаем площадь круга */
    area = PI * r * r;
    fprintf(stdout, "Площаль круга: %.15f\n", area);

    return 0;
}
```

chistyak@kappa:~/c> ./example1

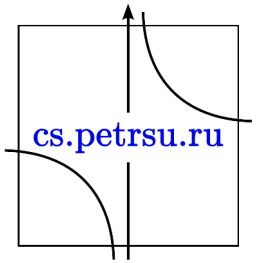
Введите радиус: 1

Площадь круга: 3.141592653589793

chistyak@kappa:~/c> ./ example2

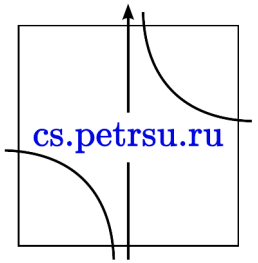
Введите радиус: 1

Площадь круга: **3.141592**741012573



Оператор-выражение

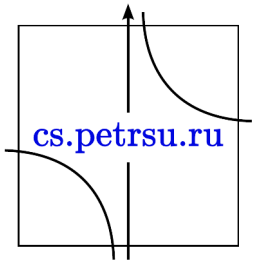
- включает непосредственные константы, переменные, знаки операций и вызовы функций;
- запись оканчивается точкой с запятой;
- после выполнения управление передается следующему за ним оператору (стандартный поток управления).



Оператор-выражение

Непосредственные константы:

- Целочисленные: 101, 0777, 0xFF
- Символьные: 'A', '\n', '\777', '\xFF'
- С плавающей точкой: 1.0, 5.2E-2
- Строковые: "Hello"



Оператор-выражение

Операции:

- Арифметика:

  `+ - * / %`

- Отношения:

  `< > <= >= == !=`

- Логические связки:

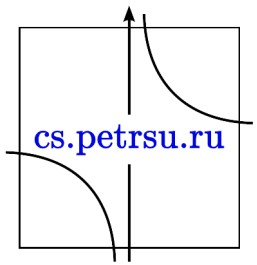
  `&& || !`

- Присваивания:

  `= += -= *= /= %=`

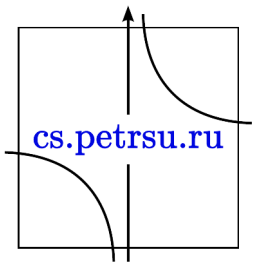
- Скобки:

  `( )`



В порядке приоритета:

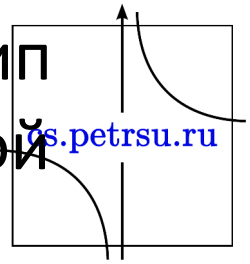
1. !
2. * / %
3. + -
4. < > <= >=
5. == !=
6. &&
7. ||
8. = += -= *= /= %=



Присваивание:

L-value = expression;

- L-value — «леводопустимое выражение»: определяет модифицируемый объект (доступную для записи область памяти), например, имя переменной.
- Expression — «праводопустимое выражение»: допустимое синтаксисом языка выражение, тип которого совпадает с типом выражения в левой части или приводим к нему.



```c
#include <stdio.h>

int main()
{
    int a = 10; /* Допустимая инициализация */
    int b = 20; /* Допустимая инициализация */

    /* Допустимые присваивания */
    a = 30;
    a = 2 * b + 1;
    a = a + 1;

    /* Недопустимые присваивания */
    1 = a;
    a + b = 40;

    return 0;
}
```

Операторы потока управления

- if

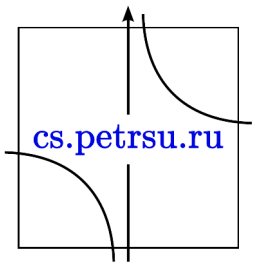
```
/* Условие — выражение любого арифметического типа.
   Ненулевое значение — истина. */
if (условие) инструкция1 else инструкция2
```

```
/* Одиночный оператор завершается точкой с запятой,
   не разрывая if */
if (x > 0)
      y = x;
else
      y = -x;
```

Оператор ветвления if

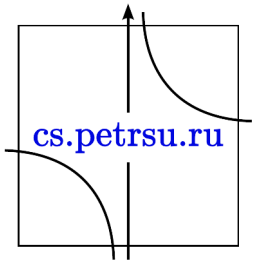
```
/* Неполный оператор */
if (x > 0)
        y = x;
```

```
/* Использование операторных скобок */
if (x > 0) {
        y = x;
        z = y;
} else {
        y = -x;
        z = 0;
}
```



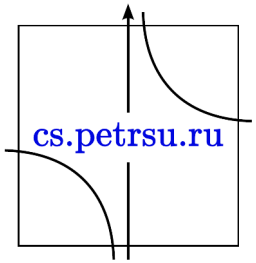
Оператор ветвления if

```
/* Каскад из нескольких if-else... */
if (a == 1) {
      printf("один\n");
} else {
      if (a == 2) {
            printf("два\n");
      } else {
            if (a == 3) {
                  printf("три\n");
            } else {
                  printf("много\n");
            }
      }
}
```



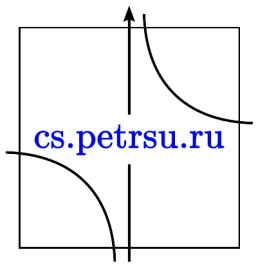
Оператор ветвления if

```
/* ...можно записать красивее, если убрать
   некоторые скобки... */
if (a == 1) {
      printf("один\n");
} else
      if (a == 2) {
            printf("два\n");
      } else
            if (a == 3) {
                  printf("три\n");
            } else {
                  printf("много\n");
            }
```



Оператор ветвления if

```
/* ...и некоторые переносы строк */
if (a == 1) {
      printf("один\n");
} else if (a == 2) {
      printf("два\n");
} else if (a == 3) {
      printf("три\n");
} else {
      printf("много\n");
}
```

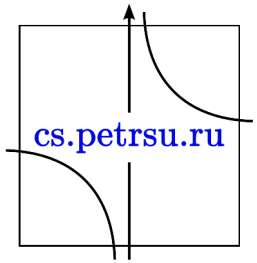


Оператор ветвления if

```
/* Присваивание допускается в условиях */
if ((c = getchar()) != EOF) {
      fprintf(stdout, "Получен символ %c\n", c);
} else {
      fprintf(stderr, "Признак конца файла!\n");
}
```

Поэтому корректно выражение:

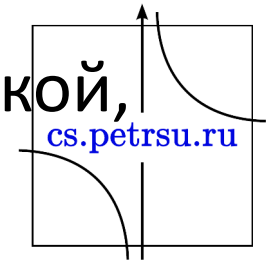
```
/* Всегда истинно, надо было: x == 1 */
if (x = 1) ...
```



Оператор switch/case

- Результат вычисления выражения последовательно сравнивается с константами в case, при совпадении выполнение начинается с инструкции после двоеточия (до конца switch или break).

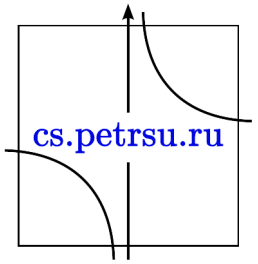
- В case допускаются константные выражения простых (скалярных) целочисленных типов: char, int, …
- В case не допускаются числа с плавающей точкой, составные объекты: массивы, строки



Оператор switch/case

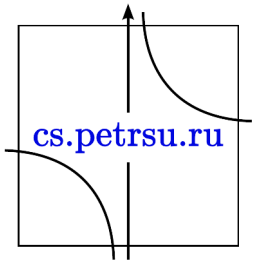
```
switch (выражение) {
case конст1 : инструкции
case конст2 : инструкции ...
default : инструкции
}
```

```
/* При x = 1 будут выведены строки 1 и 2. */
switch (x) {
case 1:
      fprintf(stdout, "1\n");
case 2:
      fprintf(stdout, "2\n");
      break;
case 3:
      fprintf(stdout, "3\n");
}
```



Оператор switch/case

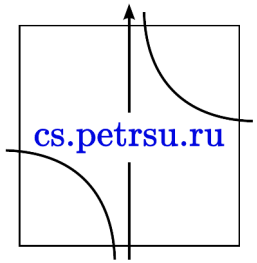
```
/* Можно группировать варианты. */
switch (x) {
case 1:
case 2:
      fprintf(stdout, "1+2\n");
      break;
case 3:
      fprintf(stdout, "3\n");
      break; /* не обязательно, но лучше оставить */
}
```



Оператор цикла while

```
while (условие) тело
```

```
do тело while (условие)
```

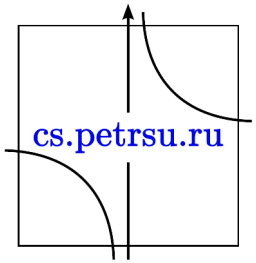


Оператор цикла while

```
/* Тело цикла — одна инструкция или блок */
while (x > 0)
      x = x – 10;

/* Или пустое */
while ((c = getchar()) != '\n');

/* Бесконечный цикл */
while (1) { ... }

/* Вариант с постусловием: минимум одна итерация */
k = 0;
do
      k++;
while ((n = n / 10) != 0);
```

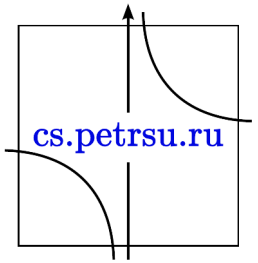
Цикл for

```
for (инициализация; выражение-условие; поствыражения)
```



Цикл for

```
s = 0;
i = 1;
while (i <= n) {
        s = s + i;
        i++;
}

/* Эквивалентно */
s = 0;
for (i = 1; i <= n; i++)
        s += i;

/* То же, но вариант выше лучше читается */
for (s = 0, i = 1; i <= n; s += i, i++);
```



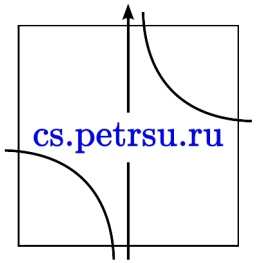
Операторы break и continue

```
x = 0;
while (x < 10) {
      fprintf(stdout, "%d", x);
      x = x + 1;
      if (x == 3) break;
      fprintf(stdout, "!");
}
```

- 0!1!2

```
x = 0;
while (x < 10) {
      fprintf(stdout, "%d", x);
      x = x + 1;
      if (x >= 3) continue;
      fprintf(stdout, "!");
}
```

- 0!1!23456789

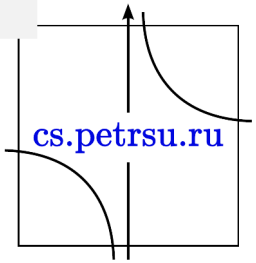


Тернарный оператор

```
условие ? выражение1 : выражение2
```

В отличие от if при вычислении получает определенное значение и может использоваться внутри других выражений.

```
int abs_a_minus_b = a > b ? a - b : b - a;
```

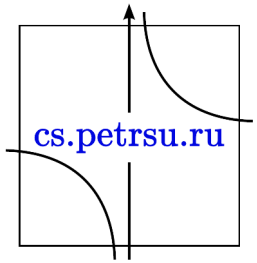


Функции

Подпрограмма (процедура, функция) — оформленный для целей повторного обращения именованный фрагмент кода программы.

– базовый механизм абстракции
– средство декомпозиции программы
– избавление от повторов кода

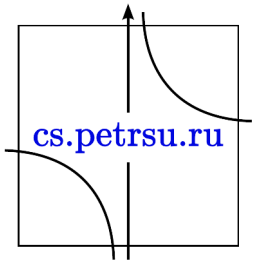
```
тип_возврата имя_функции(арг...)
{
        тело
}
```



- void в качестве типа возвращаемого значения — функция ничего не возвращает,
- void вместо списка аргументов — функция не принимает аргументы.

Дополнительные возможности:

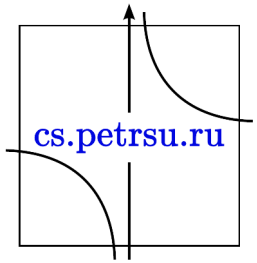
- вложенные определения (функция внутри функции)
- функции с переменным числом аргументов (stdarg.h)



```c
/* Включает прототипы функций scanf и printf */
#include <stdio.h>

/* Прототип функции: функция должна быть объявлена до ее вызова. */
int max(int a, int b);

int main(void)
{
        int k, l, m;
        scanf("%d%d", &k, &l);
        /* Вызов функции: в качестве аргументов передаются текущие
           значения k и l, m будет присвоено возвращенное значение. */
        m = max(k, l);
        printf("max(%d, %d) = %d\n", k, l, m);
        return 0;
}

int max(int a, int b)
{
        if (a > b)
                /* Завершить выполнение функции, вернув значение a. */
                return a;
        /* Это выполнится, только если не выполнено условие выше. */
        return b;
}
```

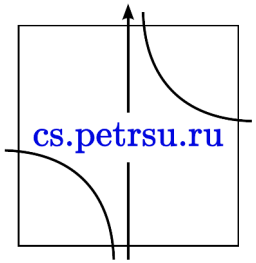
- Следующие варианты реализации max() эквивалентны предыдущему:

```
int max(int a, int b)
{
        if (a > b)
                return a;
        else
                return b;
}
```



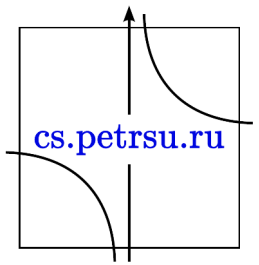
- Следующие варианты реализации max() эквивалентны предыдущему:

```
int max(int a, int b)
{
        /* Локальная переменная,
           не связана с `k` в main(). */
        int k;
        if (a > b)
                k = a;
        else
                k = b;

        return k;
}
```



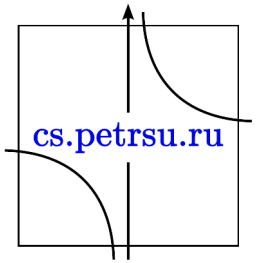
- Следующие варианты реализации max() эквивалентны предыдущему:

```
int max(int a, int b)
{
	return a > b ? a : b;
}
```



- Следующие варианты реализации max() эквивалентны предыдущему:

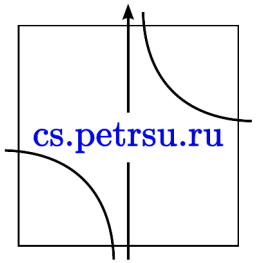
```
int max(int a, int b)
{
        if (a > b) {
                /* Присваиваем локальному 'b',
                   значение 'l' в вызывающей main()
                   не изменится. */
                b = a;
        }

        return b;
}
```



Рекурсия

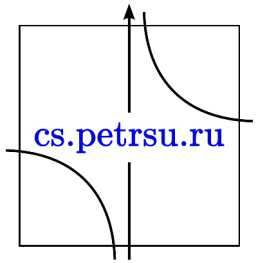
Функция вызывает себя:

```
/* Возвращает факториал числа n. */
int factorial(int n)
{
        /* база рекурсии */
        if (n == 0) {
                return 1;
        }

        return n * factorial(n - 1);
}
```



Рекурсия

= factorial(5) =
= 5 * factorial(4) =
= 5 * 4 * factorial(3) =
…
= 5 * 4 * 3 * 2 * 1 * factorial(0) =
= 5 * 4 * 3 * 2 * 1 * 1 =
= 120

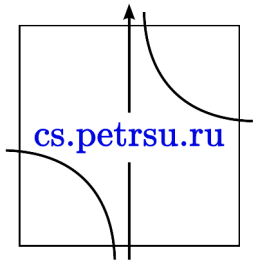


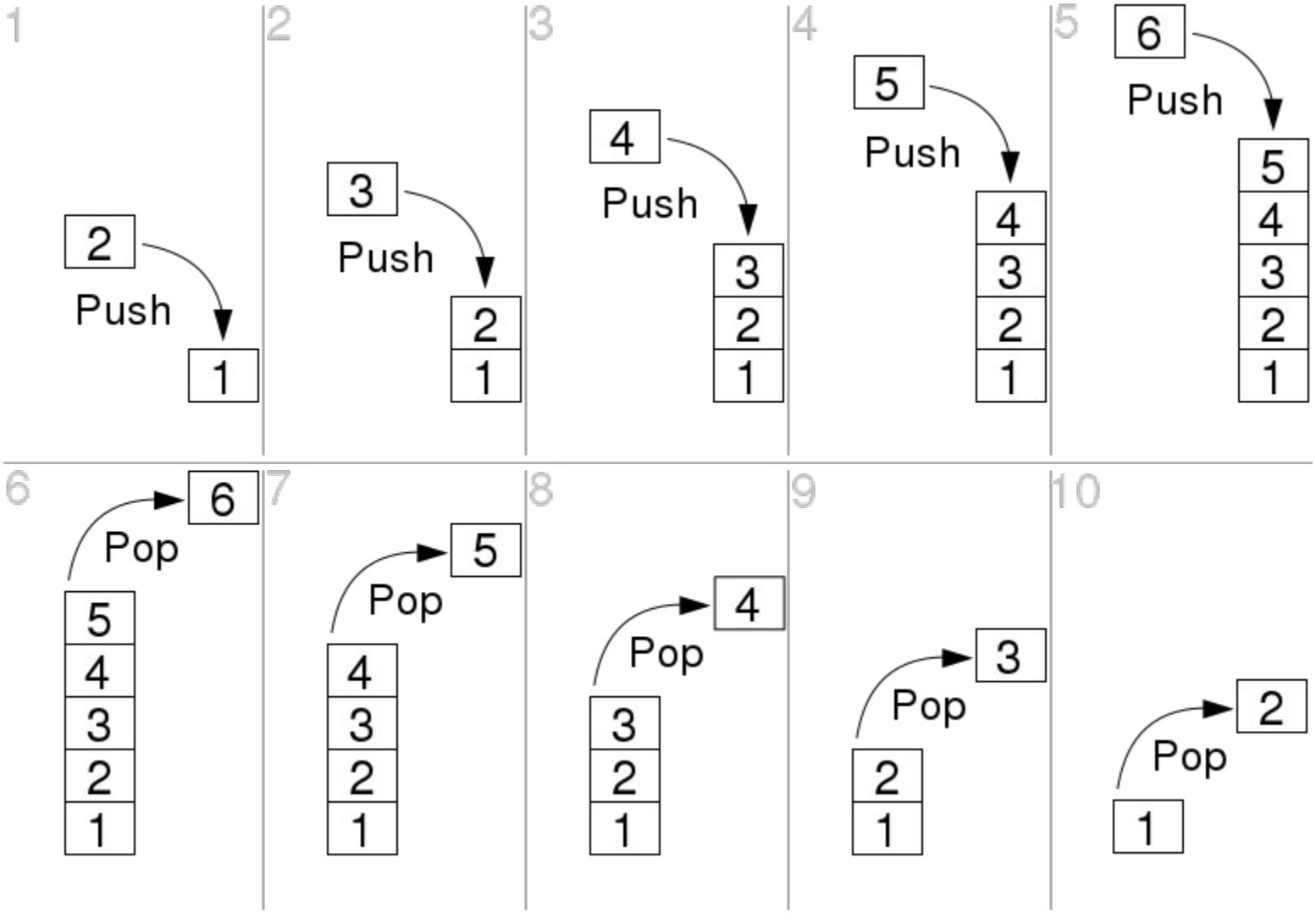
Стек вызовов

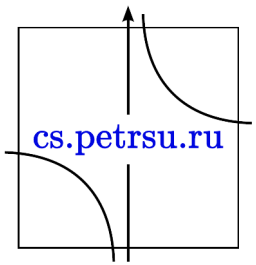
Стек — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (last in — first out, последним пришел — первым вышел).

Операции со стеком:

- push: добавить элемент,
- pop: извлечь последний (находящийся на *вершине* стека) элемент.




1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1


cs.petrsu.ru

- *Аппаратный стек* поддерживается центральным процессором и используется для хранения переменных и вызова подпрограмм.
- *Стек вызовов* (call stack) — стек, хранящий информацию для возврата управления из подпрограмм (функций) в программу (или вызывающую функцию) при вложенных или рекурсивных вызовах.
- *Кадр стека* выделяется при вызове функции и содержит связанное с ним состояние — адрес возврата, аргументы и локальные переменные. Структура кадра зависит от архитектуры процессора и соглашений системы (ABI/ApplicationBinaryInterface).

Переполнение стека приводит к аварийному завершению программы.