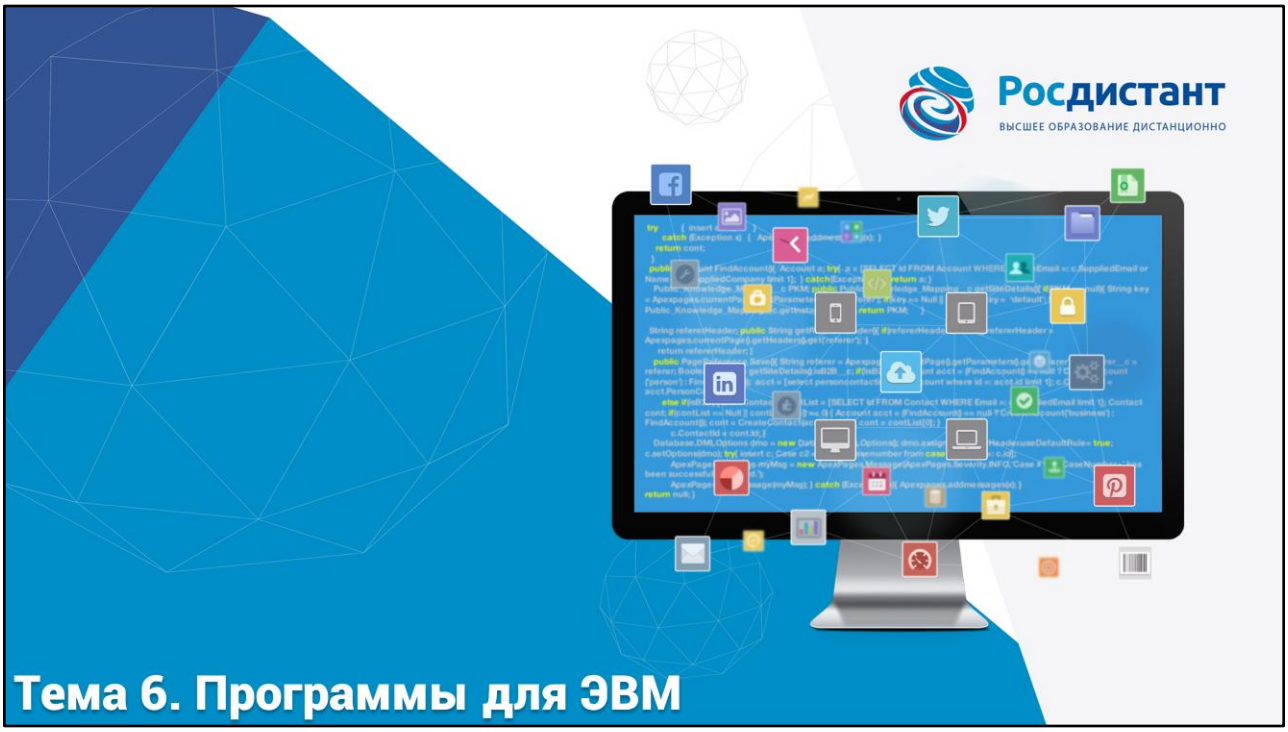

Росдистант
ВЫСШЕЕ ОБРАЗОВАНИЕ ДИСТАНЦИОННО
МЕТОДЫ РЕШЕНИЯ ПРОБЛЕМ В
ИНФОРМАТИКЕ
6 ЧАСТЬ



Тема 6. Программа для ЭВМ

Выше мы говорили, как можно выполнить программу, написанную на языке программирования высокого уровня. Мы изучили два способа выполнения:

- запуск скомпилированной программы;
- выполнение программы в режиме интерпретации.

Мы также отмечали, что ряд интерпретируемых языков программирования, таких как Java и C#, предполагают предварительную компиляцию программы в байт-код, который затем выполняется на соответствующей виртуальной машине.

Однако возникают вопросы: что действительно представляет из себя программа, способная выполняться на компьютере? Какие ресурсы выделяет компьютер для выполнения программы?

В этой теме мы рассмотрим:

- что такое указатели;
- зачем нужны типы данных;
- распределение памяти во время выполнения программы;
- способы передачи параметров.

АРХИТЕКТУРА ФОН НЕЙМАНА

- Основной принцип: данные и инструкции совместно хранятся в одной памяти
- Следствие: и программа и данные должны загружаться в память компьютера
- Данные и команды хранятся в двоичном формате

Росдистант

Архитектура фон Неймана

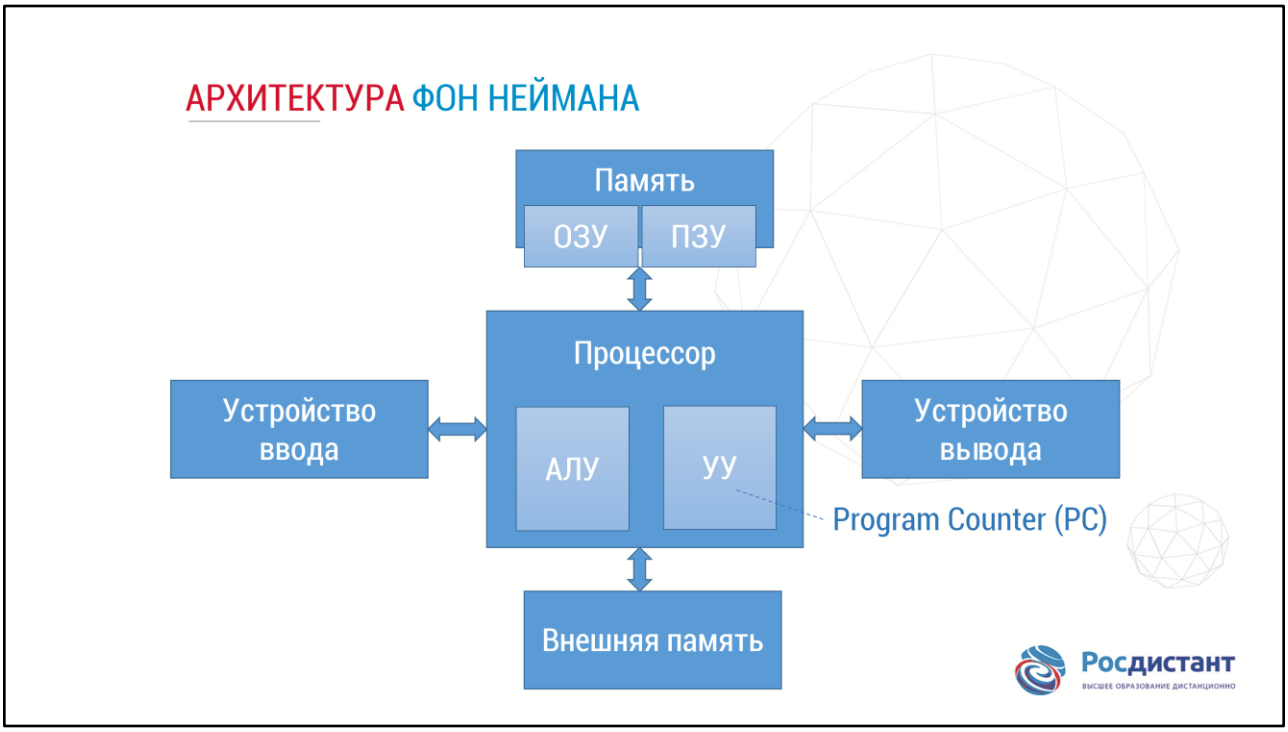
Ранее мы рассмотрели основные парадигмы программирования. Детальное изучение данного вопроса приводит нас к понятию вычислительной модели – некоторой организации вычислительной системы, способной выполнять программы, написанной на языке программирования, поддерживающего ту или иную парадигму. Действительно, существует ряд вычислительных моделей, но, в большинстве случаев, они основываются на некоторой базисной.

Принято считать, что американский математик Иоганн фон Нейман разработал общую архитектуру компьютера, на основании которой идет разработка вычислительной техники и по настоящее время.

Основой архитектуры фон Неймана является принцип совместного хранения в одной памяти данных и инструкций по их обработке.

Иными словами, чтобы программа могла обработать входные данные и получить выходные, необходимо, чтобы в память компьютера была загружена и программа, и данные. Причем компьютер фон Неймана не отличает память, в которой хранятся данные, и память, в которой хранятся инструкции – коды команд процессора.

Данные и команды хранятся в двоичном формате, то есть состоят из нолей и единиц.



Архитектура фон Неймана

Архитектура фон Неймана предполагает, что процессор состоит из двух частей:

- устройства управления – УУ;
- арифметико-логического устройства – АЛУ.

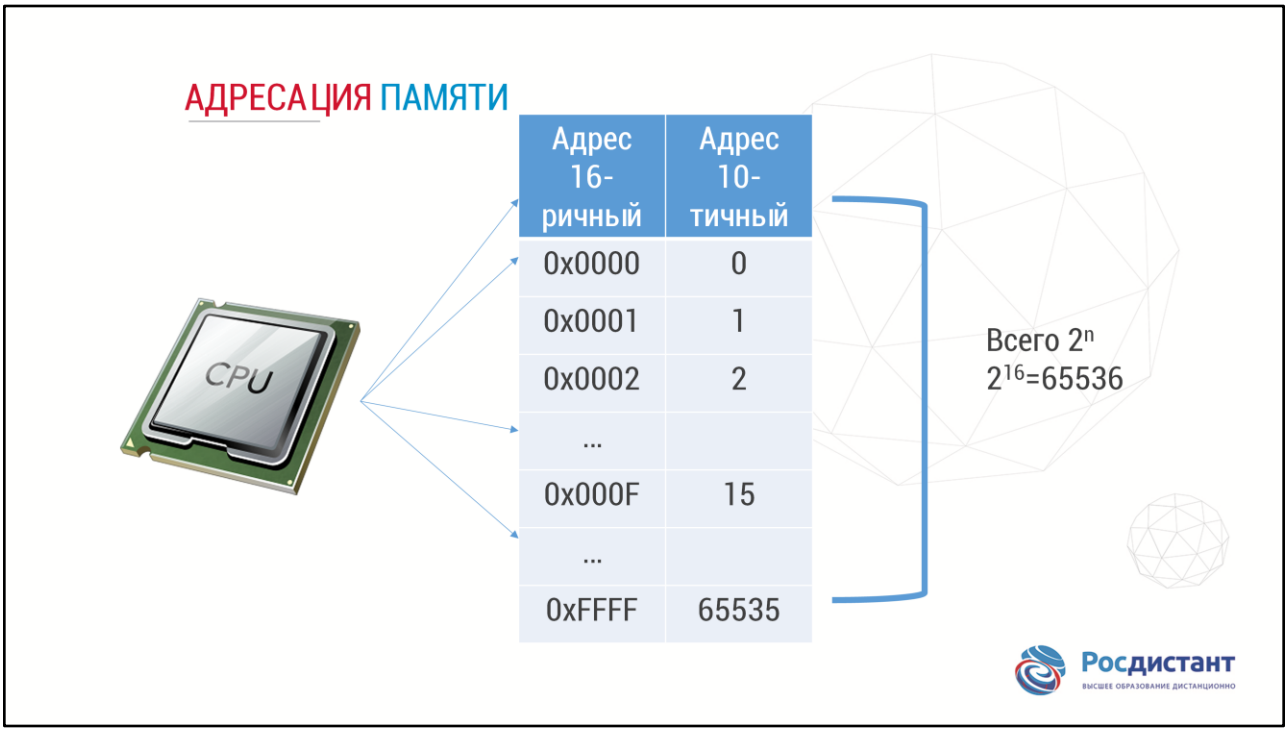
Память, непосредственно доступная процессору, состоит из оперативного и постоянного запоминающих устройств – ОЗУ и ПЗУ.

Программа и данные, с которыми взаимодействует процессор, загружаются в оперативную память. В процессоре есть особый регистр, который реализует программный счетчик. Программный счетчик выбирает следующую инструкцию для выполнения. В процессоре есть и другие регистры, которые хранят данные или ссылки на данные, необходимые для выполнения текущей инструкции.

Связь с внешней памятью, например с жестким диском, и устройствами ввода-вывода организуется через так называемую системную шину.

Как только программы стали более сложными, выявился один из недостатков архитектуры фон Неймана: программисту приходилось писать очень много системных команд, чтобы организовать сложный вычислительный процесс с большим объемом данных.

Стало ясно, что без дополнительных структур организации памяти разработку больших программ вести невозможно. Программистам нужны были инструменты, которые позволили бы им таким образом организовывать свой код, чтобы сосредоточиться на решении поставленных задач, а не на деталях оборудования.



Адресация памяти

Итак, необходимость работы с данными, которые находятся в регистрах, процессорах или оперативной памяти, привела к появлению указателей, переменных и типов данных.

Чтобы процессор мог понять, где находятся данные, которые надо загрузить в регистр, ему надо указать, по какому адресу они находятся. Так появились указатели.

Ссылаясь на ту или иную ячейку памяти, необходимо было понимать две вещи:

- первое – сколько надо взять бит, то есть нолей и единиц, начиная с указанного адреса, чтобы прочитать все данные;
- второе – как интерпретировать выбранную последовательность нолей и единиц.

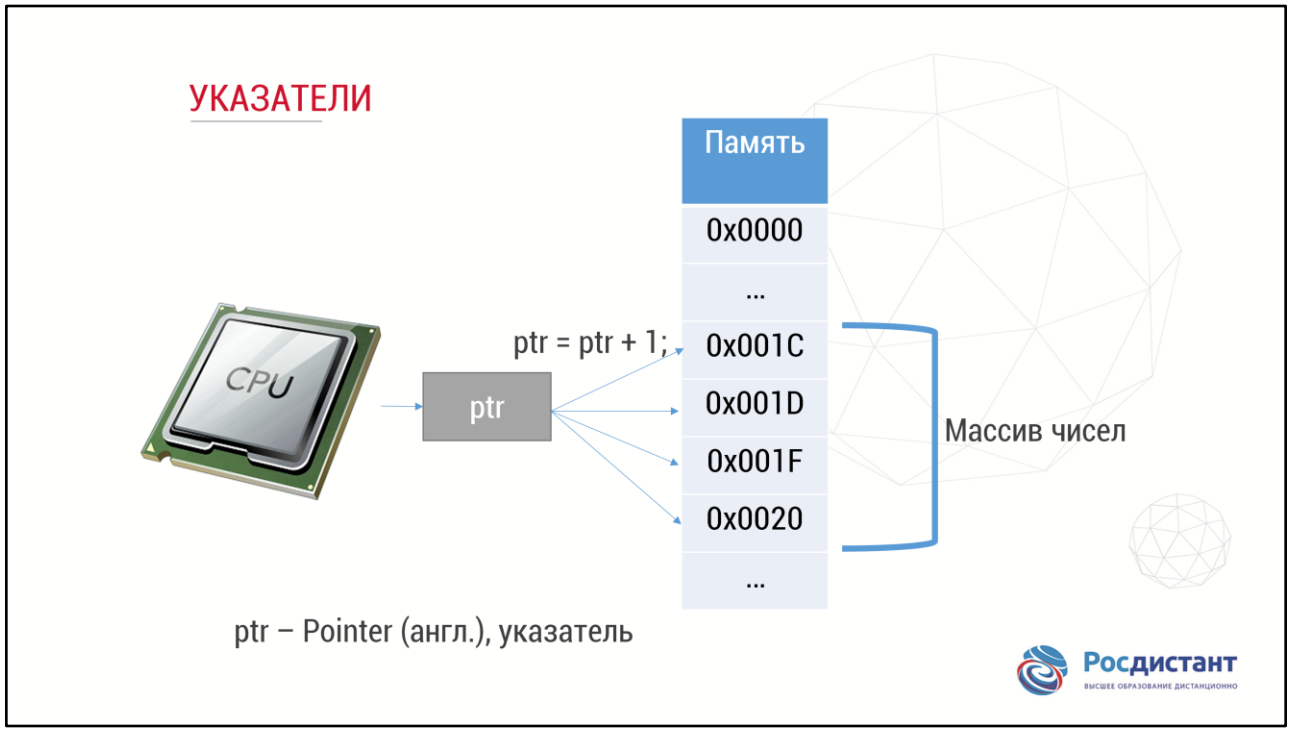
Рассмотрим пример. Будем считать, что мы имеем некоторый условный компьютер. Данные в памяти хранятся в словах, размер каждого слова – 16 бит. Таким образом, наш компьютер шестнадцатиразрядный. Адрес мы будем записывать в шестнадцатеричном формате. Для записи адреса нам потребуется четыре шестнадцатеричных цифры. Чтобы понимать, что число записано в шестнадцатеричном формате, мы будем использовать префикс «0x», как принято на языке Си.

Например:

- 0x0000 – ссылка на первую ячейку памяти;
- 0x0001 – ссылка на вторую ячейку памяти;

- 0x000F – ссылка на 16-ю ячейку памяти;
- 0xFFFF – ссылка на последнюю ячейку памяти, имеющую номер 65 535.

Таким образом, 16-разрядный процессор может адресоваться на $2^{16} = 65\,536$ ячеек с номерами от ноля до $2^{16} - 1 = 65\,535$.



Указатели

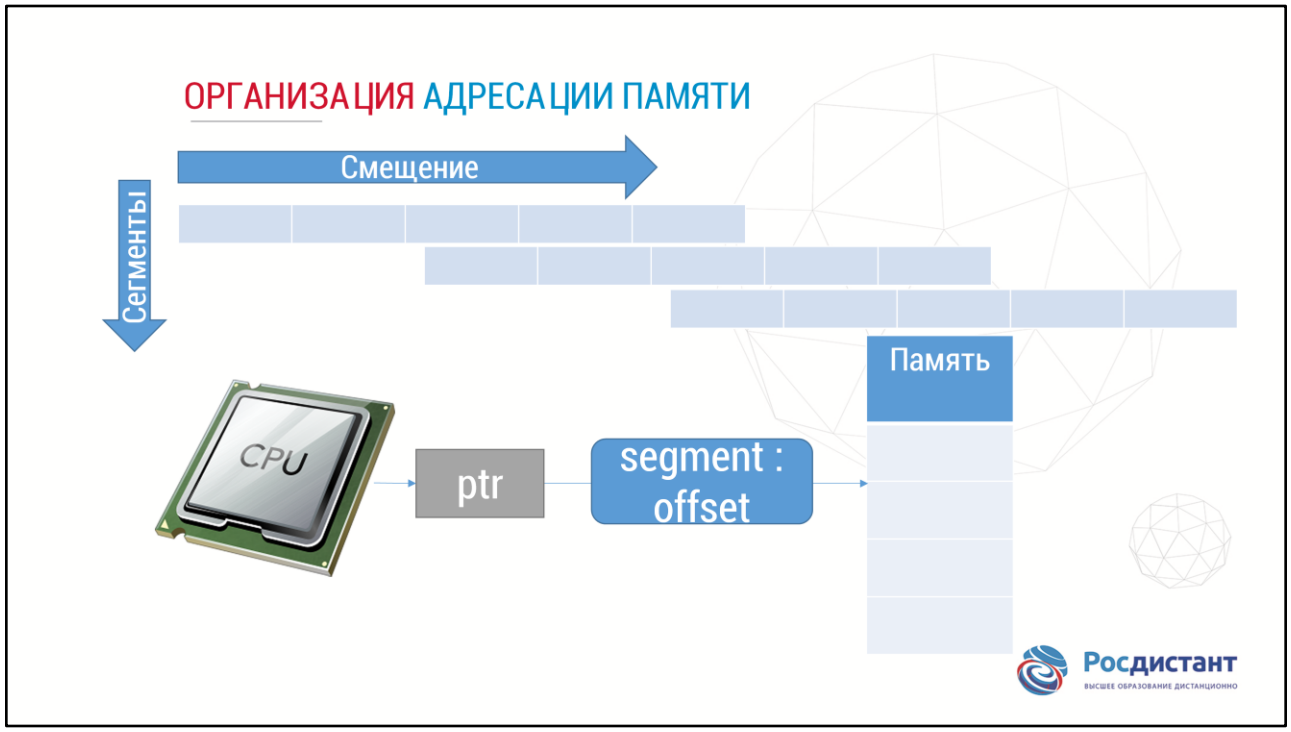
Указатель в программировании – это адрес, по которому процессор будет искать данные.

Например, указатель со значением 0x0010 показывает на ячейку с номером 16, семнадцатую в общей памяти.

Мы можем выделить какую-то ячейку памяти под переменную-указатель и в этой ячейке хранить адрес обрабатываемых процессором данных. На слайде – переменная ptr, от английского Pointer – указатель.

Если у нас в памяти хранится массив 16-разрядных целых чисел, которые в памяти хранятся последовательно, то ptr будет показывать на текущий элемент массива. А переход к следующему элементу будет означать, что к значению ptr мы должны добавить единицу, то есть сослаться на следующий элемент.

Таким образом, проходя по массиву, мы сможем посчитать сумму его элементов или иной показатель.



Организация адресации памяти

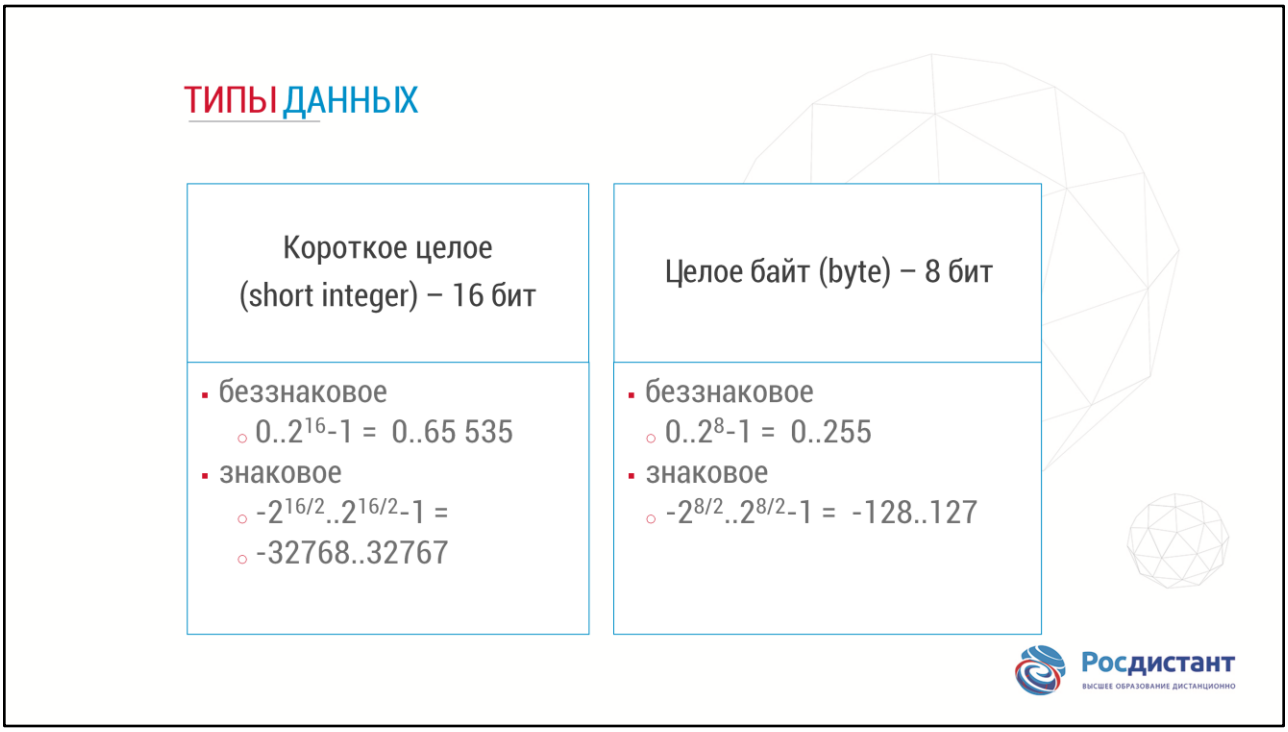
В реальных вычислительных системах память имеет довольно сложную организацию. Процессор работает с памятью, которая не только превышает максимальный размер адресации, но и размер физической памяти. Например, мы условились считать наш компьютер 16-разрядным, а, следовательно, процессор может обращаться к памяти размером только $2^{16} = 65536$. Однако современные компьютеры, ориентированные на многозадачность, позволяют программам взаимодействовать с так называемой виртуальной памятью, которая организуется по различным моделям.

Так как изучение моделей виртуальной памяти не является предметом изучения данного курса, рассмотрим понятия сегмента и смещения. Вся память разделена на сегменты – поддерживаемые на аппаратном уровне блоки памяти. Обычно размер сегмента составляет 2^{16} байт.

Доступ к данным внутри сегмента осуществляется линейно от начала сегмента. На сколько байт нужно отступить, говорит смещение. На английском языке – offset.

Таким образом, физический адрес ячейки памяти записывается в виде пары «сегмент – смещение».

Именно такая адресация используется при работе с указателями.



Типы данных

На предыдущих примерах мы рассматривали случай работы с шестнадцатиразрядными целыми числами и считали, что указатель показывает именно на эти числа.

Но что делать, если нам надо хранить знаковые и беззнаковые числа? Как организовать работы с целыми числами, которые занимают 32 или 64 бита? Как работать с вещественными числами, символами, строками и так далее?

Если хранить в голове, сколько байт надо загрузить с адреса, на который показывает указатель, и как интерпретировать эти данные, то становится ясно, что написать более-менее большую программу практически невозможно.

Для решения этой задачи были введены типы данных.

Например, пусть тип короткого целого short имеет длину 16 бит, а, следовательно, совпадает со словом нашего компьютера.

Короткие целые могут быть беззнаковыми и знаковыми. Следовательно, можно посчитать, что короткое беззнаковое целое может лежать в диапазоне $0..2^{16}–1 = 0..65535$.

А короткое знаковое – в диапазоне $–2^{16/2}..2^{16/2}–1 = –32768..32767$.

Целое длиной в байт в нашем компьютере занимает полслова. Назовем этот тип byte.

Беззнаковое целое длиной байт имеет диапазон $0..2^{8}–1 = 0..255$.

А короткое знаковое – в диапазоне $–2^{8/2}..2^{8/2}–1 = –128..127$.

ЦЕЛЫЕ ТИПЫ

	int 32 бита	long 64 бита	int128 128 бит
беззнаковое	0.. 4 294 967 295 $(0..2^{32}-1)$	0..18 446 744 073 709 551 615 $(0..2^{64}-1)$	$0..2^{128}-1$
знаковое	−2 147 483 648 .. +2 147 483 647 $(-2^{32/2}..2^{32/2}-1)$	−9 223 372 036 854 775 808 .. +9 223 372 036 854 775 807 $(-2^{64/2}..2^{64/2}-1)$	$-2^{128/2}..2^{128/2}-1$

Росдистант
ВЫСШЕЕ ОБРАЗОВАНИЕ ДИСТАНЦИОННО

Целые типы

Для разработки алгоритмов использования только коротких целых явно недостаточно, поэтому целые числа могут занимать 32, 64 и даже 128 бит, что соответствует двум, четырем и восьми словам нашего вымышленного компьютера. Назовем эти типы int, от английского integer – целый, long, от английского long – длинный.

И тот, и другой типы могут быть знаковыми и беззнаковыми. Диапазоны указанных типов приведены на слайде.

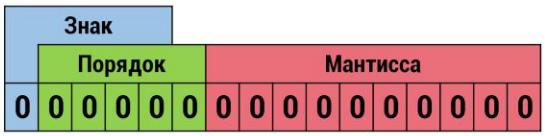
Если мы используем указатель на длинное беззнаковое целое, то процессор должен с указанного адреса скачать уже четыре слова и интерпретировать их как 64-разрядное беззнаковое целое.

Если же мы объявили указатель на знаковое целое, то процессор должен с указанного адреса скачать два слова и интерпретировать их как 32-разрядное целое, причем старший разряд будет содержать флаг знака. Если в старшем разряде ноль, то число положительное, если единица, то отрицательное.

ВЕЩЕСТВЕННЫЕ ТИПЫ

IEEE 754

$(-1)^s$ M B^E



- половинная точность - 16 бит
- M – 10 разрядов
- E – 5 разрядов
- одинарная точность - 32 бита
- M – 23 разряда
- E – 8 разрядов
- двойная точность - 64 бита
- M – 52 разряда
- E – 11 разряда
- четвертная точность - 128 бит
- M – 112 разрядов
- E – 15 разрядов

Росдистант
ВЫСШЕЕ ОБРАЗОВАНИЕ ДИСТАНЦИОННО

Вещественные типы

Аналогично целым типам в языках программирования вводятся вещественные типы, предназначенные для работы с числами с плавающей запятой.

Для унификации работы с такими числами и, как следствие, переносимости программ и данных с одной архитектуры компьютера на другую был разработан международный стандарт IEEE 754.

Данные стандарт описывает не только представление вещественных чисел в памяти компьютера, но и представление таких понятий, как «бесконечность» и «не число».

В соответствии со стандартом 754, для представления вещественных чисел используется три числа: знак, мантисса и экспонента, или порядок. Вещественные числа могут записываться в двоичной или десятичной системе счисления. С учетом сказанного значение вещественного числа может быть вычислено по формуле: $(-1)^s \; M \; B^E$,

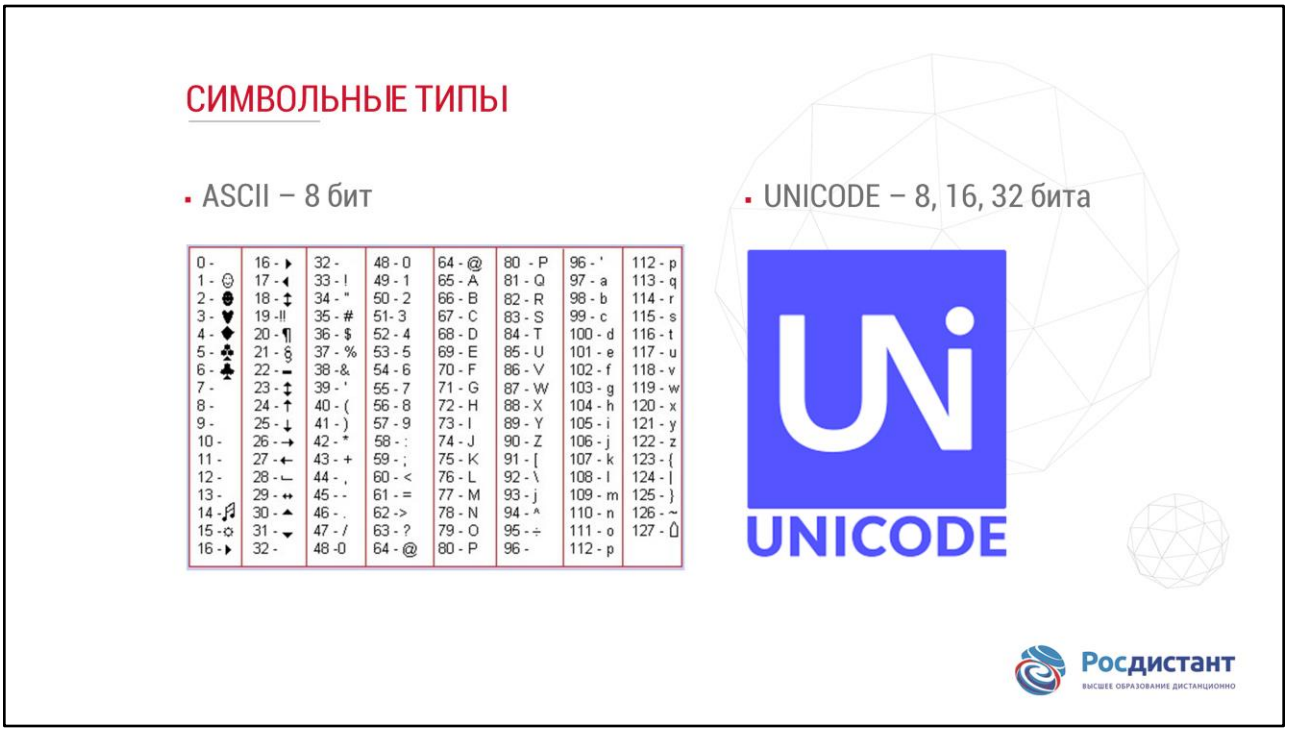
где *s* [эс] – знак числа, ноль, если положительное, и 1, если отрицательное, M – мантисса, B– основание 2 или 10, E – экспонента.

Нетрудно заметить, что мантисса определяет точность представления вещественного числа, а экспонента – диапазон.

Стандарт IEEE 754 сегодня определяет четыре размера для записи вещественных чисел:

- числа с половинной точностью, занимающие 16 бит;
- числа с одинарной точностью, занимающие 32 бита;
- числа с двойной точностью, занимающие 64 бита;
- числа с четвертной точностью, занимающие 128 бит.

Структура записи и диапазоны значений указанных вещественных типов представлены в таблице на слайде.



0 -	16 - ▶	32 -	48 - 0	64 - @	80 - P	96 - '	112 - p
1 - ☺	17 - ◀	33 - !	49 - 1	65 - A	81 - Q	97 - a	113 - q
2 - ☻	18 - ↕	34 - "	50 - 2	66 - B	82 - R	98 - b	114 - r
3 - ♥	19 - ‼	35 - #	51- 3	67 - C	83 - S	99 - c	115 - s
4 - ♦	20 - ¶	36 - $	52 - 4	68 - D	84 - T	100 - d	116 - t
5 - ♣	21 - §	37 - %	53 - 5	69 - E	85 - U	101 - e	117 - u
6 - ♠	22 - ▬	38 -&	54 - 6	70 - F	86 - V	102 - f	118 - v
7 -	23 - ↨	39 - '	55 - 7	71 - G	87 - W	103 - g	119 - w
8 -	24 - ↑	40 - (	56 - 8	72 - H	88 - X	104 - h	120 - x
9 -	25 - ↓	41 -)	57 - 9	73 - I	89 - Y	105 - i	121 - y
10 -	26 - →	42 - *	58 - :	74 - J	90 - Z	106 - j	122 - z
11 -	27 - ←	43 - +	59 - ;	75 - K	91 - [	107 - k	123 - {
12 -	28 - ∟	44 - ,	60 - <	76 - L	92 - \	108 - l	124 - |
13 -	29 - ↔	45 - -	61 - =	77 - M	93 - j	109 - m	125 - }
14 - ♫	30 - ▲	46 - .	62 ->	78 - N	94 - ^	110 - n	126 - ~
15 - ☼	31 - ▼	47 - /	63 - ?	79 - O	95 - ÷	111 - o	127 - ⌂
16 - ▶	32 -	48 -0	64 - @	80 - P	96 -	112 - p	

Символьные типы

Как вы знаете, программы оперируют не только с числами. Современные прикладные программы предназначены обычно для работы с текстовой информацией, которая представляется в символьном и строковом типах.

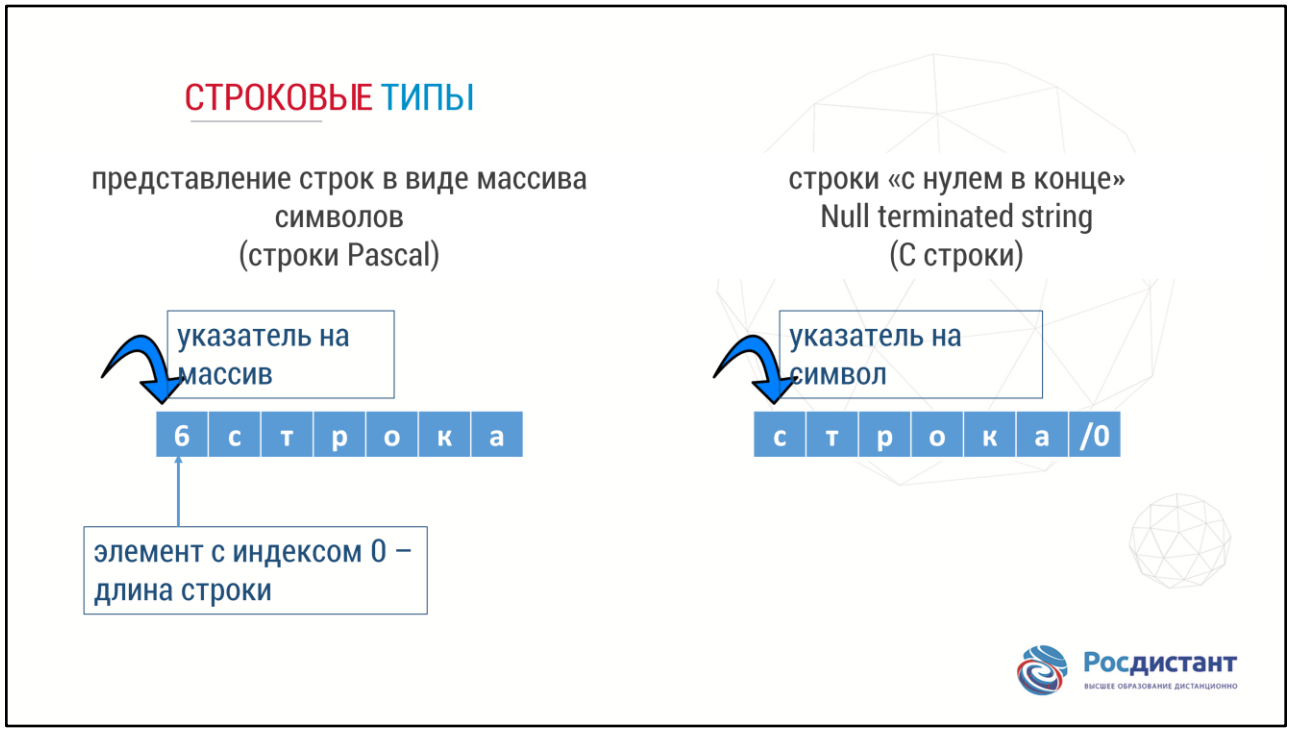
При обработке символов процессор оперирует с их кодами, которые представляются беззнаковыми целыми различной длины.

Первая массовая символьная кодировка ASCII предполагает использование целых чисел длиной в байт. Как мы отмечали выше, при такой кодировке мы можем закодировать 256 символов с номерами от ноля до 255. Первые 32 символа с номерами от ноля до 31 являются служебными. Символы с номерами от 32 до 127 закреплены в стандарте, а вторая половина с номерами от 128 до 255 служит для кодировки различных символов, включая символы национальных языков, таких как русский, испанский, французский и других. Например, символ с номером 10 – спецсимвол «конец строк», 13 – «возврат каретки», 32 – пробел, 49 – единица, 124 – вертикальная черта.

Исходя из того, что символы кодируются целыми числами, такие языки программирования, как Си, относят символьные типы к целочисленным и, следовательно, позволяют применять к ним целочисленные операции, такие как умножение и сложение.

Восьмибитовая кодировка имеет один существенный недостаток. Так как вторая часть таблицы символов используется для кодирования различных национальных языков, то для чтения текста на национальном языке

необходимо использовать специальный шрифт или таблицу подстановки национальных символов. Так, если в программе ASCII [э́ски] встречаются символы на русском языке, то для ее чтения требуется специальный адаптированный для русского языка шрифт или таблица символов.
Для кодировки бо̀льшего количества символов были предложены кодировки больше 8 разрядов. Примером такой кодировки является кодировка UNICODE [юникоуд], позволяющая хранить коды символов в восьми-, шестнадцати- и 32-хбитовом представлении. Такие кодировки позволяют представить не только символы большинства языков мира, но и математические, музыкальные и иные знаки.
Таким образом, для программирования алгоритмов вы можете использовать над символами указатели на различные по разрядности символьные типы и давать указание процессору на соответствующие загрузки и интерпретацию данных.



Строковые типы

Для обработки текстовой информации одних только символьных типов недостаточно. Действительно, бо̀льшая часть такой информации представлена строками – последовательностями символов.

Существует несколько способов представления строк в памяти компьютера, но основными для универсальных языков программирования являются два базовых:

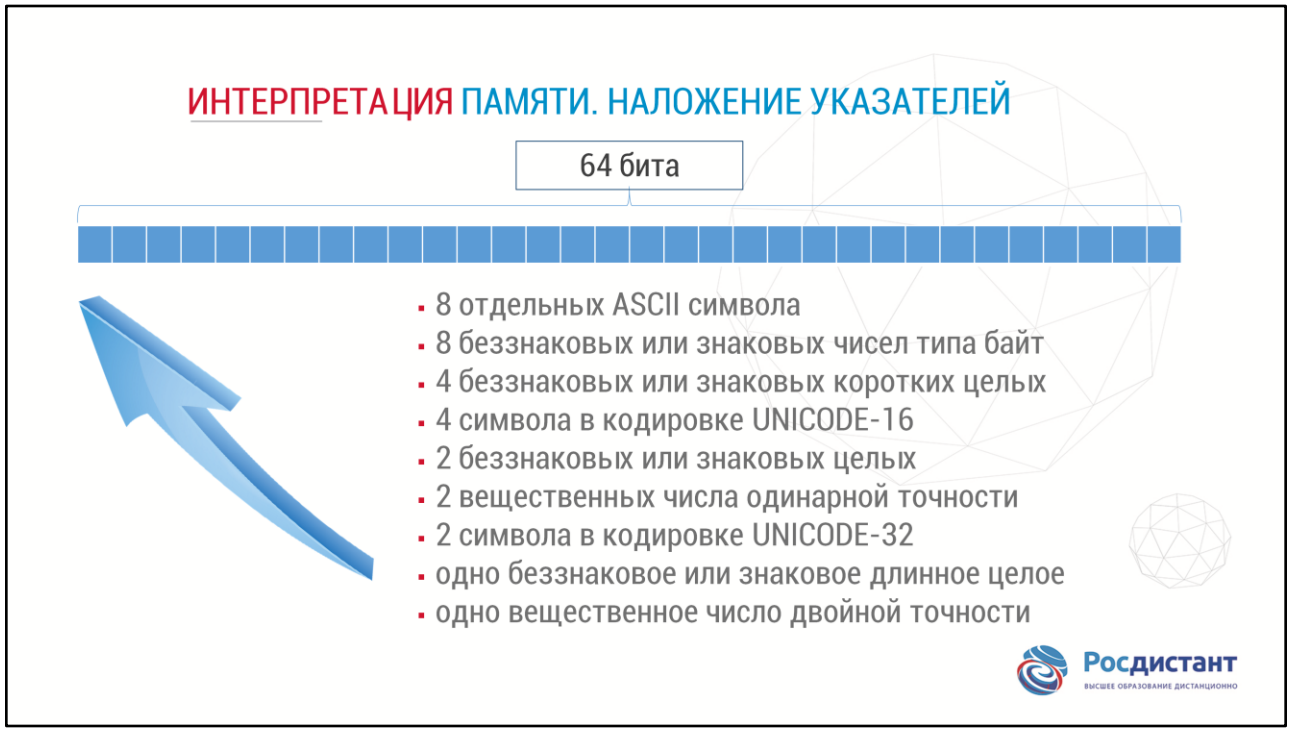
представление строк в виде массива символов;

строки «с нолем в конце».

Представление строк в виде массива символов было впервые реализовано Никлаусом Виртом при разработке языка Паскаль. Такие строки получили название «строки Паскаля» – Pascal string. В строках языка Паскаль для кодировки символов использовалось восьмибитовое представление символов. Причем первый элемент массива, по сути чисел типа байт, показывал длины строки в символах. Поэтому программисты часто в высокопроизводительных алгоритмах на Паскале использовали следующий трюк для определения длины строки. Они не использовали стандартную функцию определения длины строки, а просто обращались к элементу массива с индексом ноль.

В языке Си используется другая схема хранения строк. По сути, строка – это указатель на первый символ. Строкой считается вся память, начинающаяся с этого адреса и до ячейки памяти, в которой хранится символ с кодом ноль. Поэтому такие строки называют Си-строками, строками с нолем в конце или, на

английском языке, Null terminated string.
Каждый из способов имеет свои достоинства и недостатки. Программист, использующий тот или иной язык программирования, должен понимать, какие способы представления строк в памяти использует тот или иной язык программирования.



Интерпретация памяти. Наложение указателей

Подводя итог по указателям и типам данных, необходимо отметить, что одна и та же область памяти может интерпретироваться по-разному.

Рассмотрим, например, последовательные 8 байт, или 64 бита памяти. Что может быть в них записано? Как процессору интерпретировать последовательность из 64 нолей и единиц?

Мы только что рассмотрели основные типы данных, давайте дадим ответ исходя из пройденного материала.

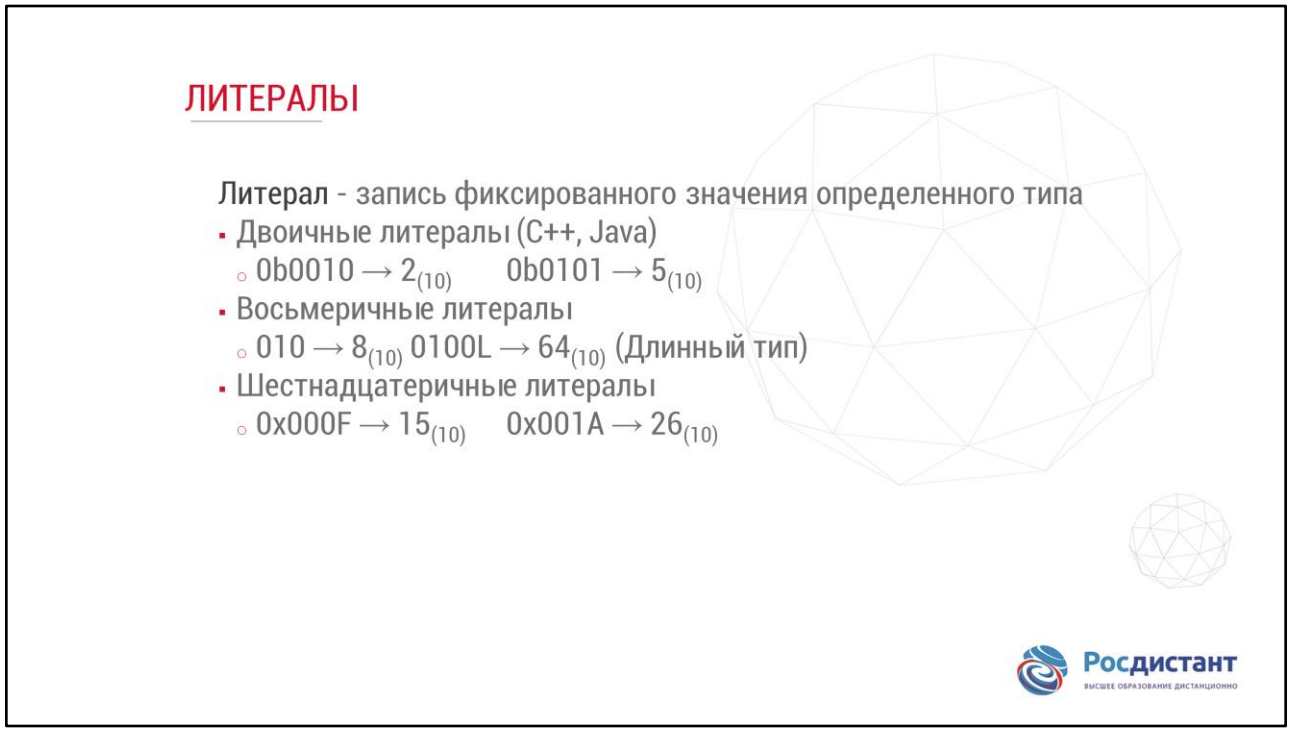
Это могут быть:

- 8 отдельных символов в кодировке ASCII;
- 8 беззнаковых или знаковых чисел типа байт;
- 4 беззнаковых или знаковых коротких целых;
- 4 символа в кодировке UNICODE-16;
- 2 беззнаковых или знаковых целых;
- 2 вещественных числа одинарной точности;
- 2 символа в кодировке UNICODE-32;
- одно беззнаковое или знаковое длинное целое;
- одно вещественное число двойной точности.

Именно тип указателя говорит о том, как интерпретировать эту память, распределяя ее, к примеру, в своих регистрах.

Такие языки, как Си и Си++, позволяют налагать разные типизированные указатели на один и тот же участок памяти. Однако это, скорее всего, опасная

необходимость системного программирования. При разработке прикладного программного обеспечения использование таких механизмов не может быть оправдано.



Литералы

Мы рассмотрели применение указателей в императивных программах. Напомним, что императивная программа на входе получает некоторые входные данные. Возникает логичный вопрос: как записать исходные дынные? Как указать значения того или иного типа?

В языках программирования для задания значений «как есть» используются литералы. Начальные значения, используя именно литералы и указатели в первых программах, задают по адресу, на который показывает указатель. Записывается значение, описанное литералом.

Литералы привязаны к типам. Поэтому рассмотрим литералы различных типов.

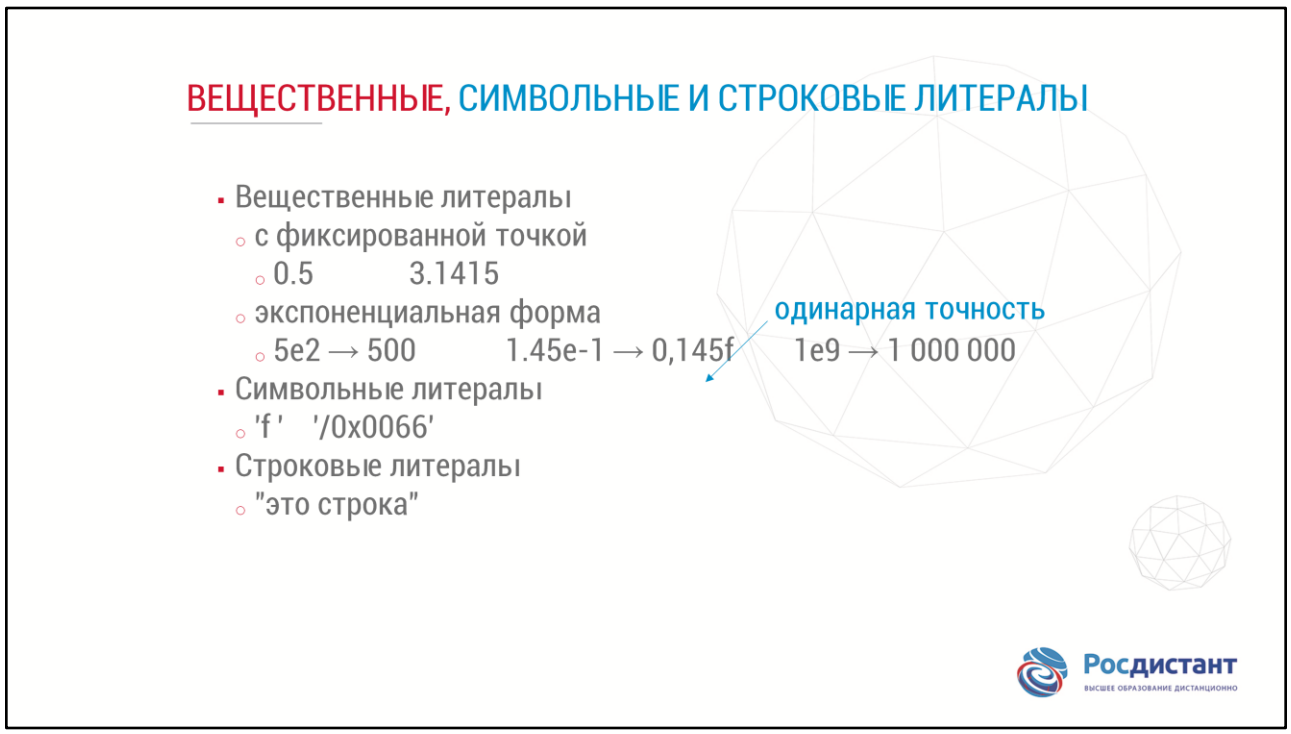
Целочисленные литералы представляют значения целых чисел в основных для вычислительных систем системах счисления: двоичной, восьмеричной, шестнадцатеричной. На практике можно встретить также двоично-восьмеричную и иные системы.

Изначально не многие универсальные языки программирования имели возможность представлять значения в виде двоичных литералов. Однако в последствии этот недостаток был устранен. Сейчас такие языки программирования, как Си++ и Java, позволяют это сделать. В качестве примера возьмем язык Си++.

Двоичные литералы начинаются с префикса 0b, например: литерал 0b0010 представляет десятичное два, а литерал 0b0101 – десятичное пять.

Восьмеричные литералы в Си++ с символа ноль, поэтому число 010 будет в

десятичной системе означать восемь, а 0100 – 64.
Десятичные литералы записываются как есть, но не с нолем в начале.
Шестнадцатеричные литералы начинаются с префикса 0x.
Для того чтобы указать, что литерал представляет собой длинное целое, в конце можно приписать английскую «эль», строчную или заглавную. Однако строчная «эль» очень похожа на единицу, поэтому существует рекомендация при записи литералов длинного целого использовать заглавную букву «эль», как показано в примере на слайде.



Вещественные, символьные и строковые литералы

Аналогично целочисленным литералам существуют вещественные, символьные и сроковые литералы.

Для записи значений вещественного типа используются две формы литералов: запись в форме с фиксированной точкой и экспоненциальные литералы.

Вещественные литералы с фиксированной точкой записывают значение «как есть», используя в качестве разделителя целой и дробной частей символ точки.

В экспоненциальной форме литералы записываются в виде «мантисса e порядок». По правилам, мантисса – это число в формате фиксированной точки на интервале, исключительно от ноля до 10. Порядок показывает, на какую степень числа 10 надо умножить мантиссу, чтобы получить исходное число.

Например, запись 5e2 [пять экспонента два] означает 500: пять умножить на 10 в степени два, то есть на сто.

1.45e–1 означает 0,145.

1e9 представляет миллион.

Для указания принадлежности к вещественному типу одинарной точности, который на Си-подобных языках называется float, можно использовать суффикс f [эф строчная] или F [эф заглавная]. Иначе литерал будет отнесен к вещественному типу двойной точности.

Символьные литералы также могут записываться в нескольких видах:

- в форме символа, который обрамляется в разных языках одинарными или двойными кавычками;

- в форме номера символа;
- в форме номера через слеш, причем запись также заключена в кавычки.

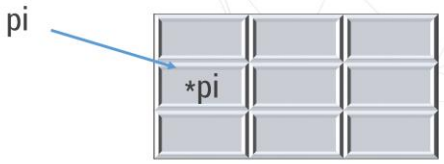
Например, следующие записи показывают запись одного и того же символа – английской буквы f:

- `f`;
- 102 или 0146 – десятичный и восьмеричный коды;
- `/0x0066’ – шестнадцатеричный код.

И наконец, строковые литералы записываются как есть и обычно в двойных кавычках, например, "это строка".

РАЗЫМЕНОВАНИЕ УКАЗАТЕЛЕЙ

- Объявление
  int* pi;
- Обращение (разыменование)
  *pi = 10;



Росдистант
ВЫСШЕЕ ОБРАЗОВАНИЕ ДИСТАНЦИОННО

Разыменование указателей

Мы научились адресоваться к памяти с помощью типизированных указателей – указателей, которые не просто указывают на ячейку памяти, но и говорят, сколько памяти надо взять и интерпретировать в указанный тип.

Очевиден вопрос: поскольку указатель – это адрес в памяти, то как «сказать» процессору, что по адресу, указанному указателем, надо записать значение? Для этого служит механизм разыменования указателя. Рассмотрим пример на языке Си.

объявление

int* pi;.

Этот пример говорит, что указатель pi будет указывать на целочисленное значение типа инт. Чтобы по адресу указателя записать значение, к примеру 10, нам надо разыменовать указатель. Тем самым мы скажем процессору «по адресу, на который указывает указатель, запиши число 10»:

*pi = 10;.

Как мы видим, работа с указателями достаточно сложна. Языки программирования, такие как Си и Си++, позволяют работать с указателями достаточно изощренно, что не способствует высокому качеству разрабатываемого прикладного программного обеспечения. Однако понимание работы с указателями критически необходимо любому программисту, даже если в выбранном языке программирования нет указателей в явном виде.

Переменные и константы

Для устранения прямой работы с указателями, повышения читабельности и надежности программ в первые высокоуровневые языки программирования были введены переменные и константы.

Переменная – это алиас, или заменитель указателя. Если мы объявляем переменную икс целого типа, то, по сути, мы говорим компилятору: «Зарезервируй в памяти место под целый тип, а указатель на эту область памяти назови икс». При этом мы избавляемся от необходимости разыменовывать указатель. Например, выполнив следующее объявление:

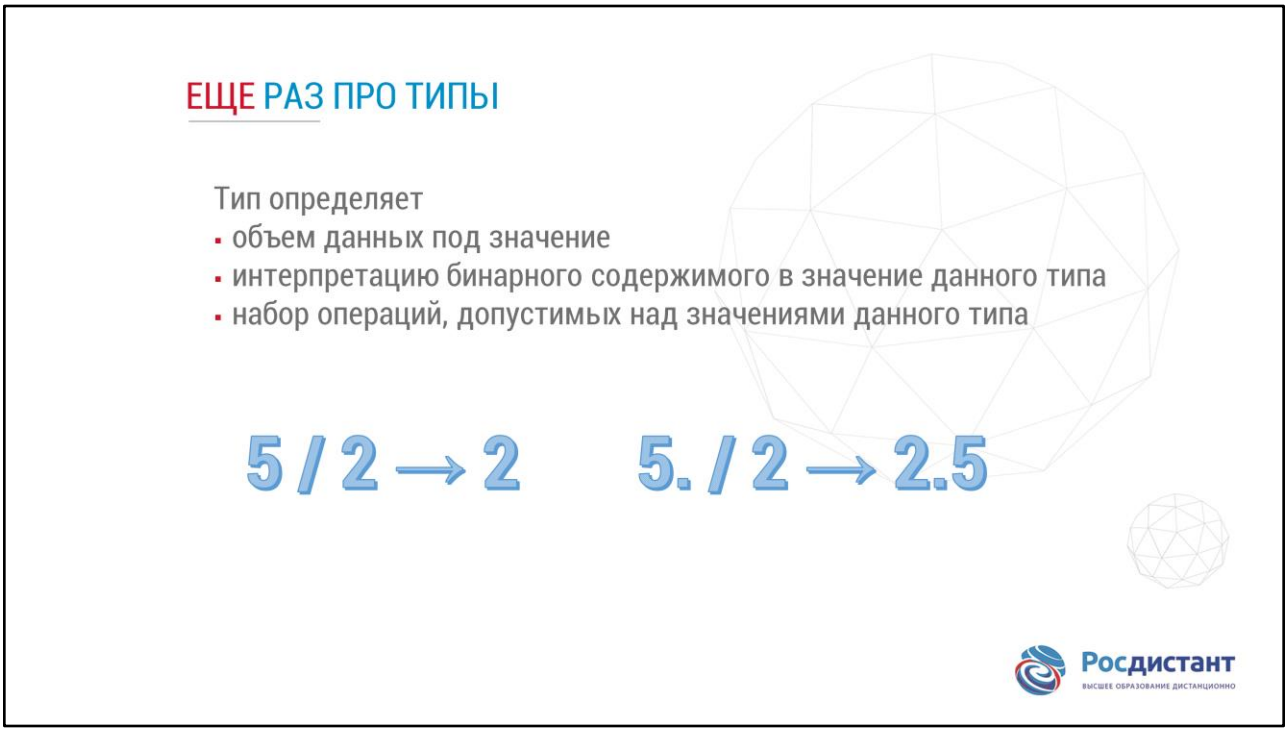
int x;,

мы получаем переменную x, которая содержит целочисленное значение. Далее мы можем как считывать, так и записывать в эту переменную данные, например:

x = 10;.

Константы аналогичны переменным, только присвоение значения константы происходит один раз во время ее инициализации и не может быть изменено в дальнейшем. Например, следующее объявление есть объявление вещественной константы:

const float pi = 3.1415;.



Еще раз про типы

Типизированные указатели и переменные не только определяют интерпретацию памяти процессором, типы данных позволяют также определить, какие операции возможны над данными того или иного типа.

Рассмотрим несколько примеров.

Целочисленные данные. Над целыми числами, как правило, определены следующие операции: сложение, умножение, вычитание, целочисленное и вещественное деление и т. д.

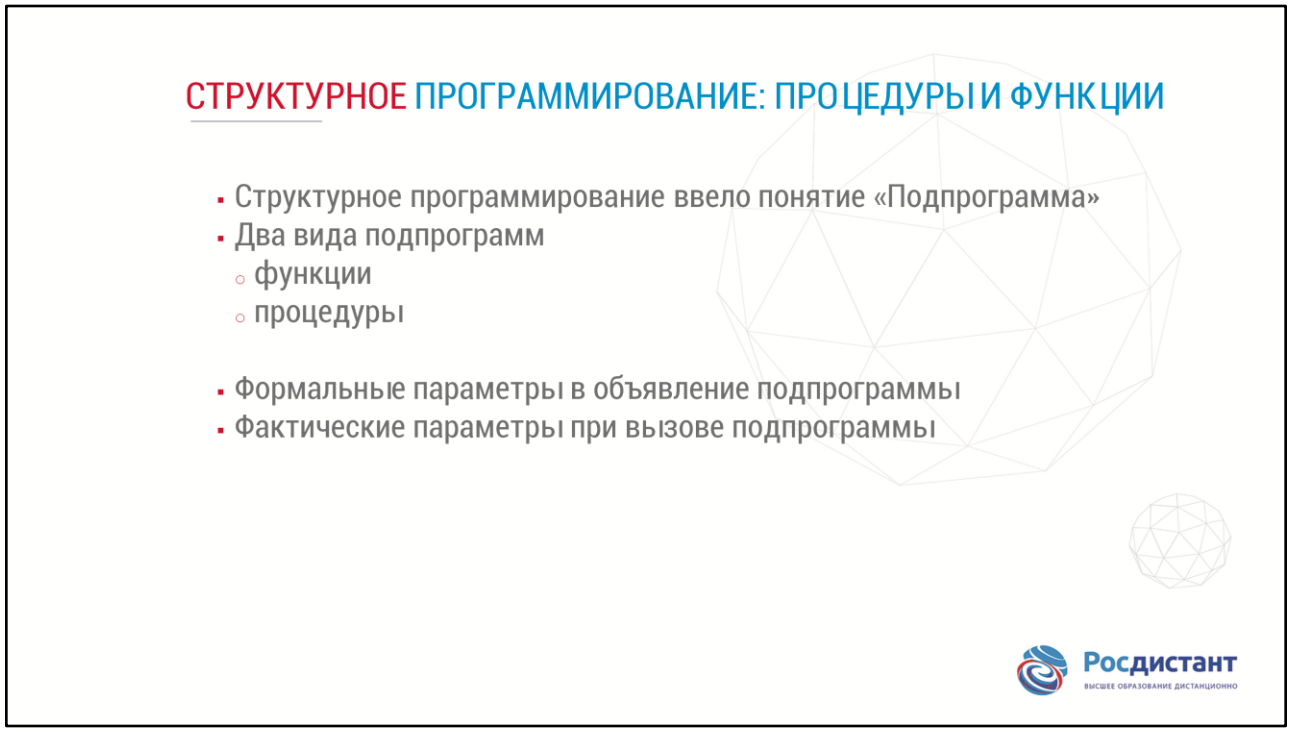
Если операции сложения, вычитания и умножения интуитивно понятны, то операции деления над целыми числами требуют отдельного пояснения.

В обыденной жизни мы можем без труда поделить 5 на 2 и получить результат 2,5. Такое действие выполняет операция вещественного деления, когда результатом операции является вещественное число.

В отличие от вещественного деления, результатом целочисленного деления будет целое число. Поэтому целочисленно поделив 5 на 2, мы получим 2!

Надо быть осторожным при использовании того или иного языка программирования. Так, ряд языков программирования четко разделяет операции целочисленного деления, например, операция див в Паскале. Другая часть языков, например Си, выполняет операцию деления в зависимости от типа операндов. Если на Си вы делите два целочисленных числа, то операция будет выполнена как целочисленная, если хотя бы один из операндов будет вещественный, то будет выполнена операция вещественного деления.

Аналогично ведут себя языки программирования и по отношению к символьным типам. Мы уже упоминали, что на Си и Си++ символьный тип отнесен к целочисленным, а значит, к ним можно применять все операции целочисленной арифметики. Другие языки, такие как Джава, Си Шарп, Паскаль, позволяют только вычислять код символа.
Над строками применима операция конкатенации, в результате которой из двух или более строк получается одна объединенная строка.

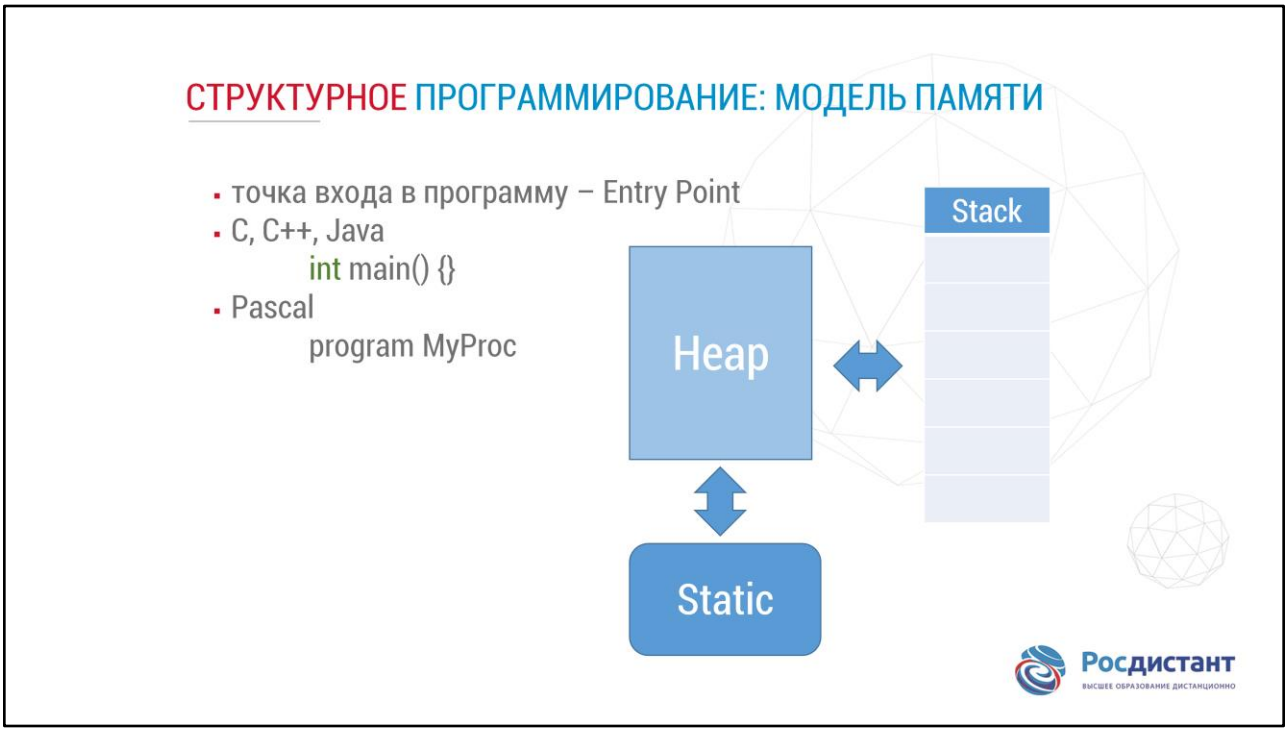


Структурное программирование: процедуры и функции

Итак, мы рассмотрели модель памяти императивного программирования, которая подразумевала выделение участков памяти определенной длины под данные некоторого типа.

Как мы говорили ранее, структурное программирование ввело понятие подпрограммы – поименованного блока операторов, в который можно передавать и из которого можно получать данные. Такие данные называются параметрами. Параметры, которые используются при описании подпрограммы, называются формальными и при вызове подпрограммы замещаются фактическими.

В современных языках программирования различают два вида подпрограмм – процедуры и функции. Функция отличается от процедур тем, что она может возвращать одно значение определенного вида. Становится очевидным, что процедура – это функция, которая ничего не возвращает. Поэтому в таких языках, как Си и Си++, процедур как таковых нет, а есть только функции.



Структурное программирование: модель памяти

Программа на структурном языке программирования начинается с некоторой программной точки, называемой точкой входа. В таких языках программирования, как Си, Си++, Java и других, точкой входа является функция или метод main. Именно с вызова этой функции начинается выполнение программы. В языке Паскаль точкой входа является модуль, объявленный ключевым словом program.

После запуска точки входа главная функция может вызывать другие функции, которые могут вызывать третьи и т.д. Через вызов функций реализуются сложные алгоритмы. Поэтому возникает вопрос, как должна быть организована память в этом случае. Для работы программы выделяется несколько областей памяти, как показано на слайде:

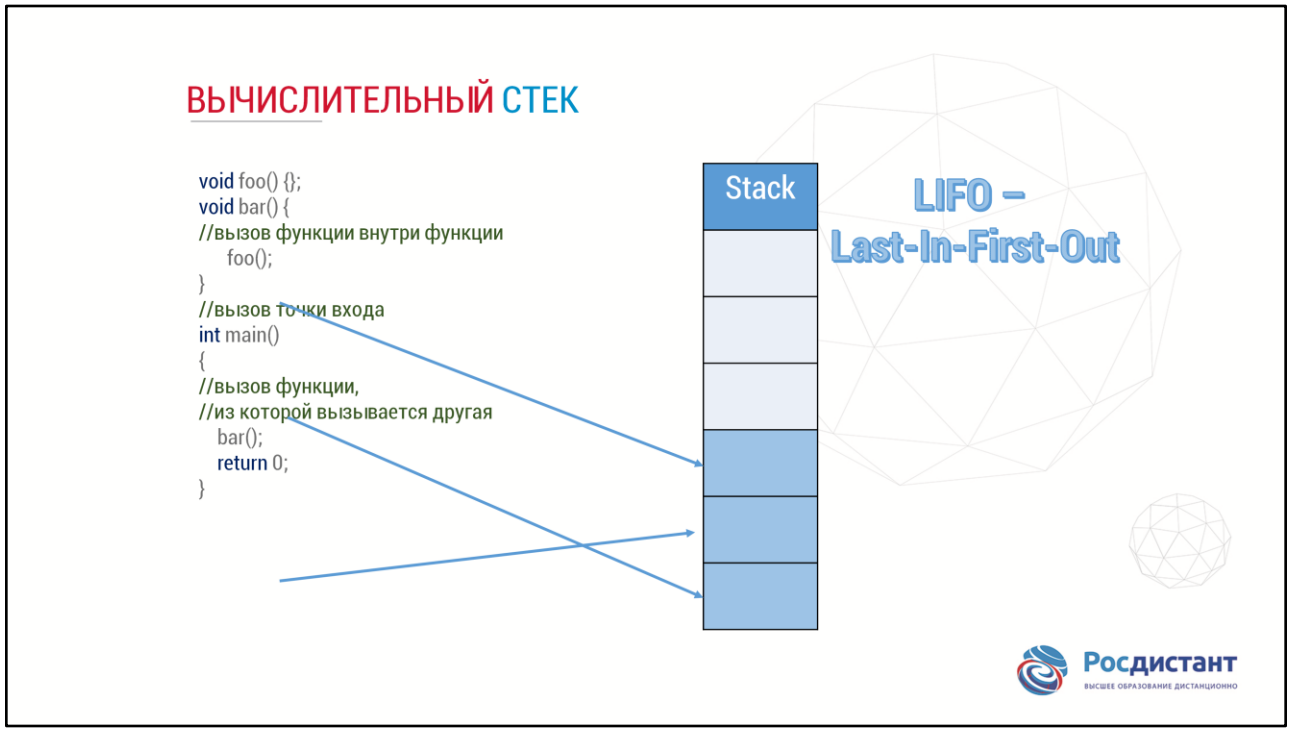
- стеки для организации вызова функций;
- область памяти для хранения статических данных;
- куча для хранения динамических данных.

При выполнении операционной системой или виртуальной машиной программы назначается вычислительный процесс, внутри которого выделяются указанные области памяти: стек, статическая область и куча.

Внутри вычислительного процесса для организации вызова функций порождается как минимум одна вычислительная нить, по-английски thread. А под каждую вычислительную нить организуется свой стек.

В простых однопоточных программах – одна вычислительная нить и,

следовательно, один стек.
В многопоточных программах создаются несколько вычислительных нитей и соответствующее количество стеков. В дальнейшем будем рассматривать однопоточные приложения.
Статическая область служит для хранения глобальных данных, доступных из любой точки программы.
Куча служит для хранения динамических данных, то есть данных, которые создаются и уничтожаются во время выполнения программы.



Вычислительный стек

При запуске программы, как мы говорили, для вычислительной нити создается стек, состоящий из специальных областей – фреймов. Фрейм стека служит для передачи параметров в функцию и из нее, а также хранения локальных данных, необходимых для вычисления.

При старте программы за счет вызова главной функции в стеке создается первый фрейм. Если главная функция вызывает вторую, то в стеке создается второй фрейм и т. д. Стеки «накладываются» друг на друга, как стопки тарелок, организуя дисциплину обслуживания LIFO – «последний-пришел-первым-вышел». Предположим, программа вызвала три функции – одну из другой. Тогда в стеке будет три фрейма. Когда третья функция завершит работу, фрейм будет разрушен, а управление перейдет ко второй функции. Когда вторая функция завершит работу, то и ее фрейм завершит работу, и управление перейдет к главной функции.

Для работы с фреймами исполняющая среда использует так называемый программный счетчик – указатель на активный фрейм.

Из описания работы стека можно сказать, что время жизни программных объектов, которые размещаются в стеке, ограничено временем выполнения функции. Это правило необходимо запомнить, поскольку оно определяет способы передачи параметров в функции, которые мы рассмотрим немного позже.

Куча и способы управления памятью

В куче, как мы уже отмечали, во время выполнения программы создаются и уничтожаются данные. Данные, располагающиеся в куче, разделяются между стеками, то есть из любого стека можно получить доступ к данным в куче. Это вызывает определенные проблемы при многопоточном программировании.

На языке Си++ объекты и данные, которые располагаются в куче, создаются через указатели.

На языке Java все объекты и массивы создаются в куче.

Знание таких особенностей языков программирования позволяет избежать множества неприятных ошибок.

Куча, как и стек, имеет ограничение на объем используемой памяти. При превышении максимального размера программа аварийно завершается. Поэтому важно удалять в куче данные, которые уже не используются.

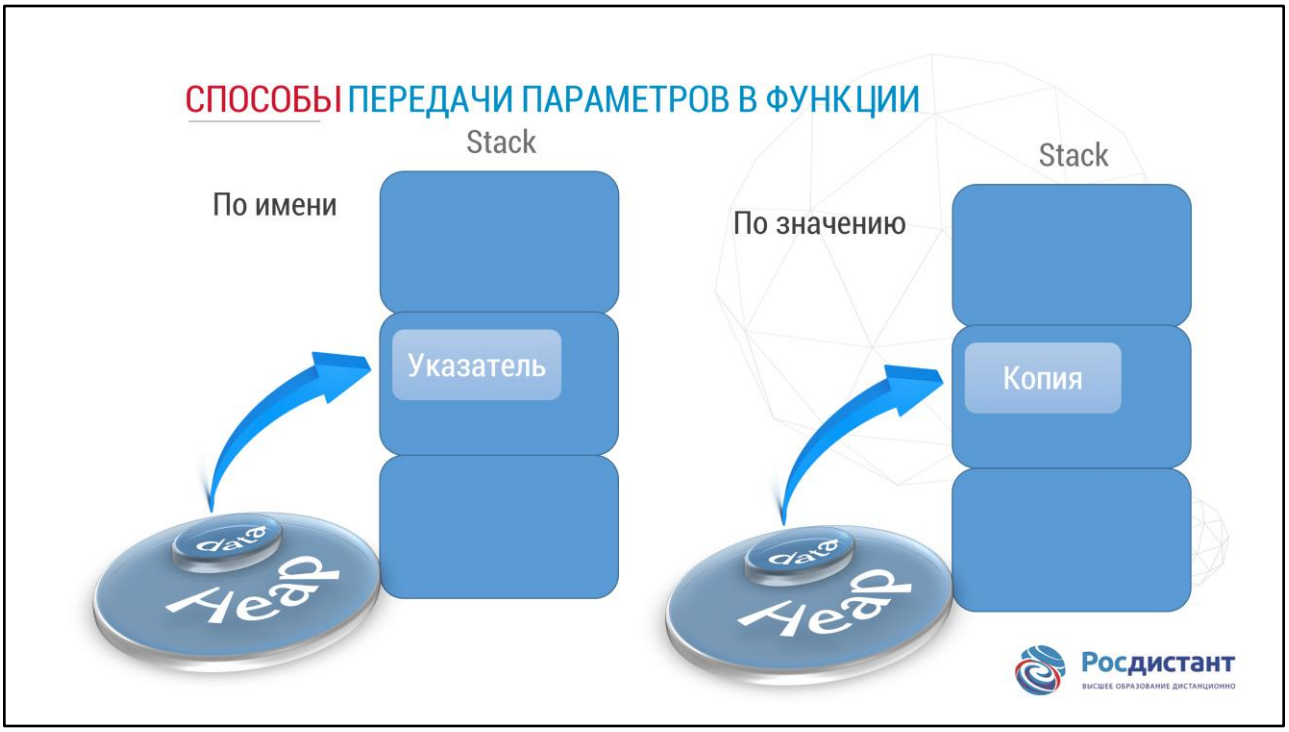
Существует два вида управления памятью, или кучей: ручное и автоматическое.

К языкам с ручным управлением памятью относятся такие языки, как Си и Си++.

Автоматическое управление памятью используется в интерпретируемых языках Java, C#, Python.

При ручном управлении памятью программист должен сам определять, когда уничтожать объект в куче. Как правило, для этой цели требуется вызов деструктора или функции освобождения памяти. Если этого не сделать, то можно получить сложно диагностируемый эффект утечки памяти. Этот эффект наблюдается, когда в программе порождается большое количество объектов в

куче, а часть из них после использования не уничтожается. Это приводит к росту кучи до максимального размера и аварийному завершению программы. Причем при разных запусках крах программы протекает по-разному.
В языках с автоматическим управлением памятью в куче действует специальный процесс, называемый сборщиком мусора. Сборщик мусора анализирует наличие ссылок на объекты. Если на объект никто не ссылается, то он уже не нужен, и сборщик удаляет данный объект.
Автоматическая сборка способствует повышению надежности разрабатываемого программного обеспечения. Поэтому похожую идею стали использовать языки с ручным управлением. Так в Си++11 появилась возможность использовать умные указатели, которые в некоторой степени реализуют алгоритм сборщика мусора.



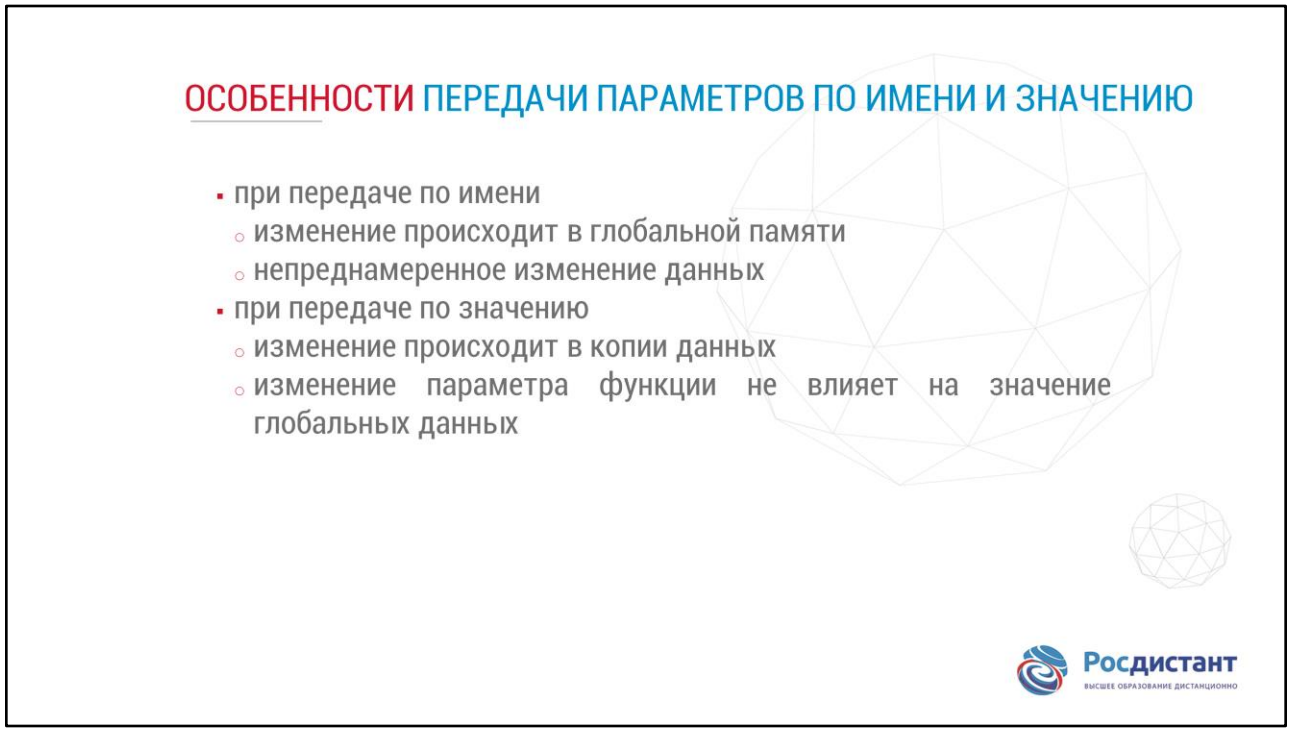
Способы передачи параметров в функции

Как мы уже говорили, при описании функций используются формальные параметры, которые определяют тип и последовательность данных, передаваемых или получаемых из функции.

При вызове функции формальные параметры заменяются фактическими. Использование стеков для организации вызова функций определяет два основных способа передачи параметров: по имени и по значению. Рассмотрим пример, когда на место фактических параметров подставляются данные из кучи. При передаче параметра по имени на место формального параметра, то есть области памяти во фрейме, подставляется указатель на данные, располагаемые в куче. Поэтому, если функция по адресу указателя производит какие-то изменения, то фактически эти изменения происходят в куче. То есть изменяются исходные данные. Если в функцию передавалось целочисленное значение пять по имени, а в функции к этому значению прибавлялась единица, то после завершения работы функции в куче будет число шесть.

При передаче параметра по значению на место формального подставляется копия значения фактического параметра, то есть во фрейме создается копия данных. Например, если в куче хранится целое число со значением пять, то во фрейме будет копия данных. Теперь, если в программе изменяется фактический параметр, то изменяется область памяти во фрейме, а не в куче. Следовательно, если в функцию передавалось целочисленное значение пять по значению, а в функции к этому значению прибавлялась единица, то после завершения работы

функции в куче будет по-прежнему значение пять.



Особенности передачи параметров по имени и значению

Рассмотрим особенности передачи параметров по имени и значению более подробно.

Очевидно, что если передать параметр по имени, то программист, реализующий функцию, может легко модифицировать глобальные данные, даже если это не нужно. Мы уже говорили, что одной из основных причин кризиса программирования 1970-х годов было плохо управляемое изменение глобальных данных. Поэтому правила хорошего тона в программировании говорят, что по умолчанию при описании функции надо использовать передачу по значению. И только в том случае, когда в функции действительно надо модифицировать глобальные данные, использовать передачу по имени.

Это важнейшая особенность языков программирования, которую необходимо четко знать.

Непонимание рассмотренных механизмов может привести к грубым ошибкам. Например, в таких языках, как Java, используется передача параметров только по значению. Но это вовсе не говорит о том, что вы застрахованы от случайного изменения глобальных данных.

Мы уже говорили, что массивы в Джава хранятся в куче и являются ссылочными типами. То есть реально обращение к массивам в Джава происходит через скрытую ссылку или указатель. Поэтому реально передавая в функцию копию указателя на массив, мы тем самым передаем функции адрес массива в куче. Следовательно, думая, что вы модифицируете копию массива – массив ведь

передался по значению, вы на самом деле модифицируете глобальные данные! Отчасти поэтому в Джава все объекты и массивы изначально могут себя клонировать.
Мы рассмотрели лишь основные особенности распределения памяти под программы на ЭВМ. Изучая в дальнейшем языки программирования и архитектуру компьютеров, необходимо рассмотреть эти вопросы более детально.