СПбГУТ им. проф. М.А. Бонч–Бруевича
Кафедра программной инженерии и вычислительной техники (ПИ и ВТ)

ПРОГРАММИРОВАНИЕ

Единственный способ изучать новый язык программирования – писать на нем программы.

Брайэн Керниган

Лекция 10: *Указатели в языке Си*

1. Указатели в языке Си
2. Операции над указателями
3. Нетипизированный указатель
4. Указатели и const
5. Указатель на указатель
6. Указатель файла
7. Указатели и массивы
8. Указатели на функции

Это, пожалуй, самая сложная и самая важная тема во всём курсе. Без понимания указателей дальнейшее изучении си будет бессмысленным. Указатели – простая концепция, логичная, но требующая внимания к деталям.

Санкт–Петербург,

Указатели. Введение

- В архитектуре ЭВМ фон Неймана (*базовый вычислитель*), одним из основных свойств является **линейность** и **однородность оперативной памяти**, а отдельные ячейки памяти идентифицируются **адресами**.
- То, что в языках высокого уровня называют **переменной**, на уровне машинного кода представляет собой не более чем ***область памяти, то есть несколько ячеек памяти, расположенных подряд***, или, иначе говоря, имеющих *последовательные адреса*.
- *Под **адресом области памяти** понимается наименьший из адресов ячеек, составляющих область*.
- Для нас здесь важно то, что **любая переменная** имеет свой **адрес**.
- Во многих языках программирования, включая **Паскаль** и **Си**, **адреса** считаются **информацией**, которую можно хранить и обрабатывать; но если при работе на языке ассемблера адреса ничем не отличаются от обычных чисел, то языки высокого уровня вводят для **адресов** отдельные **типы данных**.
- Как и в **Паскале**, в языке **Си** адресный тип привязан к типу переменной, адрес которой имеется в виду.

Отсюда существуют **два базовых принципа**, которые формулировали для **указателей**:

- ***Указатель*** *— это переменная, в которой хранится адрес.*
- **Утверждение вида «A указывает на B» означает «A содержит адрес B».**

В предыдущих лекциях, ПЗ и ЛР были введены **базовые (основные) типы** языка **Си**. Для их определения и описания используются служебные слова: **char, short, int, long, signed, unsigned, float, double, enum, void**.

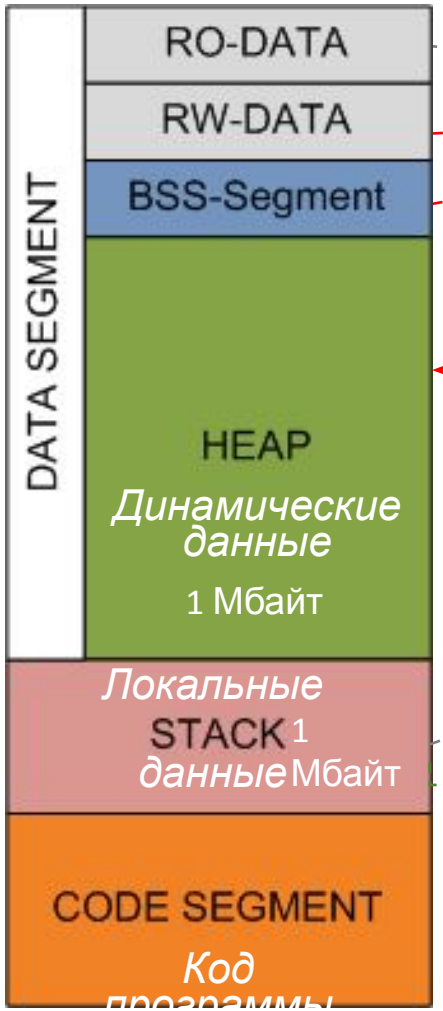
В языке **Си**, кроме базовых типов, разрешено вводить и использовать **производные типы**, каждый из которых получен

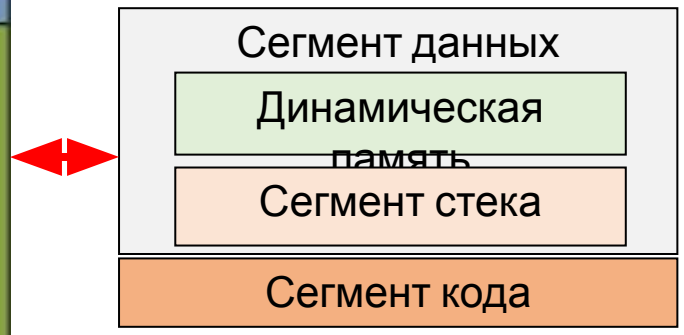
Стандарт языка **Си** определяет три способа получения **производных типов**:

- массив элементов заданного типа;
- указатель на объект заданного типа;
- функция, возвращающая значение заданного типа.

- Каждая **переменная** в программе - *это объект, имеющий имя и значение*.
- По имени можно обратиться к переменной и получить (а затем, например, напечатать) ее значение.
- В операторе присваивания выполняется обратное действие - имени переменной из левой части оператора присваивания ставится в соответствие значение выражения его правой части.
- С точки зрения машинной реализации, имя переменной соответствует адресу того участка памяти, который для нее выделен, а значение переменной - содержимому этого участка памяти.

Указатели. Введение. Упрощенная структура исполняемого файла





Статические данные *распределены в специальном* ***статическом сегменте памяти*** *программы*

- ***Глобальные данные*** объявленные вне функций.
- Данные, объявленные внутри функций как **static** – ***статические локальные данные***:
  - доступны только функции, в которой описаны, но существуют (занимают память) во время выполнения всей программы.

Локальные переменные*, объявленные внутри функций*

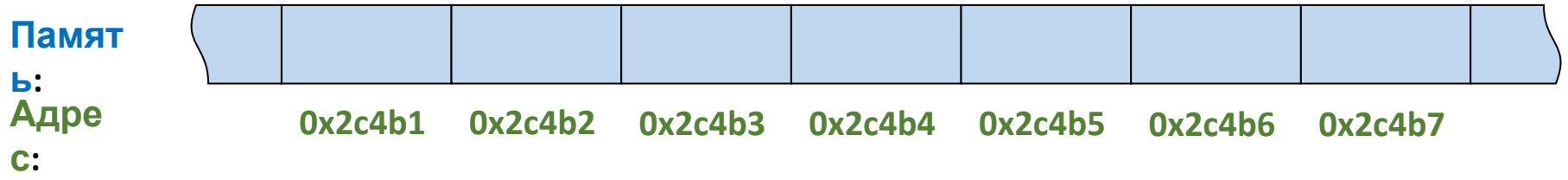
- ***Стек*** *– область памяти, в которой хранятся локальные переменные и адреса возврата*

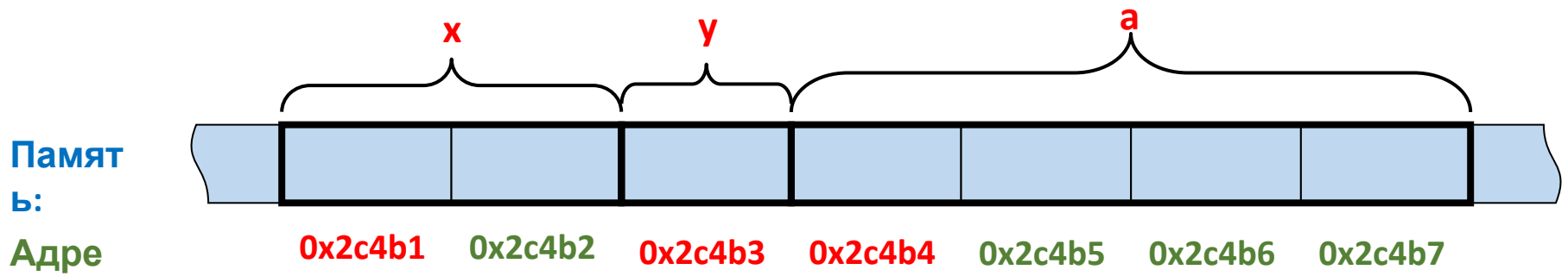
- **BSS-сегмент** (**block started by symbol**) содержит неинициализированные **глобальные переменные**, или **статические переменные без явной инициализации**.
- Этот сегмент начинается непосредственно за **data-сегментом**.
- Обычно загрузчик программ инициализирует **bss** область при загрузке приложения **нулями**.
- Дело в том, что в **data области** **переменные инициализированы** – то есть затирают своими значениями выделенную область памяти.
- Так как переменные в **bss области** **не инициализированы явно**, то они теоретически могли бы иметь значение, которое ранее хранилось в этой области, а это уязвимость, которая предоставляет доступ до приватных (возможно) данных.
- Поэтому загрузчик вынужден обнулять все значения.
- За счёт этого и **неинициализированные глобальные переменные, и статические переменные** **по умолчанию равны нулю.**

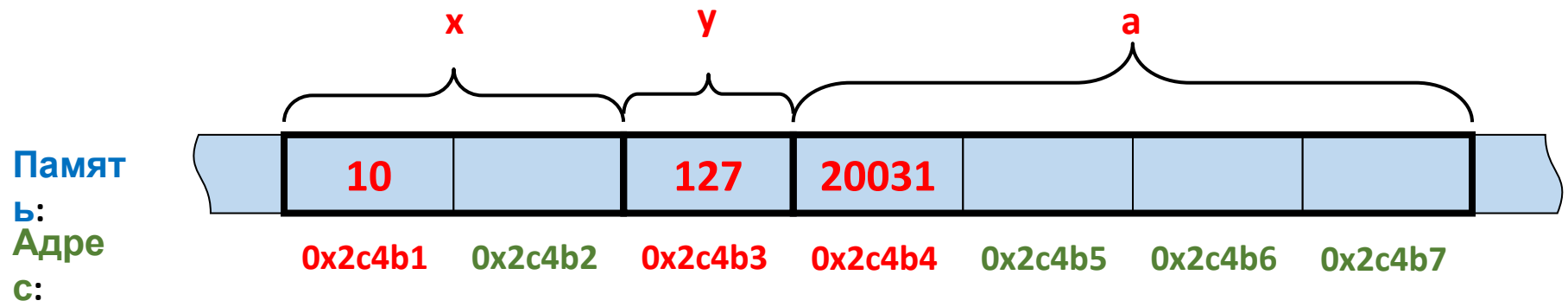
Указатели. Введение

Адрес переменной

Стандарт







- **Оперативная память** организована как последовательность ячеек (байт)
- Каждая **ячейка** имеет собственный адрес (порядковый номер)
- **Адрес** – целое число, чаще записываемое в шестнадцатеричной системе счисления
- Каждая **переменная** размещается в последовательных ячейках (количество ячеек зависит от типа переменной)
- Адрес переменной – адрес первой из этих ячеек

- **Адрес переменной** можно получить с помощью операции **&**
- Например, **&x** даст адрес **x**:

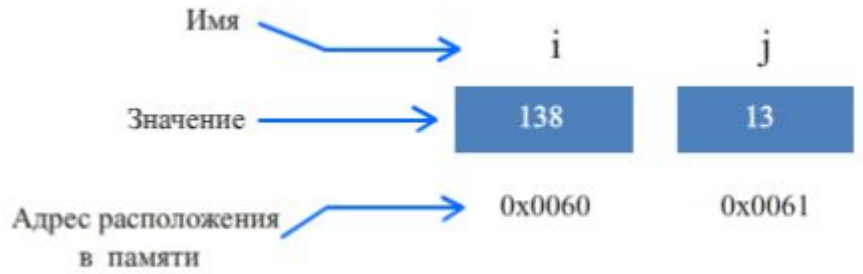
```
printf("x=%d, &x=%p", x, &x);
```

```
x=10, &x=2c4b1
```

1. Указатели в языке Си

При изучении языка **Си** у начинающих часто возникают вопросы связанные с ***указателями:***

- Для чего нужен **указатель**?
- Почему всегда пишут “**указатель типа**” и чем указатель типа uint16_t отличается от указателя типа uint8_t?
- И кто вообще выдумал указатель?



- Указатель, как и другие **переменные**, имеет **тип данных** и **идентификатор**.
- Однако указатели используются таким образом, которой принципиально отличается от того, как мы используем **«нормальные» переменные**, и при объявлении мы должны добавить **звездочку**, чтобы сообщить **компилятору**, что данная переменная должна рассматриваться как **указатель**.
- Синтаксис объявления указателей: **<тип> *<имя>;**

```
float  *pa;
long long  *ptr_b;
```

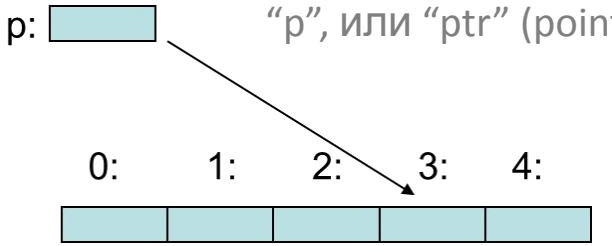
- Для объявления переменной как указателя необходимо перед её именем поставить *, а для получения адреса переменной используется **&** (унарный оператор взятия адреса).
- Идентификатор не обязательно должен содержать символы, которые помечают переменную как указатель (такие как “**p**”, или “**ptr**” (*pointer*)). Тем не менее, рекомендуется использовать это на практике. Это поможет вам сохранить ваши мысли более организованными, и если у вас все указатели будут помечены таким образом, другим

- Указатели объявляются точно так же, как и обычные переменные, только со **звёздочкой** * между **типом данных** и **идентификатором** (справа/посередине/слева???):

```
int  *iPtr; // указатель на значение типа int
double  *dPtr; // указатель на значение типа double
// ниже корректный синтаксис (допустимый, но не желательный):
int*  iPtr3;
int  *  iPtr4; // корректный синтаксис (не делайте так)
  // объявляем два указателя для переменных типа int:
int *iPtr5, *iPtr6;
```

- Синтаксически язык **Си** принимает объявление указателя, когда звёздочка находится **рядом с типом данных, с идентификатором или даже посередине!** Обратите внимание, эта звёздочка не является оператором разыменования. Это всего лишь часть синтаксиса объявления указателя.
- Однако, при объявлении нескольких указателей, звёздочка должна находиться возле каждого идентификатора. Это легко забыть, если вы привыкли указывать звёздочку возле типа данных, а не возле имени переменной.

Указатели



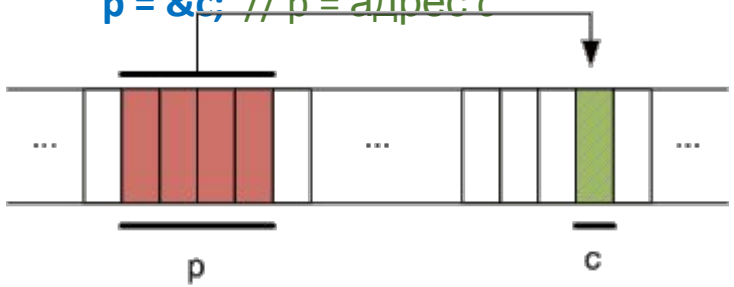

```
char  *p;
```

Итак, ***Указатель*** *– это специальная переменная для хранения адреса памяти.*

- `*` – операция «*взять содержимое*» – позволяет получить значение объекта по его адресу — определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;
- `&` – операция «*взять адрес*» – позволяет определить адрес переменной;
- Указатель, как и любая переменная, должен быть объявлен.
- ***Тип указателя*** *— это тип переменной, адрес которой он содержит.*

Пример:

```
char c;  // переменная
char *p; // указатель
p = &c;  // p = адрес c
```



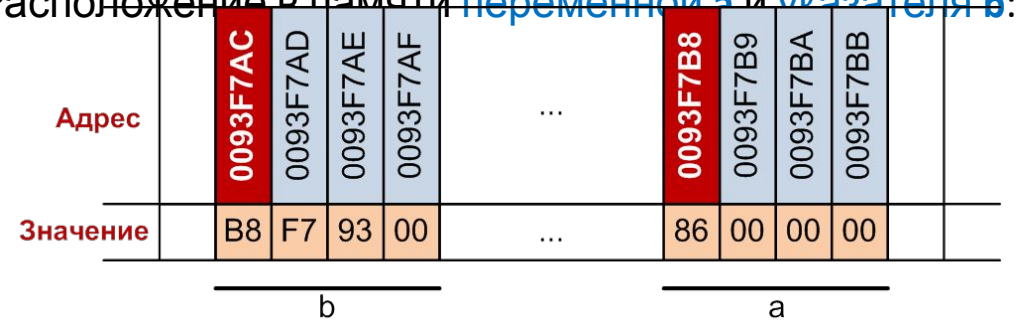
NB: ***в C++ есть ссылки, а в Си — нет***

Ссылка *— это тип переменной в языке C++, который работает как псевдоним другого объекта или значения*

```
#include <stdio.h>
int main()
{   int a, *b;
    a = 134;
    b = &a;
    // %x = вывод числа в шестнадцатеричной форме
    printf("\n Значение переменной a равно %d = %x шестн.", a, a);
    printf("\n Адрес переменной a равен %x шестн.", &a);
    printf("\n Данные по адресу указателя b равны %d = %x шестн.",
                *b, *b);
    printf("\n Значение указателя b равно %x шестн.", b);
    printf("\n Адрес расположения указателя b равен %x шестн.",
  &b);
    getchar();
    return 0;
}    // Резул
```

```
Значение переменной a равно 134 = 86 шестн.
Адрес переменной a равен 93f7b8 шестн.
Данные по адресу указателя b равны 134 = 86 шестн.
Значение указателя b равно 93f7b8 шестн.
Адрес расположения указателя b равен 93f7ac шестн.
```

Расположение в памяти переменной a и указателя b:



Указатели

Пример:
Записать по указанному адресу указанное значение (без использования переменных!!!):

```
*((int*)0x8000)=1;      /* Представили адрес как указатель и записали значение по этому адресу. В 4 байта, начиная с адреса 0x8000, будет записано значение 1      */
```

Оператор адреса &

При выполнении инициализации переменной, ей автоматически присваивается свободный адрес памяти, и, любое значение, которое мы присваиваем переменной, сохраняется по этому адресу в памяти.

Например: `int b = 8;`

При выполнении этой инструкции ЦП (CPU), выделяется часть оперативной памяти.

- В качестве примера предположим, что переменной **b** присваивается ячейка памяти под номером **150**. Всякий раз, когда программа встречает переменную **b** в выражении или в инструкции, она понимает, что для того, чтобы получить значение — ей нужно заглянуть в ячейку памяти под номером **150**.
- Хорошо, что нам не нужно беспокоиться о том, какие конкретно адреса памяти выделены для определенных переменных.
- Мы просто ссылаемся на переменную через присвоенный ей идентификатор, а компилятор конвертирует это имя в соответствующий адрес памяти.
- Однако этот подход имеет некоторые ограничения, которые мы обсудим ниже.
- Оператор взятия адреса **&** позволяет узнать, какой адрес памяти присвоен определенной переменной.

```
#include <stdio.h>
int main()
{
   int a = 7;
   printf("\n a = %d", a);
   printf("\n &a = %X", &a);
   getchar();
   return 0;
}
```

Всё довольно просто:

Результат на экране компьютера:

a = 7
&a = 0046FCF0

Операция **&** применима только к объектам, имеющим **имя** и размещенным в памяти. Ее нельзя применять к выражениям, константам-литералам, битовым полям структур.

```
char ch='G';          // 1 байт
int date=1937;        // 2 байта – для старых CPU
float summa=2.015E-6; // 4 байта
```

В этом примере (для старых CPU) переменная ch занимает **1 байт**, date - **2 байта** и summa - **4 байта**. В современных 32-разр. ПК переменная типа int может занимать **4 байта**, а переменная типа float - **8 байтов**.

В соответствии с приведенной таблицей переменные размещены в памяти, начиная с байта, имеющего шестнадцатеричный адрес:

Машинный адрес:	1A2B	1A2C	1A2D	1A2E	1A2F	1A30	1A31	1A32
	байт	байт	байт	байт	байт	байт	байт	байт
Значение в памяти:	'G'	1937		$2.015*10^{-6}$				
Имя:	**ch**	**date**		**summa**				

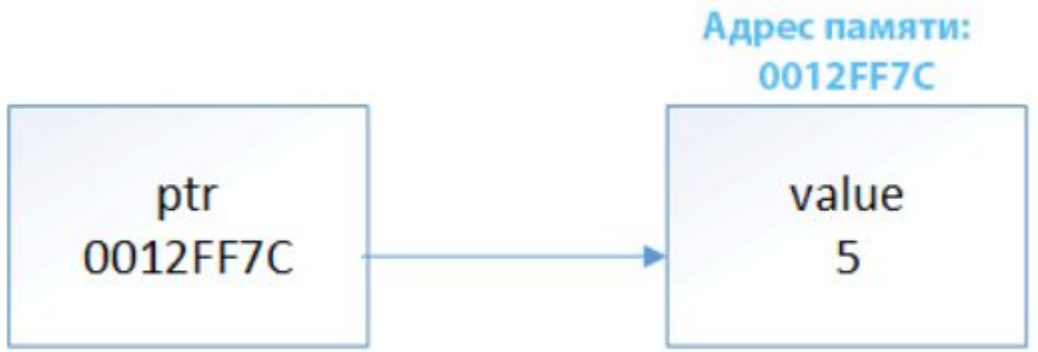
Указатели

- Имея возможность с помощью операции **&** определять ***адрес переменной*** или другого объекта программы, нужно уметь его **сохранять, преобразовывать и передавать**.
- Именно для этих целей в языке **Си** введены переменные типа «***указатель***».
- Кроме того, **значением указателя** может быть заведомо не равное никакому адресу значение, принимаемое за нулевой адрес.
- Для его обозначения в ряде заголовочных файлов, например в файле **stdio.h**, определена **специальная конст**

```
int *px;      /*указатель*/
px = NULL; /*присвоить NULL*/
```

- Помимо адресов, указатель может принимать специальное значение **NULL**, обозначающее ***недействительный адрес***
- **NULL** – макроконстанта
- **NULL** чаще всего (но не всегда!) равен **0**
- Разадресовывать указатель со значением **NULL** **небезопасно**!

```
int value = 5;
int *ptr = &value; /* инициализируем ptr адресом значения переменной */
```



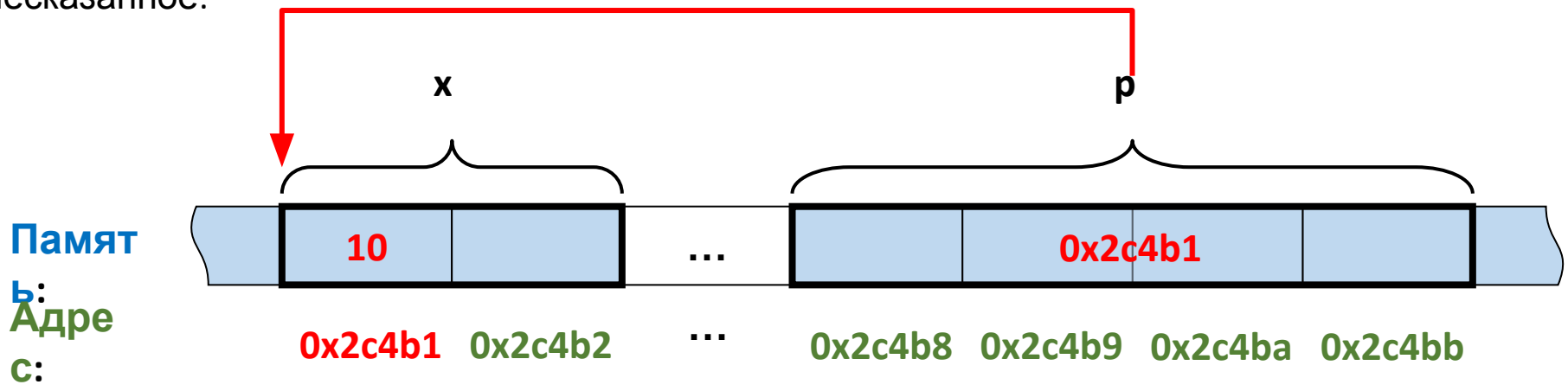
Размер указателей

Размер указателя зависит от архитектуры, на которой скомпилирован исполняемый файл:

- 32-битный исполняемый файл использует 32-битные адреса памяти
- следовательно, указатель на 32-битном устройстве занимает 32 бита (4 байта)
- с 64-битным исполняемым файлом указатель будет занимать 64 бита (8 байт)
- и это вне зависимости от того, на что указывает указатель

Указатели

Обобщим, всё вышесказанное:



Указатель – переменная, хранящая адрес

Операция разадресации * – обратная к операции &

```
int x;      /* целая переменная*/
int *px;    /* указатель        */

px = &x; /* присвоить адрес   */
```

```
int x=10, y;
int *px;

px = &x; /* взять адрес */
y = *px;  /* взять значение по
          адресу px, y=10   */
*px = 20; /* <=> x=20    */
```

2. Операции над указателями

Операции над указателями

В языке **Си** допустимы следующие (основные) операции над указателями:

- присваивание;
- получение значения того объекта, на который ссылается указатель (*синонимы: косвенная адресация, разыменование, раскрытие ссылки*);
- получение адреса самого указателя;
- унарные операции изменения значения указателя;
- аддитивные операции и операции сравнений.

Арифметические операции и указатели.

- Унарные адресные операции '&' и '*' имеют более высокий приоритет, чем арифметические операции.

Рассмотрим следующий пример, иллюстрирующий это правило:

```
float a=4.0, *u, z;
u=&z;
*u=5;
a=a + *u + 1; // a равно 10; u - не изменилось;  z равно 5
```

При использовании адресной операции '*' в арифметических выражениях следует остерегаться случайного сочетания знаков операций деления '/' и разыменования '*', так как комбинацию '/*' компилятор воспринимает как начало **комментария**.

Например, выражение a/*u следует заменить таким: a/(*u)

Унарные операции '*' и '++' или '--' имеют одинаковый приоритет и при размещении рядом выполняются справа-налево.

Рассмотрим операции над указателями подробнее.

- Указатель может быть инициализирован:

```
int y, *px=NULL, *py=&y, *pz=py; // инициализация
```

- Указателю можно присваивать значение:

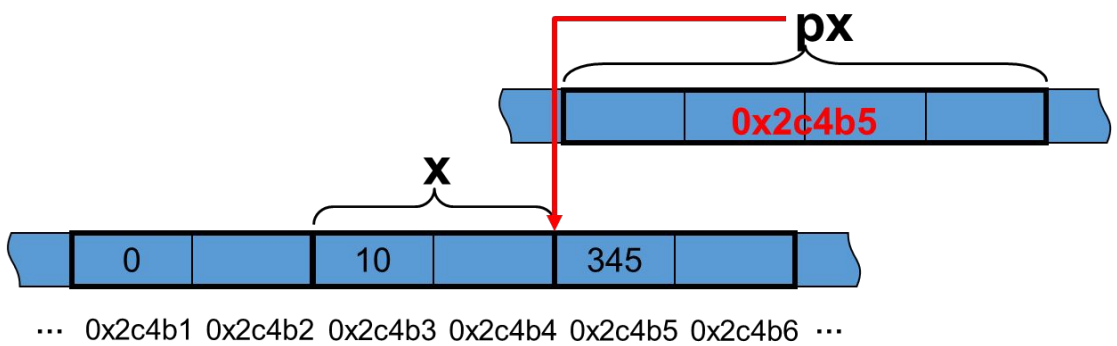
```
int x=10, y=20, *px, *py;
px=&x; py=px;
```

- Указатель можно сравнивать: < > <= >= == != (т.е. вычислять отношения адресов)

```
int x=10, y=20, *px=&x, *py=&y;
if( px == py ) ...
```

- Указатель может складываться с целым числом **N**. Результат сложения – **адрес**, смещенный на **N** компонент соответствующего типа относительно исходного:

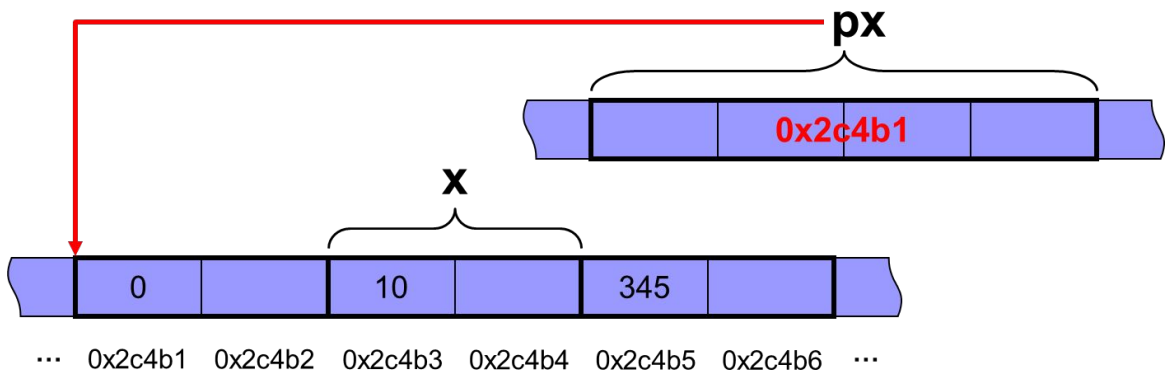
```
short x=10, *px=&x; // инициализация
px=px+1;
```



Операции над указателями

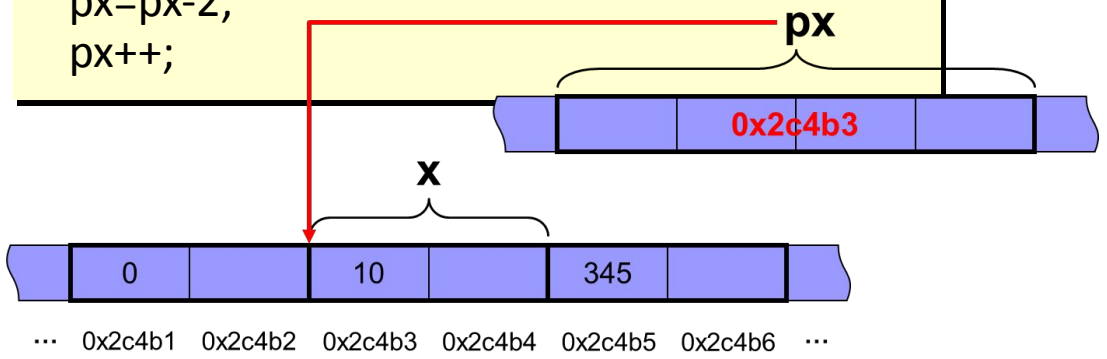
- Указатель может складываться с целым числом **N**. Результат сложения – **адрес**, смещенный на **N** компонент соответствующего типа относительно исходного:

```
short x=10, *px=&x; // инициализация
px=px+1;
px=px-2;
```

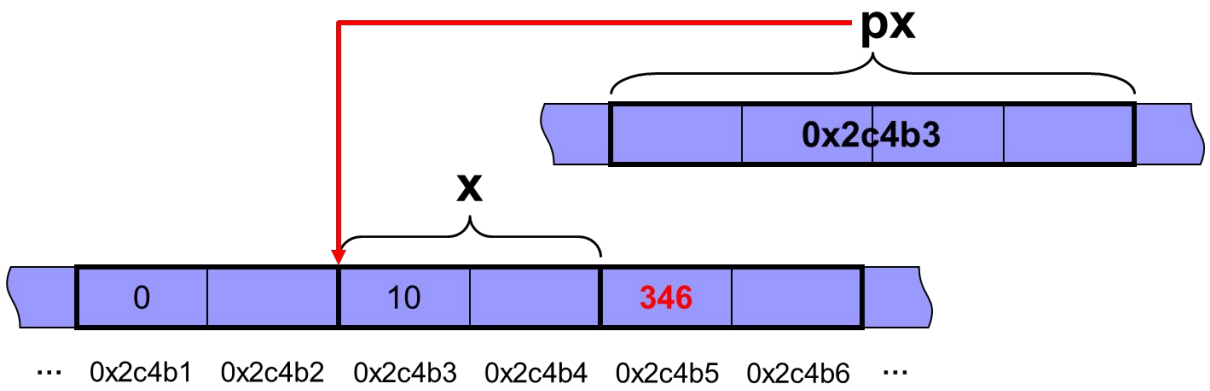


- Указатель может складываться с целым числом **N**. Результат сложения – **адрес**, смещенный на **N** компонент соответствующего типа относительно исходного:

```
short x=10, *px=&x; // инициализация
px=px+1;
px=px-2;
px++;
```

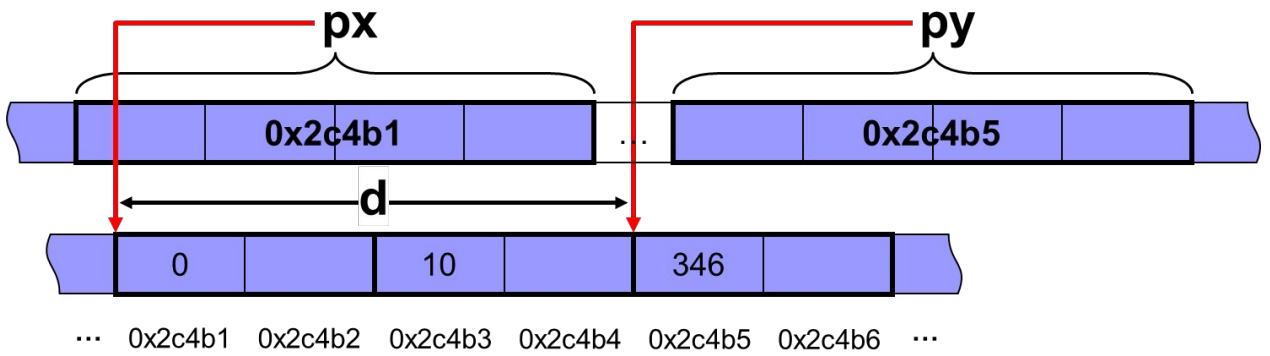



```
short x=10, *px=&x; // инициализация
px=px+1;
px=px-2;
px++; *(px+1)+=1;
```



- Можно вычислять разность однотипных указателей, которая равна относительному смещению с учетом типа указателя:

```
short *px=0x2c4b1, *py=0x2c4b5, d;
d = py-px; // 2
```



Операции над указателями

Приведем пример, в котором используются операции над указателями и выводятся (печатаются) получаемые значения.

Обратите внимание, что для **вывода значений указателей (адресов)** в форматной строке функции **printf( )** используется спецификация преобразования **%p**.

```
#include <stdio.h>
float x[ ] = { 10.0, 20.0, 30.0, 40.0, 50.0 };
void main( )
{
  float *u1, *u2;
  int i;
  printf("\n Адреса указателей: &u1=%p &u2=%p", &u1, &u2 );
  printf("\n Адреса элементов массива: \n");
  for(i=0; i<5; i++)
  {
    if (i==3) printf("\n");
    printf(" &x[%d] = %p", i, &x[i]);
  }
  printf("\n Значения элементов массива: \n");
  for(i=0; i<5; i++)
  {
    if (i==3) printf("\n");
    printf(" x[%d] = %5.1f ", i, x[i]);
  }
  for(u1=&x[0], u2=&x[4]; u2>=&x[0]; u1++, u2--)
  {
    printf("\n u1=%p *u1=%5.1f  u2=%p *u2=%5.1f",u1,*u1,u2,*u2);
    printf("\n u2-u1=%d", u2-u1);
  }
}
```

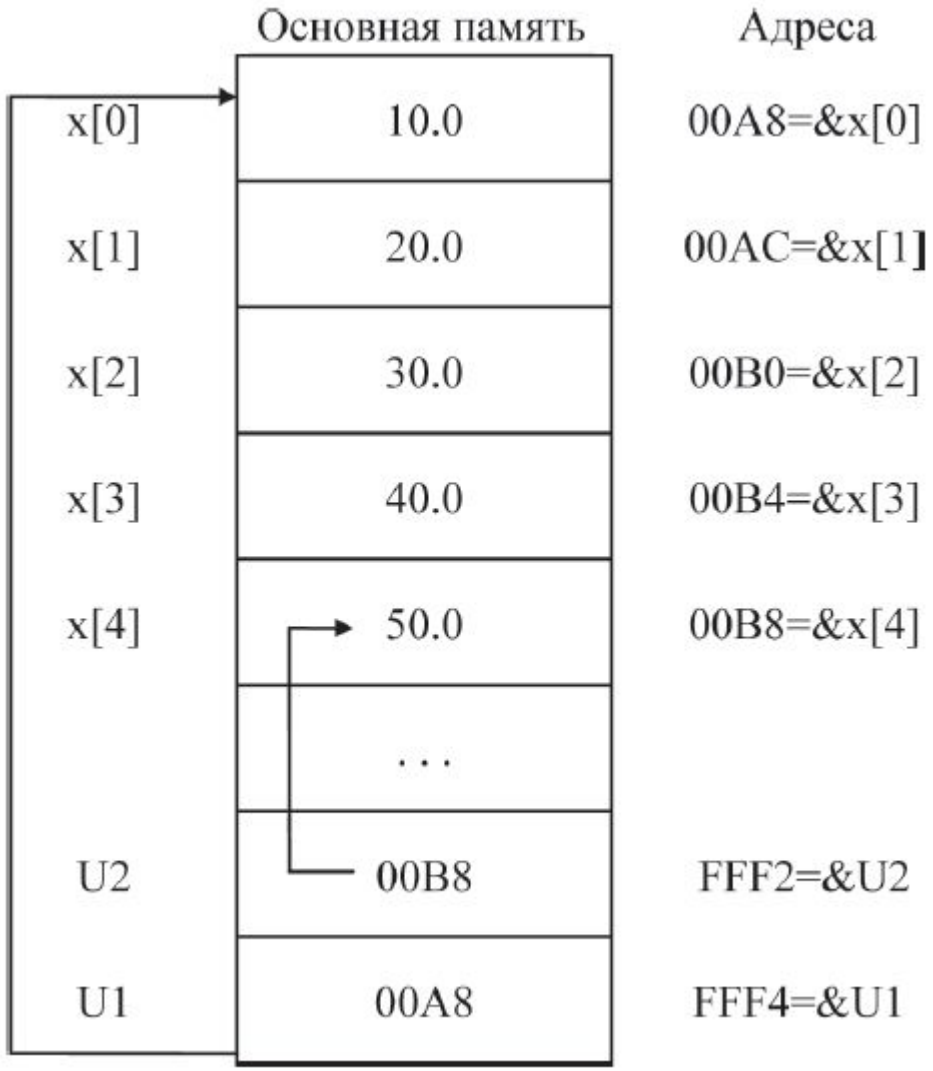
При печати значений разностей указателей и адресов в функции printf( ) использована спецификация преобразования **%d** - вывод знакового десятичного целого.

Возможный результат выполнения программы (конкретные значения адресов могут быть другими):

```
Адреса указателей: &u1=FFF4 &u2=FFF2
Адреса элементов массива:
&x[0]=00A8  &x[1]=00AC    &x[2]=00B0
&x[3]=00B4  &x[4]=00B8
Значения элементов массива:
x[0]=10.0     x[1]=20.0 x[2]=30.0
x[3]=40.0     x[4]=50.0
u1=00A8     *u1=10.0 u2=00B8  *u2=50.0
u2-u1=4
u1=00AC     *u1=20.0 u2=00B4  *u2=40.0
u2-u1=2
u1=00B0     *u1=30.0 u2=00B0  *u2=30.0
u2-u1=0
u1=00B4     *u1=40.0 u2=00AC  *u2=20.0
u2-u1=-2
u1=00B8     *u1=50.0 u2=00A8  *u2=10.0
u2-u1=-4
```

Операции над указателями

На рисунке приводится **схема размещения в памяти массива** **float x[5]** и **указателей** до начала выполнения цикла изменения указателей



Иногда требуется присвоить указателю одного типа значение указателя (адрес объекта) другого типа.

В этом случае используется «**приведение типов**», механизм которого понятен из следующего примера:

```
char *z;            // z - указатель на символ
int *k;          // k - указатель на целое
z=(char *)k;  // Преобразование указателей
```

Подобно любым переменным, *переменная типа указатель имеет имя, собственный адрес в памяти и значение.*

Значение можно использовать, например печатать или присваивать другому указателю, как это сделано в рассмотренных примерах.

Адрес указателя может быть получен с помощью унарной операции &.

Выражение &имя_указателя определяет, где в памяти размещен указатель.

Содержимое этого участка памяти является **значением указателя**. ***Соотношение между именем, адресом и значением указателя иллюстрирует схема ниже:***

Операции над указателями

- Как рассматривалось выше, унарные операции '*' и '++' или '--' имеют одинаковый приоритет и при размещении рядом выполняются **справа-налево**.
- Добавление целочисленного значения **n** к указателю, адресующему некоторый элемент массива, приводит к тому, что указатель получает значение адреса того элемента, который отстоит от текущего на **n** позиций (элементов).
- Если длина элемента массива равна **d** байтов, то численное значение указателя изменяется на (**d*n**).
- Рассмотрим следующий фрагмент программы,

```
int x[4]={ 0, 2, 4, 6 }, *i, y;
i=&x[0];  /* i равно адресу элемента x[0]*/
y=*i;     /* y равно 0;   i равно &x[0]  */
y=*i++;   /* y равно 0;   i равно &x[1]  */
y=++*i;   /* y равно 3;   i равно &x[1]  */
y=*++i;   /* y равно 4;   i равно &x[2]  */
y=(*i)++; /* y равно 4;   i равно &x[2]  */
y=++(*i); /* y равно 6;   i равно &x[2]  */
```

Название	Знак	Пояснения
Присваивание	=	Присвоить указателю адрес переменной или 0.
Инкремент	++	Увеличить указатель на 1. После увеличения указатель будет указывать на следующий элемент массива.
Декремент	--	Уменьшить указатель на 1. После уменьшения указатель будет указывать на предыдущий элемент массива.
Сложение	+	Увеличить указатель на целое значение.
Сложение с замещением	+=	Увеличить существующий указатель на целое значение.
Вычитание	–	Уменьшить указатель на целое значение или вычесть другой указатель, если оба указателя указывают на элементы одного и того же массива.
Вычитание с замещением	–=	Уменьшить существующий указатель на целое значение или вычесть другой указатель, если оба указателя указывают на элементы одного и того же массива.
Отношения	== != > < >= <=	Сравнить указатели. Возвращается истина, если операция сравнения закончилась успешно, иначе возвращается ложь. Сравниваемые указатели должны быть каким-то образом связаны, иначе операция не имеет смысла.

3. Нетипизированный указатель

- Отметим, что в **Си** можно работать с адресами, не уточняя, на переменные какого типа эти адреса будут указывать.
- Соответствующий тип указателя называется void*; если описать указатель такого типа:

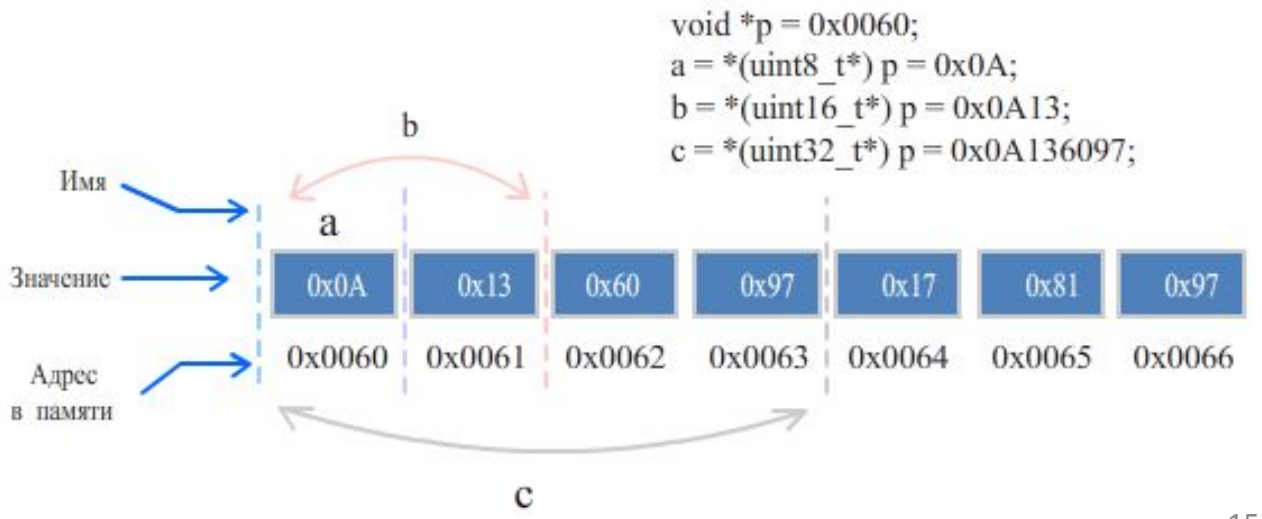
  void *z;

  то в переменную z можно будет занести совершенно любой адрес, и такое присваивание компилятор рассматривает как легитимное, не выдавая ни ошибок, ни предупреждений.
- Более того, разрешено также и присваивание в другую сторону, то есть **любому типизированному указателю** можно присвоить **нетипизированный адрес**.
- Интересно, что значение адреса можно использовать в качестве логического значения везде, где таковое требуется, в том числе в заголовках операторов ветвления и циклов;
  при этом «нулевой указатель» (то есть значение **NULL**) считается «**ложью**», а любой другой адрес — «**истиной**».
- **Типизированные указатели (int *, char *, double *, …)** неявно задают длину фрагмента памяти (**4,1,8, …** байт), начинающегося с адреса, хранимого указателем
- Длина важна при **разадресации** и **адресной арифметике**
- Однако иногда приходится использовать **указатели**, **не подразумевая длины адресуемого фрагмента памяти** –

  void *
- *Разадресация указателя* void * *невозможна*!
- Указатель void * совместим по типу со всеми типизированными указателями

```
int i=10, *pi=&i;
double d=3.14, *pd=&d;
void *p;
p=pi;          /* Ok */
p=pd;          /* Ok */
*p=*p+1;       /* Ошибка! */
```

- Указывая **тип указателя**, мы говорим компилятору, *вот тебе адрес начала массива, один элемент массива занимает 2 байта, таких элементов в массиве 10.* Итого сколько памяти выделить под этот массив?
  20 байт — отвечает компилятор.
  Для наглядности возьмите указатель типа void, для него не определено сколько места он занимает — это ***просто адрес***.



4. Указатели и const

- Кроме переменных в программе на **Си** для хранения данных могут использоваться **константы**, которые предваряются ключевым словом **const**, и указатели также могут указывать на **константы**.
- Два способа описания константного указателя:

✔ **Неизменяемый указатель**

- Синтаксис: TYPE *const ptrName = &aTYPEVar;
- **Переменная-указатель** – константа (**не может изменяться**)
- **Данные**, адресуемые указателем – **изменяемые**

Мы можем определять **константные указатели**.

Они не могут изменять адрес, который в них хранится, но могут изменять значение по этому адресу:

```
int a=42, b=42;
Int *const ptr=&a;
*ptr=1; /* Ok */
ptr=&b /* Ошибка! Это всё равно, что 5=x;  */
```

```
int a = 10;
int *const pa = &a;
printf("value=%d \n", *pa);     // 10
*pa = 22;                       // меняем значение
printf("value=%d \n", *pa);     // 22
int b = 45;
            // pa = &b;          так нельзя сделать
```

✔ Через **указатель на константу** мы ***не можем изменять значение переменной/константы.***
Но ***мы можем присвоить указателю адрес любой другой переменной или константы***:

```
int a = 10;
/* указатель указывает на переменную a
*/
const int *pa = &a;
const int b = 45;
/* указатель указывает на константу b */
pa = &b;
```

✔ И объединение обоих предыдущих случаев - **константный указатель на константу**, который не позволяет менять ни хранимый в нем адрес, ни значение по этому адресу:

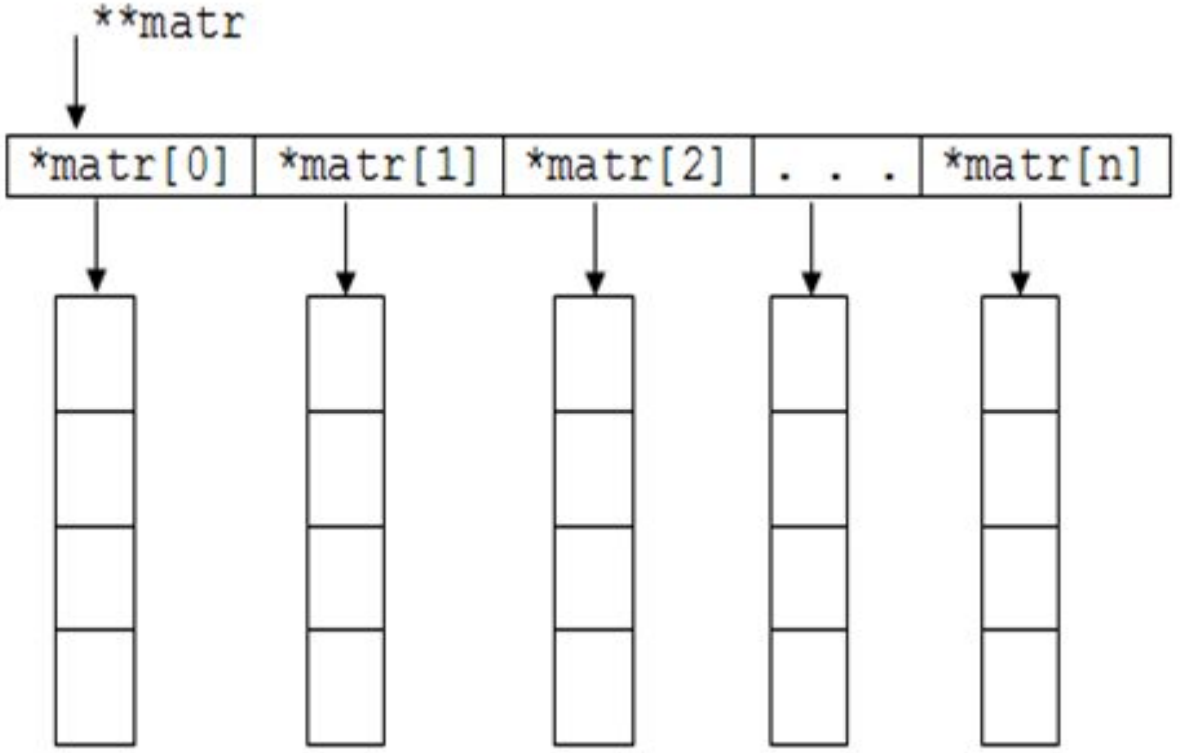
```
int a = 10;
const int *const pa = &a;
//*pa = 22;  так сделать нельзя
int b = 45;
// pa = &b;  так сделать нельзя
```

5. Указатель на указатель

Можно создать *указатель на указатель*, тогда он будет хранить адрес указателя и сможет обращаться к его содержимому. Указатель на указатель определяется как:

<тип> **<имя>;

- Ничто не мешает создать и *указатель на указатель на указатель*, и *указатель на указатель на указатель на указатель* и так далее.
- Это нам понадобится в дальнейшем при работе с *двумерными* и *многомерными массивами*.




```c
/* Простой пример, как можно работать с указателем на указатель */
#include <stdio.h>
#define SIZE 10
void main()
{  int A;
   int B;
   int *p;
   int **pp;
   A = 10;
   B = 111;
   p = &A;
   pp = &p;

   printf("A = %d\n", A);
   *p = 20;
   printf("A = %d\n", A);
   *(*pp) = 30;    //здесь скобки можно не писать
   printf("A = %d\n", A);

   *pp = &B;
   printf("B = %d\n", B);
   **pp = 333;
   printf("B = %d", B);
   getch();
}
```

```
A = 10
A = 20
A = 30
B = 111
B = 333
```

6. Указатель файла

Указатель файла — это то, что соединяет в единое целое всю систему ввода-вывода языка **Си**.

❖ *Указатель файла — это указатель на структуру типа **FILE**.*

- ❑ Он указывает на структуру, содержащую различные сведения о файле, например, его имя, статус и указатель текущей позиции в начало файла.
- ❑ В сущности, указатель файла определяет конкретный файл и используется соответствующим потоком при выполнении функций ввода/вывода.
- ❑ Чтобы выполнять в файлах операции чтения и записи, программы должны использовать указатели соответствующих файлов.
- ❑ Чтобы объявить переменную-указатель файла, используйте такого рода оператор:

```
FILE *fp;
```

Файлы в Си используются для того, чтобы сохранять результат работы программы Си и использовать его при новом запуске программы .

Например можно сохранять результаты вычислений , статистику игр.

Чтобы работать с файлами в Си необходимо подключить библиотеку **stdio.h**

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

Чтобы работать с файлом в си необходимо задать указатель на файл по образцу:

```
FILE *имя указателя на файл;
```

Например

```
FILE *fin;
```

Задает указатель fin на файл

Дальше необходимо открыть файл и привязать его к файловому указателю.

Для открытия файла в Си на чтение используется команда

```
Имя указателя на файл= fopen("путь к файлу", "r");
```

❖ ***Файл** (file) — блок информации на внешнем запоминающем устройстве компьютера, имеющий определенное логическое представление (начиная от простой последовательности битов или байтов и заканчивая объектом сложной СУБД), соответствующие ему операции чтения-записи и, как правило, фиксированное имя (символьное или числовое), позволяющее получить доступ к этому файлу и отличить его от других файлов*

Работа с файлами реализуется средствами операционных систем (ОС).

Файл характеризуется набором параметров: именем, размером, датой создания, датой последней модификации и атрибутами, которые используются операционной системой для его обработки: является ли файл системным, скрытым или предназначен только для чтения.

❖ ***Файл** (по ГОСТ 20886-85) — идентифицированная совокупность экземпляров полностью описанного в конкретной программе типа данных, находящихся вне программы во внешней памяти и доступных программе посредством специальных операций*

7. Указатели и массивы

Указатели и массивы очень тесно связаны в языке Си

- ***Имя массива*** *– константный указатель на 0-й элемент массива*

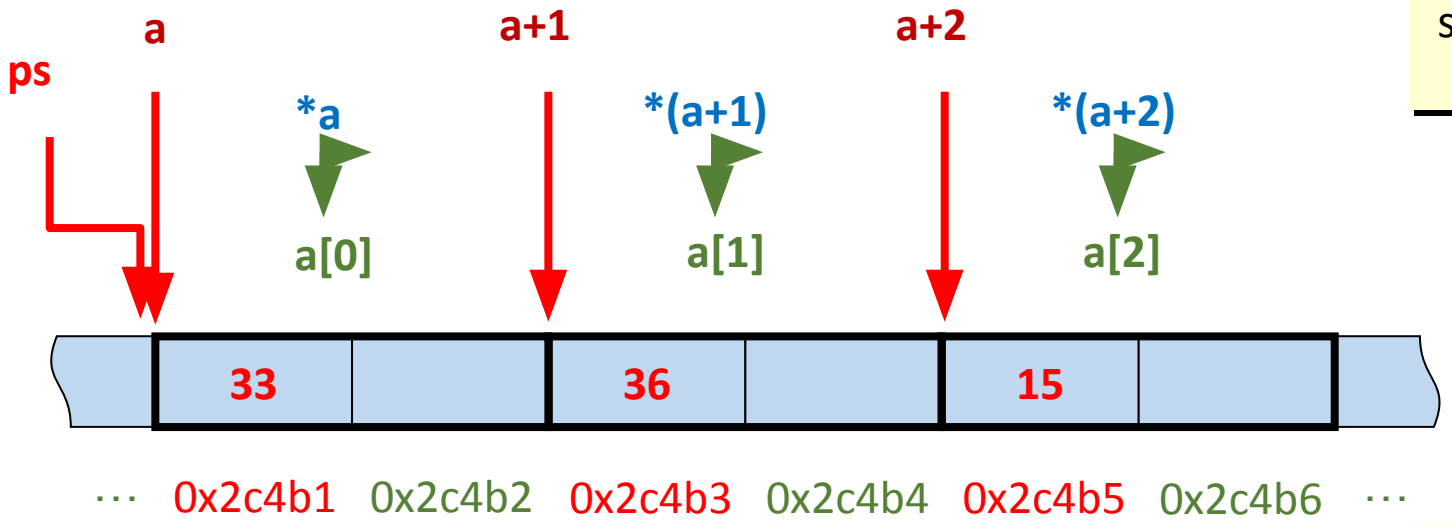
```
short a[100];
short *ps;

ps = &a[0];
```

⟺

```
short a[100];
short *ps;

ps = a;
```



a[i] == *(a+i) == *(i+a) == i[a]

a[2] == *(a+2) == *(2+a) == 2[a]

```
short a[100];
for (i=0; i<100; i++)
 scanf("%h", &a[i]);
```

⟺

```
short a[100];
for (i=0; i<100; i++)
   scanf("%h", a+i);
```

```
short a[100];
short *ps;
for (ps=a; ps==a+100; ps++)
      scanf("%h", ps);
```

```
short a[100];
short *ps=a;
for (i=0; i<100; i++)
      scanf("%h", &ps[i]);
```

Указатели и массивы

Синонимичные выражения:

```
pa = &a[0];     ⇔    pa = a;
&a[i]           ⇔    a+i
pa[i]           ⇔    *(pa+i)
char s[]        ⇔    char* s
```

Передача массива в функцию как параметра:

```
int a[10];

f(a);    /* или */
f(&a[0]);
```

```
void f(int *array)
{
  ...
}

/* или */
void f(int array[])
{
  ...
}
```

Указатели на многомерные массивы

- Для вычисления адреса элемента двумерного массива компилятору нужно «знать» количество столбцов в матрице (т.е. мало знать начальный адрес массива)
- Пусть нужно передать в функцию массив **int array[3][15]**, чтобы ее вызов выглядел так: **f(array)**
- Возможны следующие идентичные варианты описания функции **f:**

```
f(int x[3][15]) { … }
f(int x[][15])  { … }
f(int (*x)[15]) { … }
```

Важно: *в последнем случае нельзя опустить скобки!*

```
f(int *x[15]) { … } /* передается массив из 15 указателей на int, а не указатель на массив из 15 int-ов */
```

8. Указатели на функции

❖ ***Указатель на функцию*** *содержит адрес тела функции*

Описание указателя на функцию:

```
returnType   (*functPtr)(paramType1, paramType2)
```

- Тип возвращаемого значения функции
- Указатель на функцию
- Тип параметра функции
- Тип параметра функции

```
float myfun(int a, float b)
{
  return a+b;
}
...
float (*fptr)(int, float);
fptr = myfun;
...
x=fptr(42, 3.14f);
...
```

Указатель на функцию, воспринимающую параметры типов int и float и возвращающую float

Вызов функции по указателю

Пример (фрагмент):

```
int add(int x, int y)    { return x+y; }
int sub(int x, int y)    { return x-y;  }
int mul(int x, int y)    { return x*y; }
int div (int x, int y)   { return x/y;  }

int evaluate(unsigned int op, int x, int y)
{
  int (*eval[])(int, int) = { add, sub, mul, div };

  if (op>3)
  {
    printf("Недопустимая операция");
    return 0;
   }

  return eval[op](x, y);
}

void main()
{
  printf("%d\n", evaluate(3, 42, 3));
}
```

Операции пронумерованы:
0 – add, 1 – sub, 2 – mul, 3 – div

Массив из указателей на функции вида **int f(int, int)**

Вызов подходящей функции

Указатели в параметрах функций

В функцию передается не значение, а **адрес переменной**:

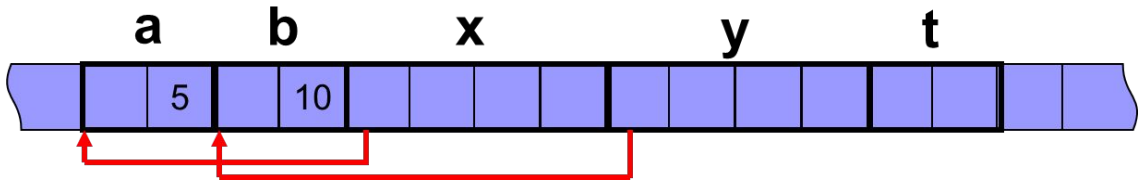
Для доступа к значению переменной используется **операция разадресации**

```
#include <stdio.h>
void swap(int *x,  int *y)
{
 int t;
 t = *x;    *x = *y;    *y = t;
}
void main()
{
  int a=5, b=10;
  swap(&a,  &b);
  printf("a=%d, b=%d\n", a, b);
}
```

При **описании параметров функции** используются **указатели**

При вызове функции **как параметр передается адрес переменной**

Как изменить переменную в вызывающей функции?




```
x = &a; /*в x – адрес a                    */
y = &b; /*в y – адрес a                    */
t = *x;  /* в t поместить значение,
          хранящееся по адресу x           */
*x = *y; / *по адресу x записать
значение,
          хранящееся по адресу y           */
*y = t; / *по адресу y записать значение,
          хранящееся в t                   */
```

Примеры сложных описаний с указателями

```
char **argv          // argv: указатель на указатель на char
int (*x)[13]         // x: указатель на массив из 13 int-ов
int *x[13]           // x: массив из 13 указателей на int
void *comp()         // comp: функция, возвращающая указатель на void
void (*comp)()       // comp: указатель на функцию, возвращающую void
char (*(*x())())[5]  /* x: функция, возвращающая указатель на массив из 5 указателей на функцию,
возвращающую
                                                                    char     */
char (*(*x[3])())[5]  // x: массив из 3 указателей на функцию, возвращающую указатель на массив из 5 char-ов
```

Плюсы указателей

1. Массивы реализованы с помощью указателей. Указатели могут использоваться для итерации по массиву (рассмотрим на следующих занятиях)

2. Они являются единственным способом динамического выделения памяти.
Это, безусловно, самый распространенный вариант использования указателей.

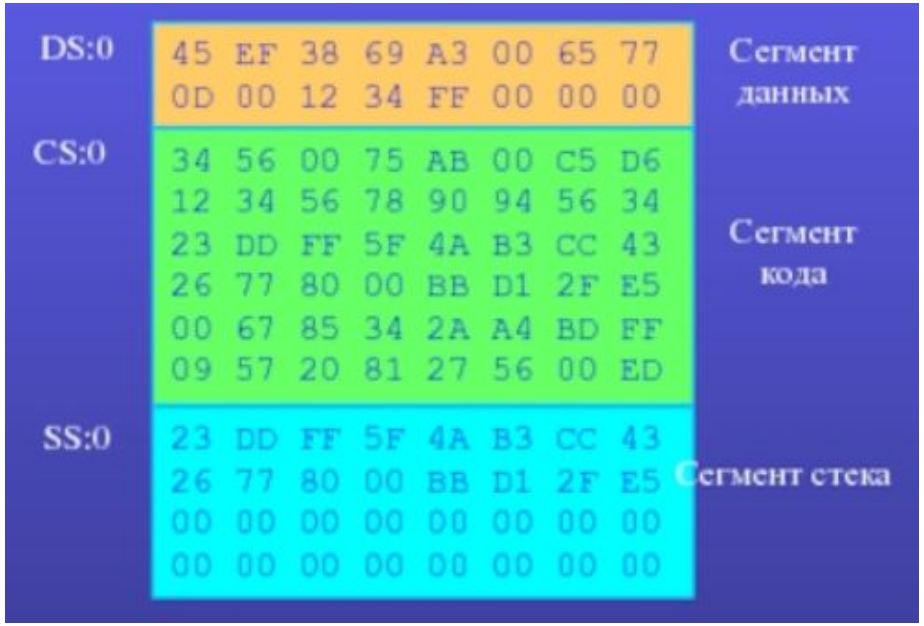
3. **Указатель** занимает 2-8 байт, а **объект** может занимать несколько Кбайт/Мбайт (и содержать указатели на другие объекты).
Объект может быть один, а указателей на него много.

4. Они могут использоваться для **передачи большого количества данных в функцию** *без копирования этих данных*.

5. Они могут использоваться для передачи одной функции в качестве параметра другой функции.

6. Вы можете обращаться к объекту по адресу, не зная его имени.
В указатель мы можем «подставлять» адреса самых разных объектов. И одна и таже функция сможет обработать эти объекты…

Упрощенная схема размещения исполняемого кода в ОП



Стек имеет *ограниченный размер* и, следовательно, может содержать только ограниченный объем информации.
В ОС **Windows** размер стека по умолчанию составляет **1МБ**.
На некоторых **Unix-системах** этот размер может достигать и **8МБ**. Если программа пытается поместить в стек слишком много информации, то это приведет к переполнению стека.
Переполнение стека («**stack overflow**») происходит, когда запрашиваемой памяти нет в наличии (вся память уже занята).

Переполнение стека является результатом добавления слишком большого количества переменных в стек и/или создания слишком большого количества вложенных вызовов функций (например, когда функция **A()** вызывает функцию **B()**, которая вызывает функцию **C()**, а та, в свою очередь, вызывает функцию **D()** и т.д.).
Переполнение стека обычно приводит к **сбою** в программе

Недостатки указателей

- Использование указателей синтаксически загрязняет код и усложняет его понимание. (Приходится использовать операторы * и &.)
- Указатели могут быть неинициализированными (некорректный код).
- Указатель может быть нулевым (корректный код), а значит указатель нужно проверять на равенство нулю.
- Арифметика указателей может сделать из корректного указателя некорректный (легко промахнуться).

оператор разыменования и взятия адреса

если объявлен указатель, но не проинициализирован, то там хранится какой-то адрес

например, выйти за границы массива

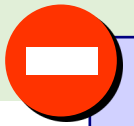
обращение к неинициализированному указателю – ошибка.

обращение к нулевому указателю – это ошибка.

Выводы

Что надо знать об указателях:

- *указатель* – это переменная, в которой можно хранить адрес другой переменной;
- при объявлении *указателя* надо указать тип переменных, на которых он будет указывать, а перед именем поставить знак ***;
- знак ***&*** перед *именем переменной* обозначает ее *адрес*;
- знак * перед *указателем* в рабочей части программы (не в объявлении) обозначает *значение ячейки*, на которую указывает указатель;
- для обозначения *недействительного указателя* используется *константа* ***NULL*** (нулевой указатель);
- при изменении значения *указателя* на ***n*** он в самом деле сдвигается к *n*-ому следующему числу данного типа, то есть для указателей на целые числа на ***n*sizeof(integer)** байт;
- указатели печатаются по формату ***%p***.



Нельзя использовать указатель, который указывает неизвестно куда (будет сбой или зависание)!

ЛИТЕРАТУРА

1. Демидович Е. Основы алгоритмизации и программирования. Язык Си: учебное пособие – СПб.: БХВ – Петербург, 2006. – 440с.
2. Жешке Р. Толковый словарь стандарта языка Си. – СПб.: Питер, 1994. – 221с.
3. Керниган Б., Ритчи Д. Язык программирования Си: Пер. с англ. – 2-е изд., перераб. и доп. – М.: Финансы и статистика, 1992. – 272с.
4. Кочан С. Программирование на языке С, 3-е издание: Пер. с англ. – М.: ООО “И. Д. Вильямс”, 2007. – 496с.
5. Подбельский В., Фомин С. Программирование на языке Си: учебное пособие. 2-е доп. Изд. – М: Финансы и статистика, 2001. – 2001. – 600с.
6. Прата С. Язык программирования С. Лекции и упражнения, 5-е издание.: Пер. с англ. – М.: Издательский дом “Вильямс”, 2006. – 960с.
7. Шилдт Г. Полный справочник по С. 4-е издание.: Пер. с англ. – М.: Издательский дом “Вильямс”, 2002. – 704с.
8. Харбисон С., Стил Г. Язык программирования С.: Пер. с англ. – М: ООО Бином Пресс, 2004. – 528с.