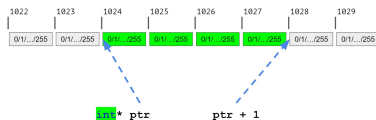


Массивы



Владимир Валерьевич Соловьёв
Huawei, НГУ
vladimir.conwor@gmail.com
t.me/conwor
vk.com/conwor

sizeof

Оператор, возвращающий размер операнда в байтах - целое число типа **size_t**

sizeof

Оператор, возвращающий размер операнда в байтах - целое число типа **size_t**

size_t, соответственно - целый беззнаковый тип, определённый в **stddef.h**, множество значений которого включает все возможные адреса

sizeof

Оператор, возвращающий размер операнда в байтах - целое число типа **size_t**

size_t, соответственно - целый беззнаковый тип, определённый в **stddef.h**, множество значений которого включает все возможные адреса

Это именно тот тип, который нужно использовать для преобразования указателей в числа и обратно

sizeof

```
int main() {  
    int x;  
    int* ptr;  
    printf("%zu %zu\n", sizeof(x), sizeof(ptr));  
}
```

sizeof

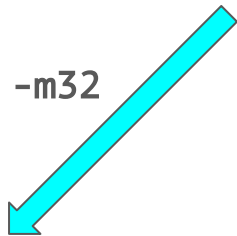
```
int main() {  
    int x;  
    int* ptr;  
    printf("%zu %zu\n", sizeof(x), sizeof(ptr));  
}
```

sizeof

```
int main() {  
    int x;  
    int* ptr;  
    printf("%zu %zu\n", sizeof(x), sizeof(ptr));  
}
```

gcc -m32

> 4 4

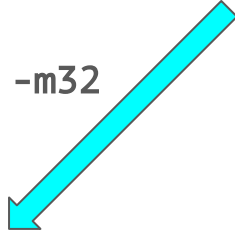


sizeof

```
int main() {  
    int x;  
    int* ptr;  
    printf("%zu %zu\n", sizeof(x), sizeof(ptr));  
}
```

gcc -m32

> 4 4

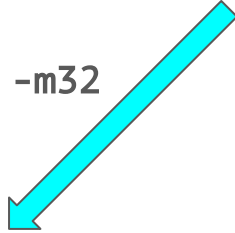


sizeof

```
int main() {  
    int x;  
    int* ptr;  
    printf("%zu %zu\n", sizeof(x), sizeof(ptr));  
}
```

gcc -m32

> 4 4



sizeof

```
int main() {  
    int x;  
    int* ptr;  
    printf("%zu %zu\n", sizeof(x), sizeof(ptr));  
}
```

gcc -m32

> 4 4

gcc -m64

> 4 8

sizeof

Напоминание: это ID, на другой реализации языка Си результаты могут отличаться

```
int main() {  
    int x;  
    int* ptr;  
    printf("%zu %zu\n", sizeof(x), sizeof(ptr));  
}
```

gcc -m32

> 4 4

gcc -m64

> 4 8

sizeof

Напоминание: это ID, на другой реализации языка Си результаты могут отличаться

```
int main() {  
    int x;  
    int* ptr;  
    printf("%zu %zu\n", sizeof(x), sizeof(ptr));  
}
```

gcc -m32

> 4 4

gcc -m64

> 4 8

Напоминание: все примеры в лекциях будут с использованием 32-битной системы

Адресная арифметика

Для указателей определены операции сложения и вычитания с целыми числами

Адресная арифметика

Для указателей определены операции сложения и вычитания с целыми числами

Результатом является указатель такого же типа, значение которого получается прибавлением или вычитанием аргумента, умноженного на размер типа, на который указывает указатель

То есть, `ptr + 1` указывает не на байт дальше, а на следующий элемент за элементом, на который указывает `ptr`

Адресная арифметика

```
int main() {  
    int* ptr = (int*) 1024;  
    ptr += 1;  
    printf("%p\n", ptr);  
}
```

Адресная арифметика

```
int main() {  
    int* ptr = (int*) 1024;  
    ptr += 1;  
    printf("%p\n", ptr);  
}
```

> 0x404

Адресная арифметика

```
int main() {  
    int* ptr = (int*) 1024;  
    ptr += 1;  
    printf("%p\n", ptr);  
}
```

Вывод адресов происходит в 16-ричной
системе счисления - это стандарт
системного программирования

> 0x404

Адресная арифметика

```
int main() {  
    int* ptr = (int*) 1024;  
    ptr += 1;  
    printf("%p\n", ptr);  
}
```

> 0x404

Вывод адресов происходит в 16-ричной
системе счисления - это стандарт
системного программирования

Чем раньше вы начнёте считать в ней,
тем лучше для вас

Адресная арифметика

```
int main() {  
    int* ptr = (int*) 1024;  
    ptr += 1;  
    printf("%p\n", ptr);  
}
```

> 0x404

Вывод адресов происходит в 16-ричной
системе счисления - это стандарт
системного программирования

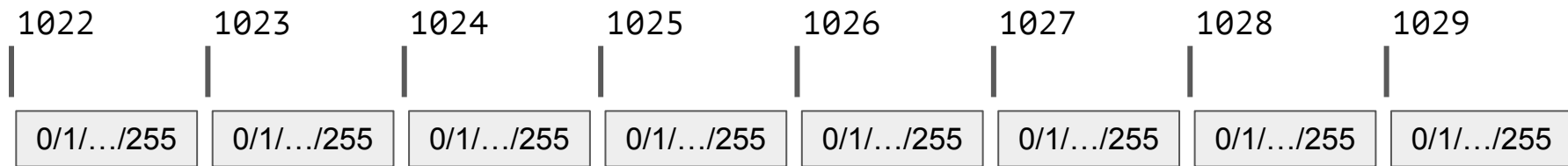
Чем раньше вы начнёте считать в ней,
тем лучше для вас

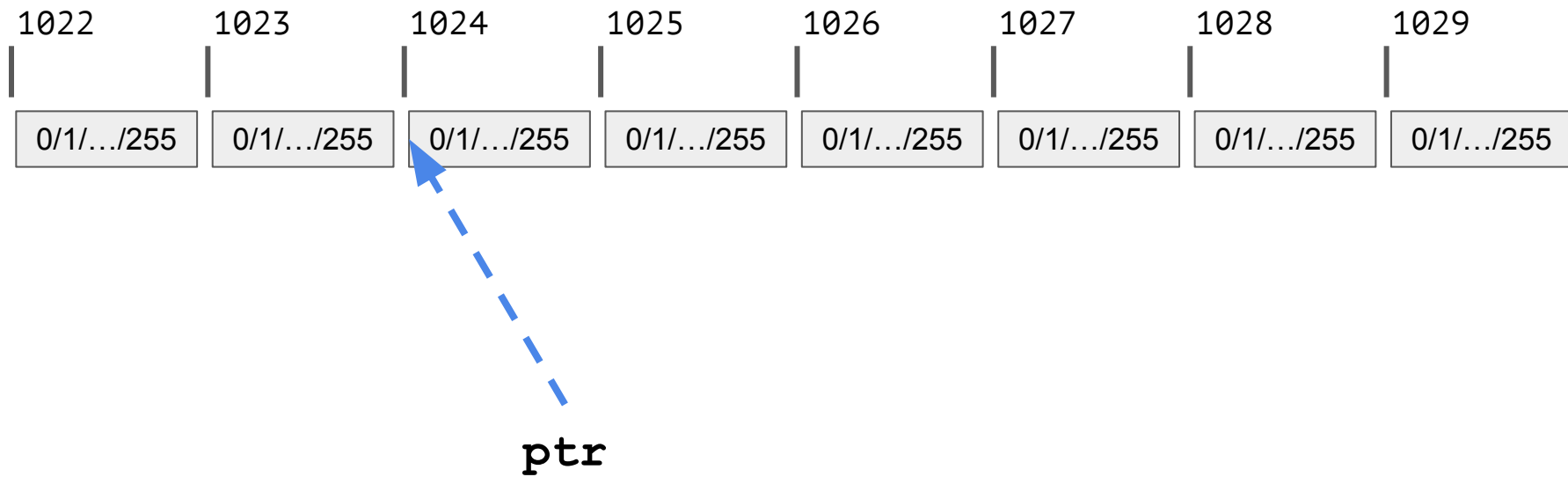
Но пока непривычно, можно пользоваться
`size_t`

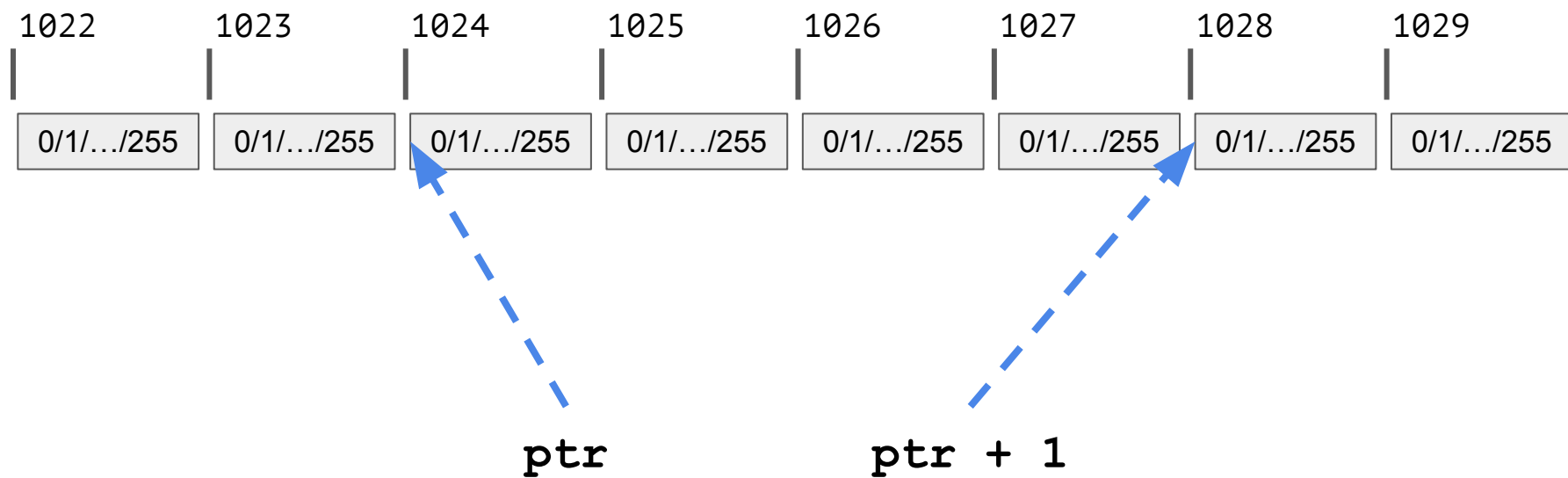
Адресная арифметика

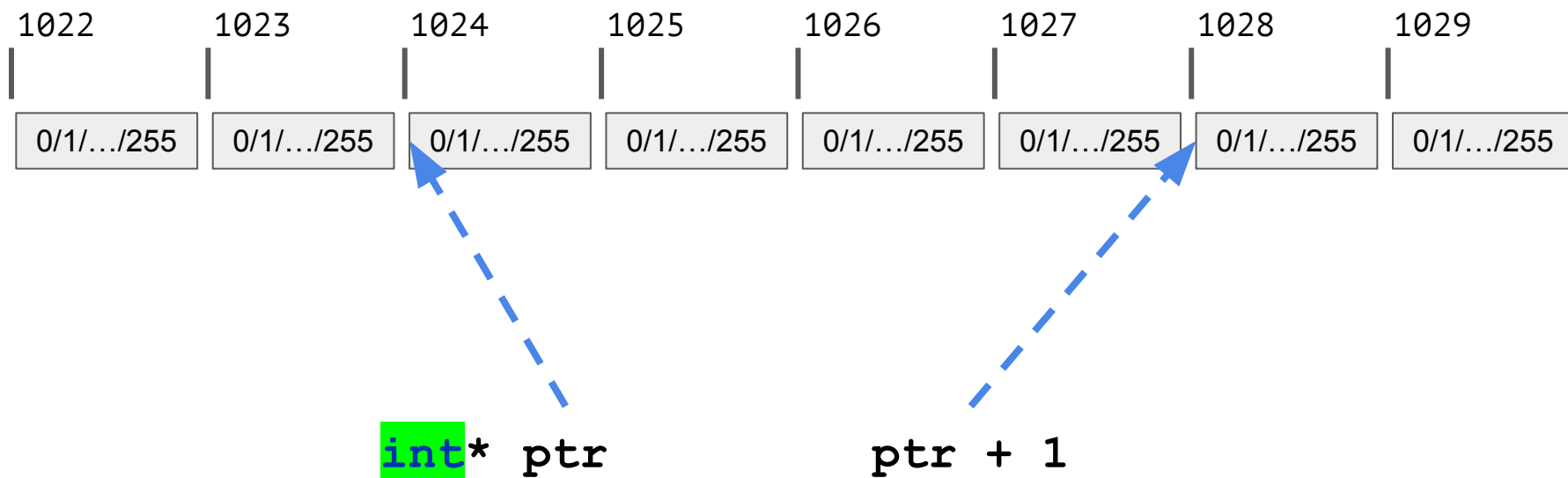
```
int main() {  
    int* ptr = (int*) 1024;  
    ptr += 1;  
    printf("%zu\n", (size_t) ptr);  
}
```

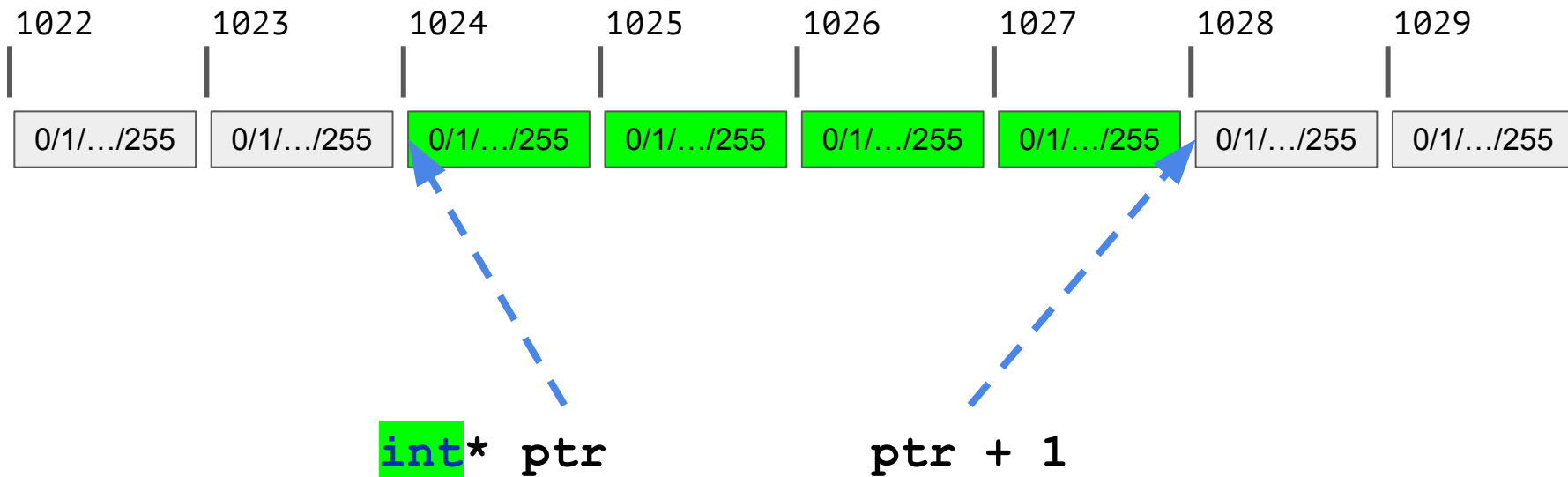
> 1028

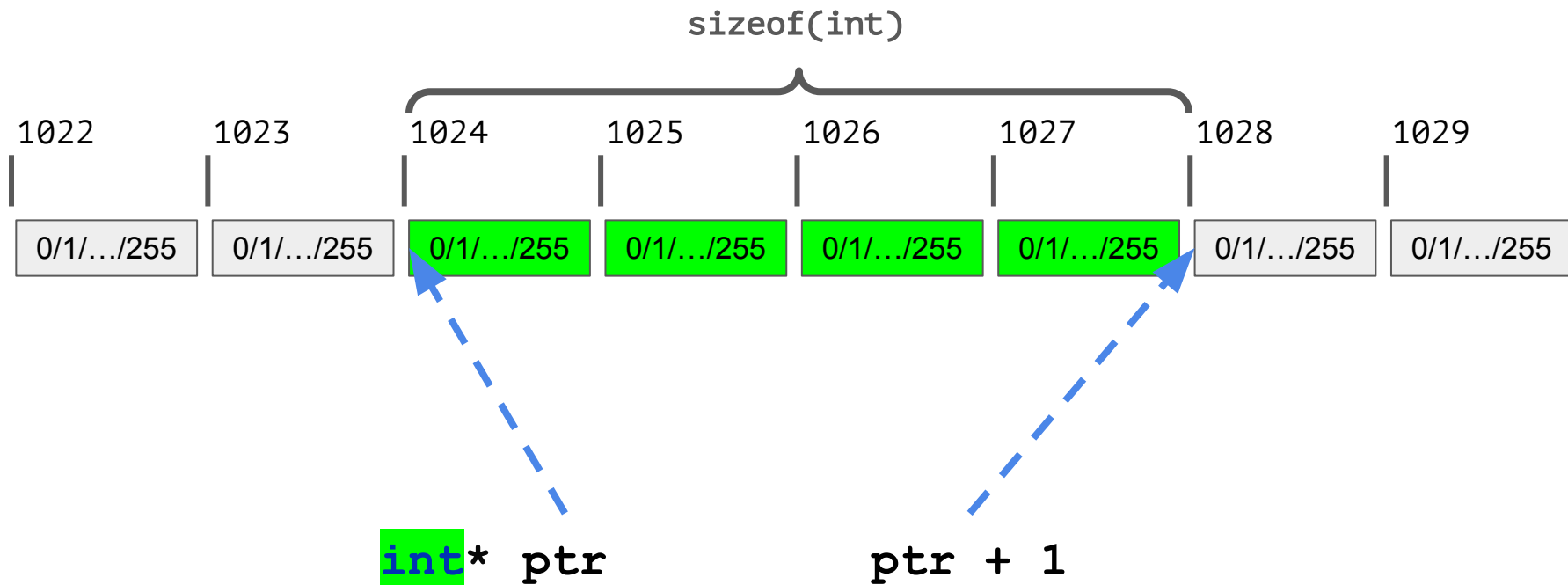












Адресная арифметика

Смысл адресной арифметики не в том, чтобы просто изменять значения адресов, как целых чисел, а в том, чтобы вычислять адреса следующих или предшествующих в памяти **таких же** элементов

Адресная арифметика

Смысл адресной арифметики не в том, чтобы просто изменять значения адресов, как целых чисел, а в том, чтобы вычислять адреса следующих или предшествующих в памяти **таких же** элементов

Если вам нужна манипуляция именно со значениями адресов, можно преобразовать указатель к типу **size_t**, который является обычным целым типом с обычной арифметикой, либо к типу указателя на **char**

`sizeof(char), sizeof(unsigned char), sizeof(signed char)` - всегда равны 1

Упражнение на адресную арифметику

A: 3

B: 4

A + B: ?

Упражнение на адресную арифметику

A: 3
B: 4
A + B: ?

	<code>int</code> A	<code>int*</code> A
<code>int</code> B	?	
<code>int*</code> B		

Упражнение на адресную арифметику

A: 3
B: 4
A + B: ?

	<code>int</code> A	<code>int*</code> A
<code>int</code> B	7	?
<code>int*</code> B		

Упражнение на адресную арифметику

A: 3
B: 4
A + B: ?

	<code>int</code> A	<code>int*</code> A
<code>int</code> B	7	19
<code>int*</code> B	?	

Упражнение на адресную арифметику

A: 3
B: 4
A + B: ?

	<code>int</code> A	<code>int*</code> A
<code>int</code> B	7	19
<code>int*</code> B	16	?

Упражнение на адресную арифметику

A: 3
B: 4
A + B: ?

	<code>int</code> A	<code>int*</code> A
<code>int</code> B	7	19
<code>int*</code> B	16	

Массив

Последовательность данных (*элементов* массива) одного типа, идентифицируемых по *индексу*

Массив

Последовательность данных (**элементов** массива) одного типа, идентифицируемых по **индексу**

С массивом всегда связана его **длина** - количество элементов

Массив

Последовательность данных (**элементов** массива) одного типа, идентифицируемых по **индексу**

С массивом всегда связана его **длина** - количество элементов

Полноценный тип данных - определён множеством значений и операциями, объекты могут быть заведены с любым типом хранения

Массив константной длины

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
    printf("%d\n", arr[0] + arr[4]);  
}
```

Массив константной длины

Определение массива с именем `arr`
длины 5

константной (compile-time known) ,
то есть известной в момент
компиляции программы

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
    printf("%d\n", arr[0] + arr[4]);  
}
```



Массив константной длины

Определение массива с именем `arr`
длины 5

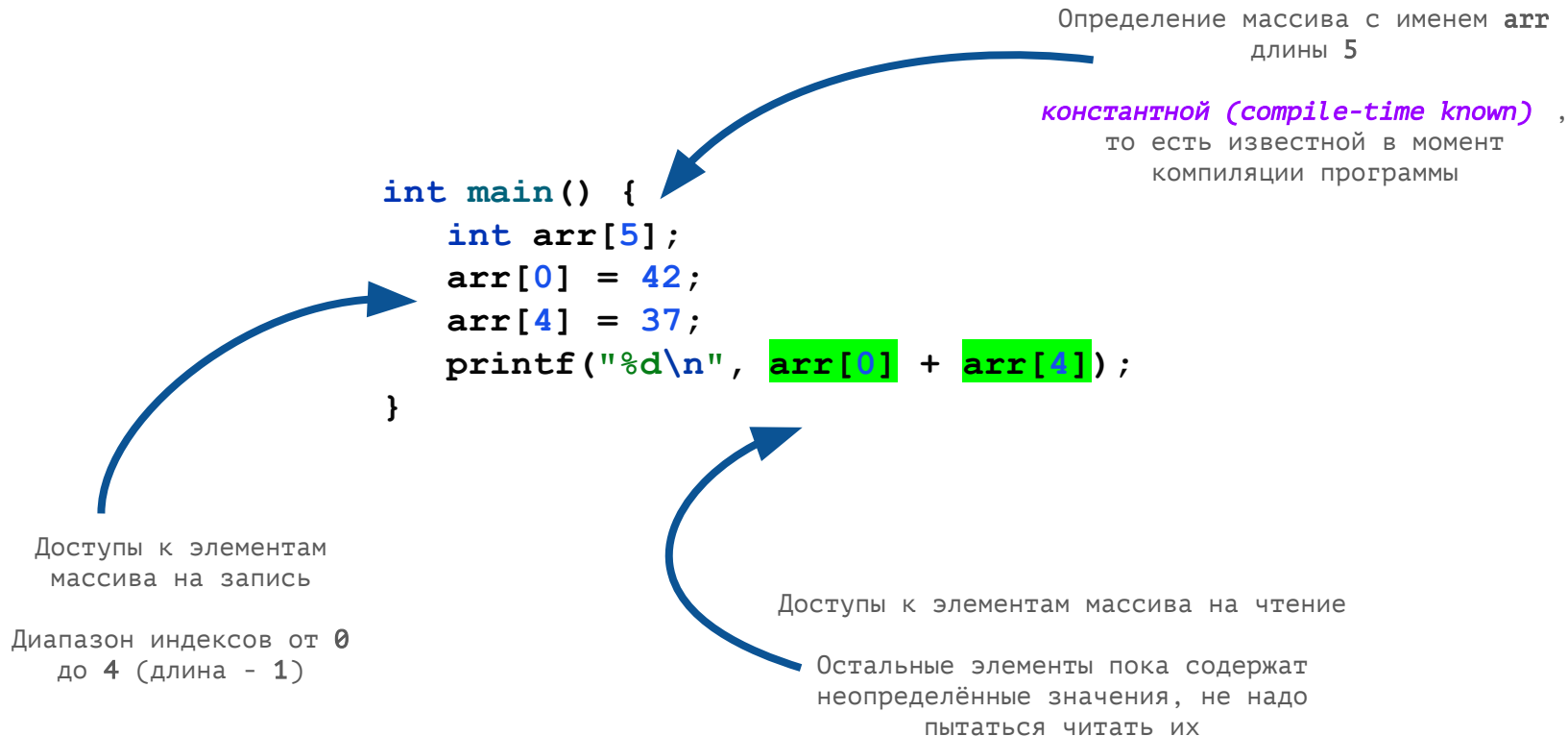
константной (compile-time known) ,
то есть известной в момент
компиляции программы

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
    printf("%d\n", arr[0] + arr[4]);  
}
```

Доступы к элементам
массива на запись

Диапазон индексов от 0
до 4 (длина - 1)

Массив константной длины



Реализация массивов

Особенность массива - доступ к любому его элементу должен быть за константное ($O(1)$) время, то есть не зависеть от индекса элемента, к которому достигаются, и длины массива

Реализация массивов

Особенность массива - доступ к любому его элементу должен быть за константное ($O(1)$) время, то есть не зависеть от индекса элемента, к которому достигаются, и длины массива

Массив константной длины, к элементам которого обращаются по константным индексам, может быть просто реализован как множество переменных, разложенных в памяти в любом порядке

Реализация массивов

Особенность массива - доступ к любому его элементу должен быть за константное ($O(1)$) время, то есть не зависеть от индекса элемента, к которому достигаются, и длины массива

Массив константной длины, к элементам которого обращаются по константным индексам, может быть просто реализован как множество переменных, разложенных в памяти в любом порядке

Доступ по *переменному (run-time known)* индексу требует правильной организации

Compile-time и run-time

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
}
```

Compile-time и run-time

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
}
```

Compile-time и run-time

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
}
```



```
int main() {  
    int arr_0, arr_1, arr_2, arr_3, arr_4;  
    arr_0 = 42;  
    arr_4 = 37;  
}
```

Compile-time и run-time

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
  
    int i;  
    scanf("%d", &i);  
    arr[i] = 19;  
}
```



```
int main() {  
    int arr_0, arr_1, arr_2, arr_3, arr_4;  
    arr_0 = 42;  
    arr_4 = 37;  
}
```

Compile-time и run-time

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```



```
int main() {  
    int arr_0, arr_1, arr_2, arr_3, arr_4;  
    arr_0 = 42;  
    arr_4 = 37;  
  
    int i;  
    scanf("%d", i);  
    arr_i = 19;  
}
```

Compile-time и run-time

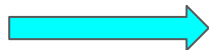
```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```



```
int main() {  
    int arr_0, arr_1, arr_2, arr_3, arr_4;  
    arr_0 = 42;  
    arr_4 = 37;  
  
    int i;  
    scanf("%d", i);  
    if (i == 0) {  
        arr_0 = 19;  
    } else if (i == 1) {  
        ...  
    }  
}
```

Compile-time и run-time

```
int main() {  
    int arr[5];  
    arr[0] = 42;  
    arr[4] = 37;  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```



Пропорционально
длине массива

```
int main() {  
    int arr_0, arr_1, arr_2, arr_3, arr_4;  
    arr_0 = 42;  
    arr_4 = 37;  
  
    int i;  
    scanf("%d", i);  
    if (i == 0) {  
        arr_0 = 19;  
    } else if (i == 1) {  
        ...  
    }  
}
```

Реализация массивов

Для эффективной реализации доступа к элементу массива по переменному индексу, его элементы размещают в памяти подряд

Реализация массивов

Для эффективной реализации доступа к элементу массива по переменному индексу, его элементы размещают в памяти подряд

Сам массив является указателем на нулевой элемент

Реализация массивов

Для эффективной реализации доступа к элементу массива по переменному индексу, его элементы размещают в памяти подряд

Сам массив является указателем на нулевой элемент

Операция `arr[i]` - это синтаксический сахар над выражением `*(arr + i)`

Реализация массивов

```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```

Реализация массивов

```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```

N + 28		}	Фрейм foo
N + 24	ret_addr		
N + 20	...		
N + 16	arr[4]: ???		
N + 12	arr[3]: ???		
N + 8	arr[2]: ???		
N + 4	arr[1]: ???		
N	arr[0]: ???		
N - 4	i: ???		

Реализация массивов

```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```

N + 28		}	Фрейм foo
N + 24	ret_addr		
N + 20	...		
N + 16	arr[4]: ???		
N + 12	arr[3]: ???		
N + 8	arr[2]: ???		
N + 4	arr[1]: ???		
N	arr[0]: ???		
N - 4	i: ???		

Элементы массива лежат подряд по увеличению адресов

Реализация массивов

```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```

N + 28		}	Фрейм foo
N + 24	ret_addr		
N + 20	...		
N + 16	arr[4]: ???		
N + 12	arr[3]: ???		
N + 8	arr[2]: ???		
N + 4	arr[1]: ???		
N	arr[0]: ???		
N - 4	i: ???		

Переменная **arr** не существует (не обязательно), но можно представить её себе, как переменную типа **int*** со значением **N**

Реализация массивов

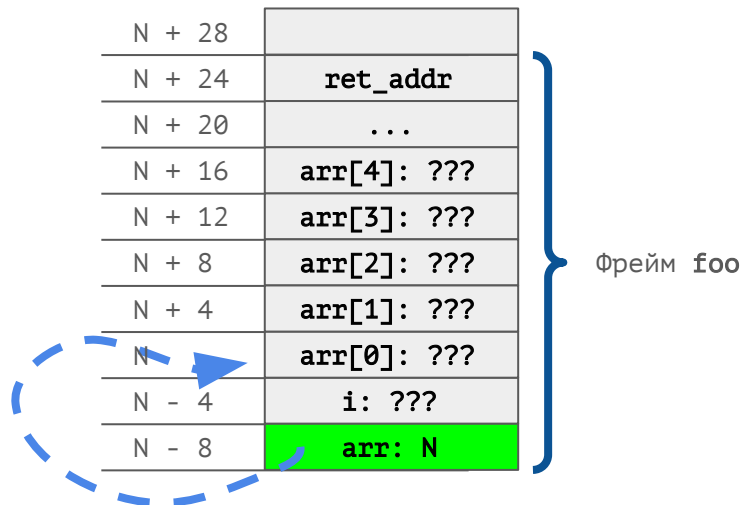
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```

N + 28		} Фрейм foo
N + 24	ret_addr	
N + 20	...	
N + 16	arr[4]: ???	
N + 12	arr[3]: ???	
N + 8	arr[2]: ???	
N + 4	arr[1]: ???	
N	arr[0]: ???	
N - 4	i: ???	
N - 8	arr: N	

Переменная **arr** не существует (не обязательно), но можно представить её себе, как переменную типа **int*** со значением **N**

Реализация массивов

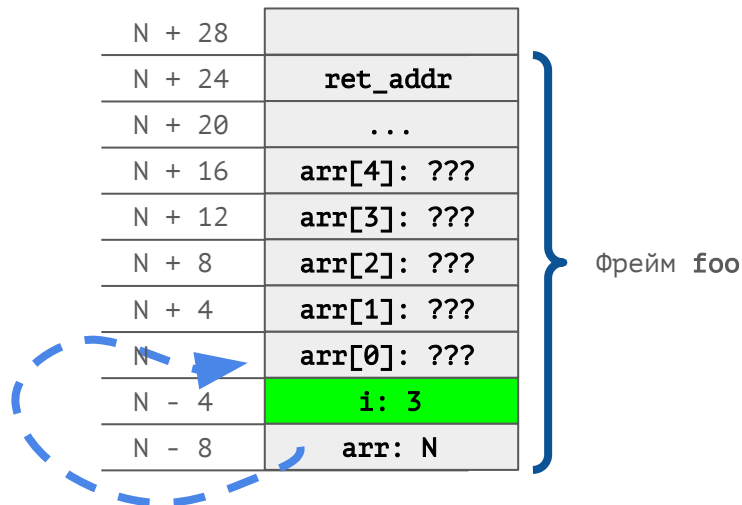
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```



Переменная `arr` не существует (не обязательно), но можно представить её себе, как переменную типа `int*` со значением `N`

Реализация массивов

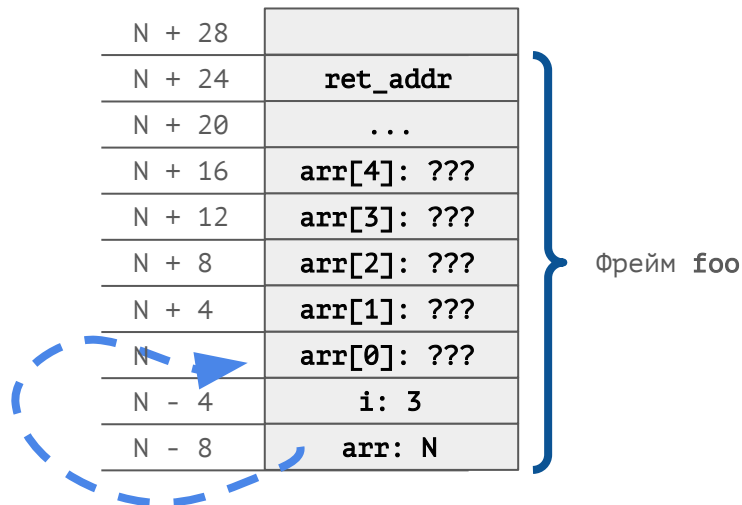
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```



Предположим, пользователь ввёл число 3

Реализация массивов

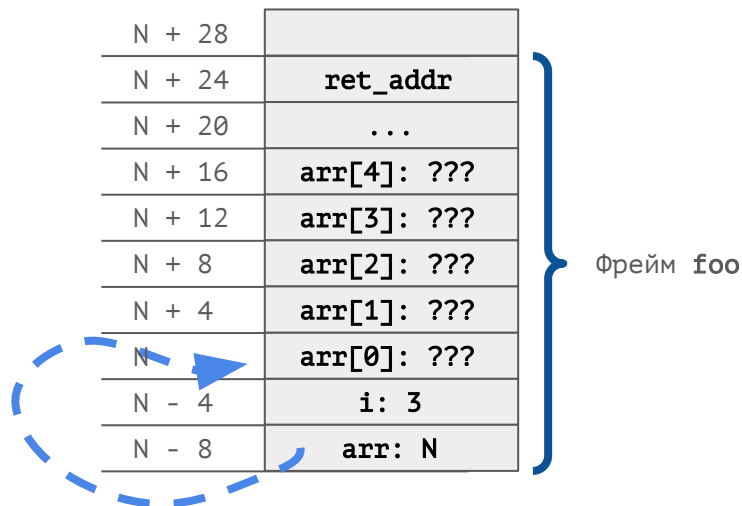
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```



Выражение `arr[i]` эквивалентно `*(arr + i)`

Реализация массивов

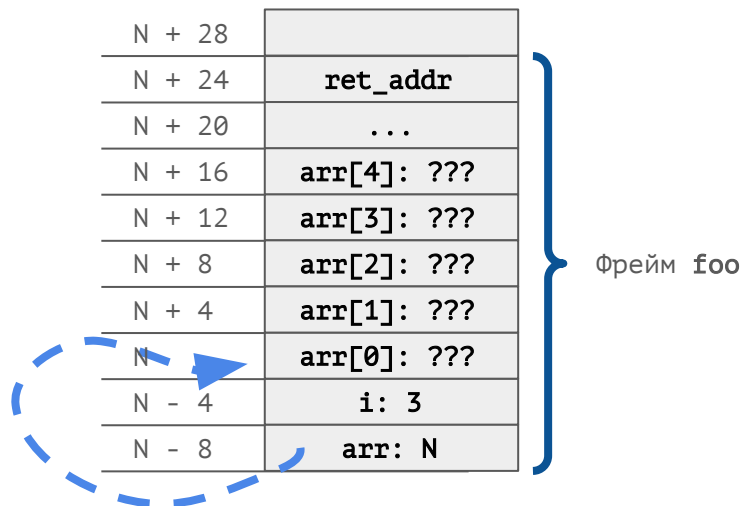
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    *(arr + i) = 19;  
}
```



arr равен N, i равно 3, следовательно arr + i равен

Реализация массивов

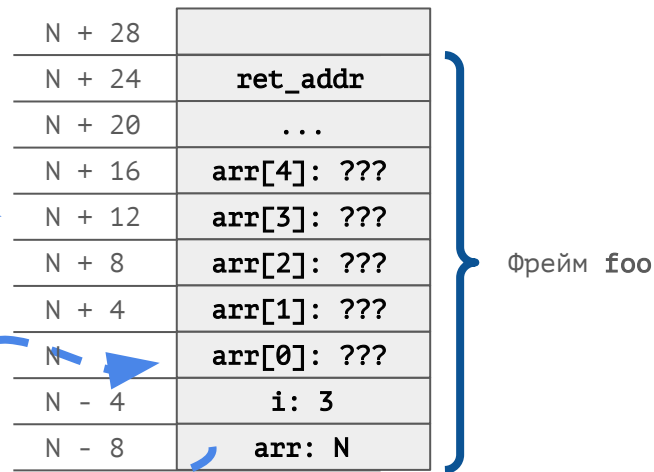
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    *(arr + i) = 19;  
}
```



arr равен N, i равно 3, следовательно arr + i равен N + 12

Реализация массивов

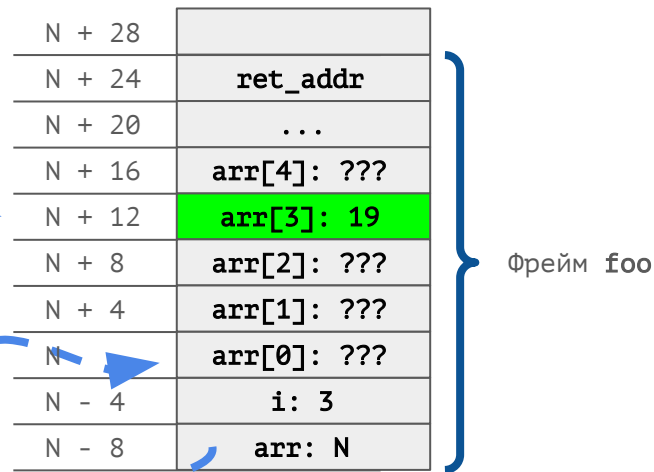
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    * (arr + i) = 19;  
}
```



`arr` равен `N`, `i` равно `3`, следовательно `arr + i` равен `N + 12`

Реализация массивов

```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    *(arr + i) = 19;  
}
```



arr равен N, i равно 3, следовательно arr + i равен N + 12

Реализация массивов

Укладка элементов подряд в памяти не является очевидным или единственно возможным решением - это следствие выбора, какие операции мы хотим иметь эффективными

Реализация массивов

Укладка элементов подряд в памяти не является очевидным или единственно возможным решением - это следствие выбора, какие операции мы хотим иметь эффективными

Это решение оптимизирует операцию доступа по переменному индексу, но, например, операцию добавления элемента в начало последовательности пока вообще непонятно, как сделать

А когда станет понятно, также станет понятно, что она очень неэффективная

Реализация массивов

Укладка элементов подряд в памяти не является очевидным или единственно возможным решением - это следствие выбора, какие операции мы хотим иметь эффективными

Это решение оптимизирует операцию доступа по переменному индексу, но, например, операцию добавления элемента в начало последовательности пока вообще непонятно, как сделать

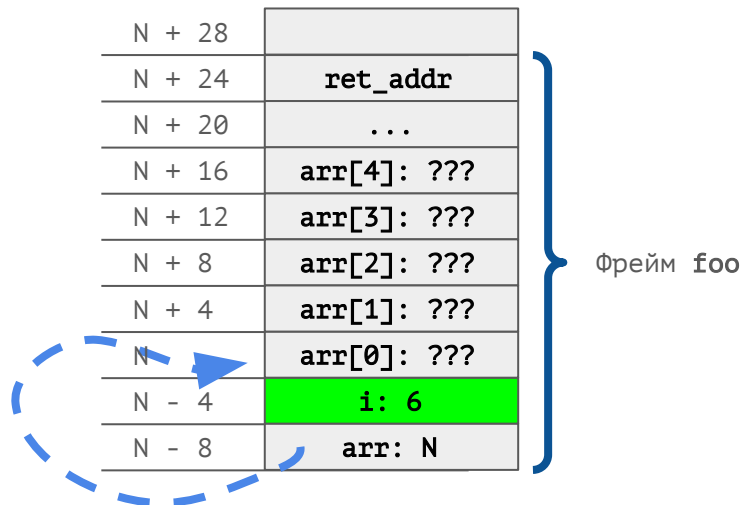
А когда станет понятно, также станет понятно, что она очень неэффективная

Никакие структуры данных не оптимизируют все возможные с ними операции; мы подбираем структуры данных под алгоритмы, учитывая, какие операции чаще производим со структурой

Всё это вы подробно изучите в курсе алгоритмов; эти темы не сильно относятся к нашему курсу, в отличие от ...

Выход за границы массива

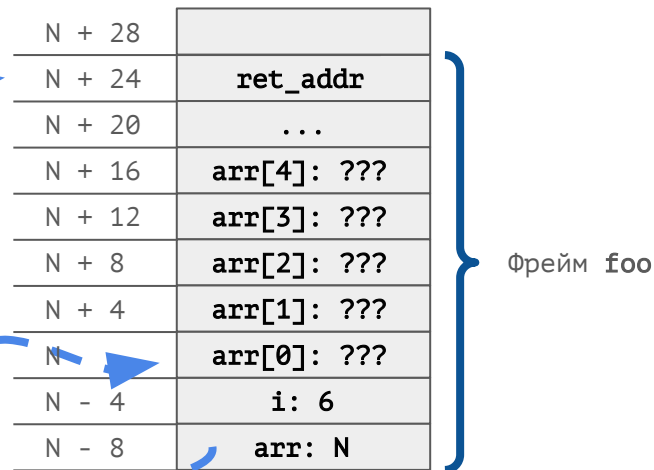
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    arr[i] = 19;  
}
```



Предположим, пользователь ввёл число 6

Выход за границы массива

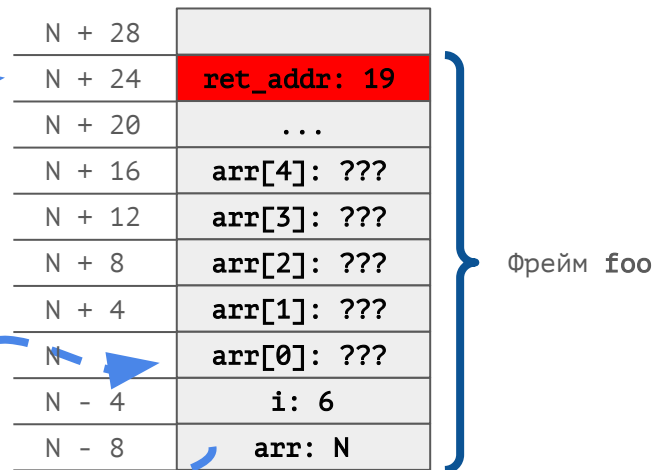
```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    * (arr + i) = 19;  
}
```



`arr` равен `N`, `i` равно 6, следовательно `arr + i` равен `N + 24`

Выход за границы массива

```
int foo() {  
    int arr[5];  
  
    int i;  
    scanf("%d", i);  
    *(arr + i) = 19;  
}
```



`arr` равен `N`, `i` равно `6`, следовательно `arr + i` равен `N + 24`

Выход за границы массива

Операции доступа к элементам массива никак не проверяют, что индекс попадает в допустимый диапазон - это позволяет делать мощные оптимизации и генерировать эффективный код

Выход за границы массива

Операции доступа к элементам массива никак не проверяют, что индекс попадает в допустимый диапазон - это позволяет делать мощные оптимизации и генерировать эффективный код

Выход за границы провоцирует **UB** (один из самых часто встречающихся и трудно отлаживаемых)

Формально, уже в момент вычисления адреса, выходящего за границы, и то не всегда, но об этом пока рано говорить

Индексация массива

Индексировать массивы типом `int` неправильно - отрицательные индексы не нужны, а положительный диапазон `int` может не вмещать все существующие индексы

На самом деле отрицательные индексы могут существовать и даже быть полезными, но об этом пока рано говорить

Индексация массива

Индексировать массивы типом `int` неправильно - отрицательные индексы не нужны, а положительный диапазон `int` может не вмещать все существующие индексы

На самом деле отрицательные индексы могут существовать и даже быть полезными, но об этом пока рано говорить

Правильная индексация - типом `size_t` - он беззнаковый и так как вмещает все возможные размеры объектов в байтах, вмещает и все возможные индексы

Индексация беззнаковыми типами имеет несколько недостатков

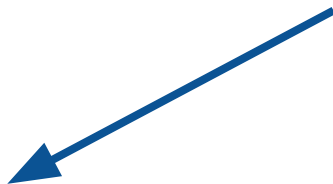
Итерация по массиву

```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
}
```

Итерация по массиву

```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
}
```

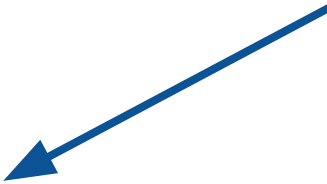
Стандартная форма перебора
индексов массива от начала
к концу



Итерация по массиву

```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
}
```

Стандартная форма перебора
индексов массива от начала
к концу

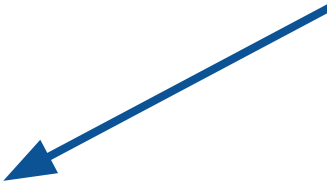


Давайте распечатаем элементы
массива в обратном порядке
через пробел

Итерация по массиву

```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
  
    for (size_t i = 4; i >= 0; i--) {  
        printf("%d ", arr[i]);  
    }  
}
```

Стандартная форма перебора
индексов массива от начала
к концу

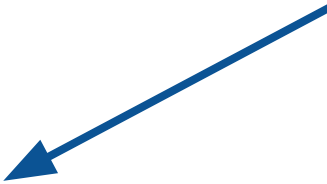


Давайте распечатаем элементы
массива в обратном порядке
через пробел

Итерация по массиву

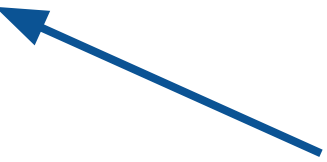
```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
  
    for (size_t i = 4; i >= 0; i--) {  
        printf("%d ", arr[i]);  
    }  
}
```

Стандартная форма перебора
индексов массива от начала
к концу



Давайте распечатаем элементы
массива в обратном порядке
через пробел

Всегда истина!



Итерация по массиву

```
int main() {  
    int arr[5];
```

```
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }
```

```
    for (size_t i = 4; i >= 0; i--) {  
        printf("%d ", arr[i]);  
    }
```

```
}
```

Бесконечный
цикл!

Стандартная форма перебора
индексов массива от начала
к концу

Давайте распечатаем элементы
массива в обратном порядке
через пробел

Всегда истина!

Итерация по массиву

```
int main() {  
    int arr[5];
```

```
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;
```

```
    }
```

```
    for (size_t i = 4; i >= 0; i--) {  
        printf("%d ", arr[i]);
```

```
    }
```

```
}
```

Стандартная форма перебора
индексов массива от начала
к концу

Давайте распечатаем элементы
массива в обратном порядке
через пробел

Всегда истина!

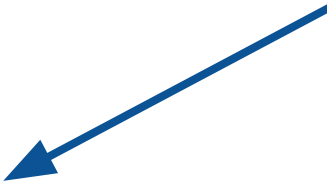
UB после `i == 0`

Бесконечный
цикл!

Итерация по массиву

```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
  
    for (size_t x = 5; x > 0; x--) {  
        size_t i = x - 1;  
        printf("%d ", arr[i]);  
    }  
}
```

Стандартная форма перебора
индексов массива от начала
к концу



Давайте распечатаем элементы
массива в обратном порядке
через пробел

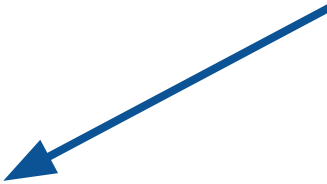
Можно перебирать диапазон,
сдвинутый на 1



Итерация по массиву

```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
  
    for (size_t i = 5; i-- > 0;) {  
        printf("%d ", arr[i]);  
    }  
}
```

Стандартная форма перебора
индексов массива от начала
к концу



Давайте распечатаем элементы
массива в обратном порядке
через пробел

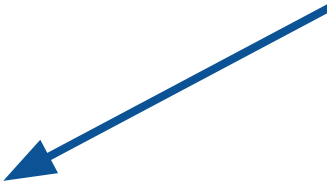
Можно воспользоваться
оригинальной техникой



Итерация по массиву

```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
  
    for (size_t i = 4; i > 0; i--) {  
        printf("%d ", arr[i]);  
    }  
    printf("%d", arr[0]);  
}
```

Стандартная форма перебора
индексов массива от начала
к концу



Давайте распечатаем элементы
массива в обратном порядке
через пробел

Наконец, можно просто
вынести последний шаг за
цикл.



Вынос шага итерации за цикл

Часто бывает, что первый или последний шаг итерации отличаются от остальных - тогда его иногда выносят, дублируя код

Вынос шага итерации за цикл

Часто бывает, что первый или последний шаг итерации отличаются от остальных - тогда его иногда выносят, дублируя код

Заметим, что только это решение из рассмотренных корректно - на последнем шаге итерации не надо выводить пробел после числа

Итерация по массиву

```
int main() {  
    int arr[5];  
  
    for (size_t i = 0; i < 5; i++) {  
        arr[i] = (int) i;  
    }  
  
    for (size_t i = 4; i > 0; i--) {  
        printf("%d ", arr[i]);  
    }  
    printf("%d", arr[0]);  
}
```

Вынос шага итерации за цикл

Часто бывает, что первый или последний шаг итерации отличаются от остальных - тогда его иногда выносят, дублируя код

Заметим, что только это решение из рассмотренных корректно - на последнем шаге итерации не надо выводить пробел после числа

Нужно с осторожностью применять эту технику только там, где дублирование кода компенсируется упрощением каждой из копий, а также учитывать, работает ли этот код с циклами с 0 итераций (такие тоже бывают)

Вынос шага итерации за цикл

```
for (size_t i = 4; i > 0; i--) {  
    printf("%d ", arr[i]);  
}  
printf("%d", arr[0]);
```

```
for (size_t i = 5; i-- > 0;) {  
    printf("%d", arr[i]);  
    if (i != 0) {  
        printf(" ");  
    }  
}
```

Какой код проще?

Поиск в массивах

Стандартная задача - найти в массиве первый или последний элемент, удовлетворяющий какому-нибудь свойству

Поиск в массивах

Стандартная задача - найти в массиве первый или последний элемент, удовлетворяющий какому-нибудь свойству

Логично было бы возвращать из функций поиска индекс найденного элемента, но из-за индексации массивов типом `size_t` нет никакого невалидного значения, которое можно было бы вернуть в качестве результата, что такого элемента не существует

Поиск в массивах

Стандартная задача - найти в массиве первый или последний элемент, удовлетворяющий какому-нибудь свойству

Логично было бы возвращать из функций поиска индекс найденного элемента, но из-за индексации массивов типом **size_t** нет никакого невалидного значения, которое можно было бы вернуть в качестве результата, что такого элемента не существует

В языке Си принят де-факто стандарт возвращать не индекс, а адрес найденного элемента, либо **NULL**, как признак отсутствия элемента

Преобразование адреса в индекс

Имея адрес элемента в массиве, его индекс можно получить операцией вычитания адреса начала массива (то есть, самого массива)

Преобразование адреса в индекс

Имея адрес элемента в массиве, его индекс можно получить операцией вычитания адреса начала массива (то есть, самого массива)

Вычитание адресов возвращает значение знакового типа `ptrdiff_t`, определённого в `stddef.h`, и работает по принципам адресной арифметики: `&arr[i] - &arr[j] == i - j`

Преобразование адреса в индекс

Имея адрес элемента в массиве, его индекс можно получить операцией вычитания адреса начала массива (то есть, самого массива)

Вычитание адресов возвращает значение знакового типа `ptrdiff_t`, определённого в `stddef.h`, и работает по принципам адресной арифметики: `&arr[i] - &arr[j] == i - j`

Вычитание адресов из разных объектов провоцирует **UB**

Передача массива в функцию

Всегда происходит по адресу (передаётся значение указателя на нулевой элемент)

Передача массива в функцию

```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

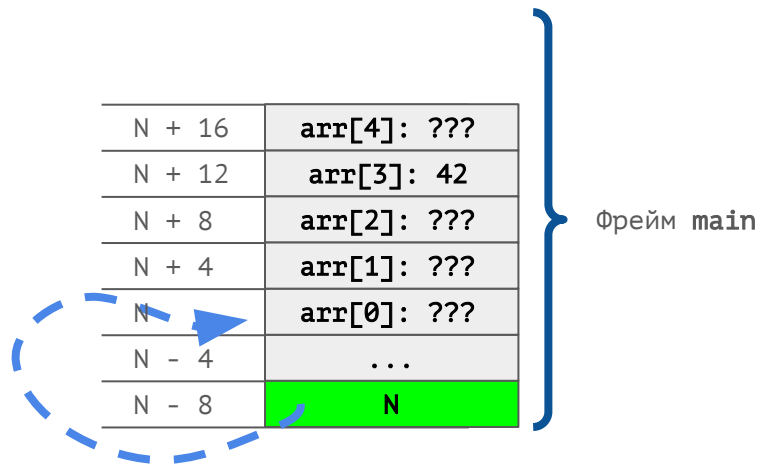


N + 16	arr[4]: ???
N + 12	arr[3]: 42
N + 8	arr[2]: ???
N + 4	arr[1]: ???
N	arr[0]: ???
N - 4	...
N - 8	

Фрейм main

Передача массива в функцию

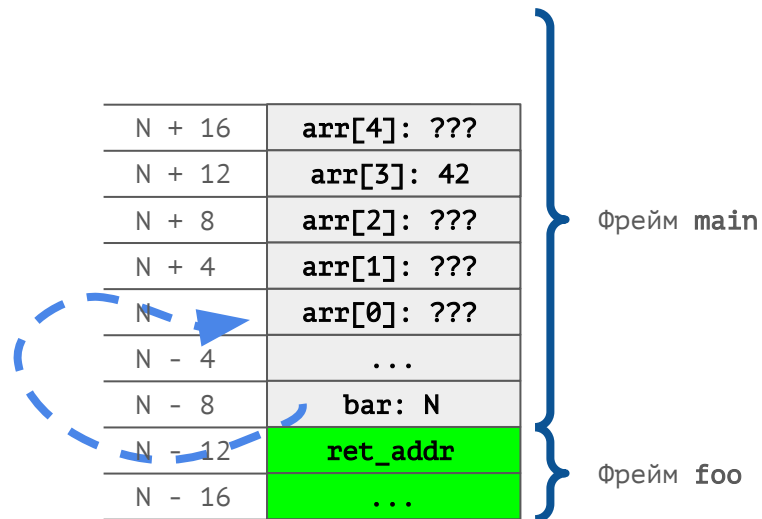
```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```



Передача массива в функцию



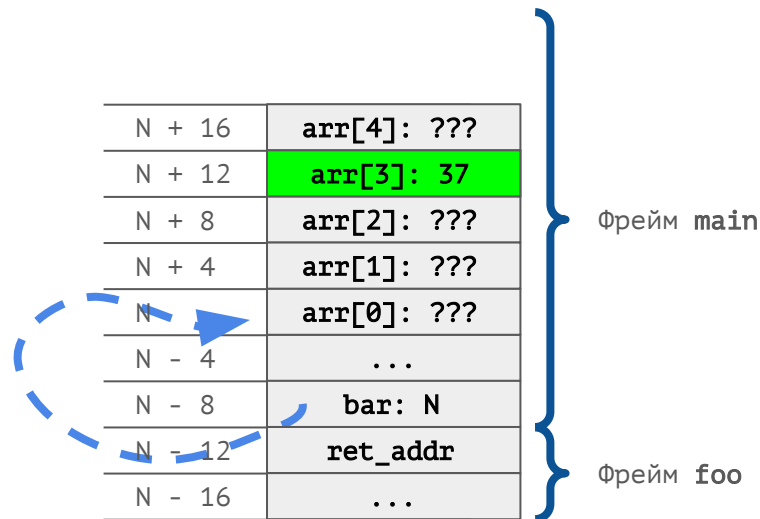
```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```



Передача массива в функцию



```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```



Передача массива в функцию

Всегда происходит по адресу (передаётся значение указателя на нулевой элемент)

Передавать массивы следует осторожно - функции могут изменить их элементы; но этим можно и нужно пользоваться, когда вам именно это и нужно по семантике (например, в функциях сортировки)

Передача массива в функцию

```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int bar[5]) {  
    bar[3] = 37;  
}
```

Вот это число вообще ни на
что не влияет

```
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int bar[2]) {  
    bar[3] = 37;  
}
```

Можно даже вот так написать

```
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int bar[2]) {  
    bar[3] = 37;  
}
```

Можно даже вот так написать

```
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

И эта операция не будет ошибкой

Массив - это указатель

Массив - это только адрес в памяти, от которого отсчитываются индексы; даже длина не является частью массива, и её невозможно из него получить

Массив - это указатель

Массив - это только адрес в памяти, от которого отсчитываются индексы; даже длина не является частью массива, и её невозможно из него получить

Можно взять адрес какого-нибудь элемента массива и представить массивом, тогда индексы будут отсчитываться от него

Здесь внимательный слушатель может вспомнить про то, что отрицательные индексы бывают полезны

Передача массива в функцию

```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(&arr[1]);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(&arr[1]);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(&arr[1]);  
    printf("%d\n", arr[3]);  
}
```

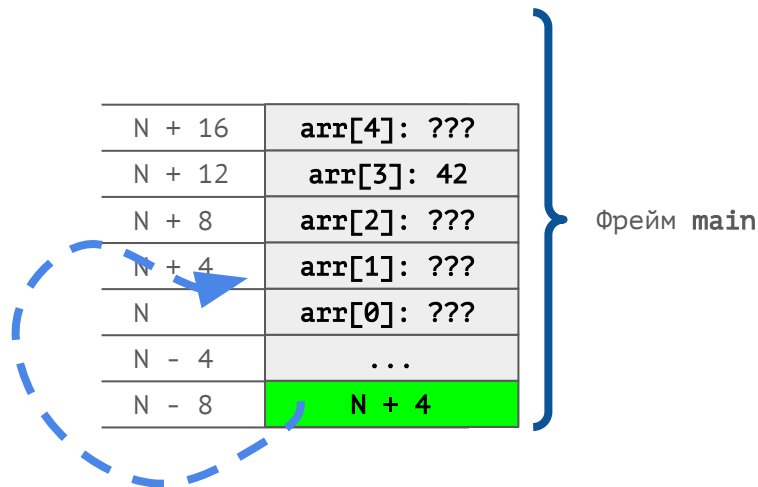


N + 16	arr[4]: ???
N + 12	arr[3]: 42
N + 8	arr[2]: ???
N + 4	arr[1]: ???
N	arr[0]: ???
N - 4	...
N - 8	

Фрейм main

Передача массива в функцию

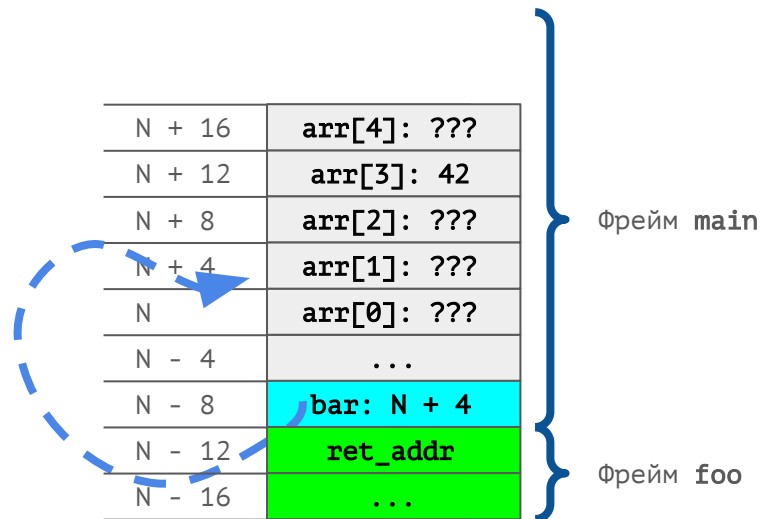
```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(&arr[1]);  
    printf("%d\n", arr[3]);  
}
```



Передача массива в функцию



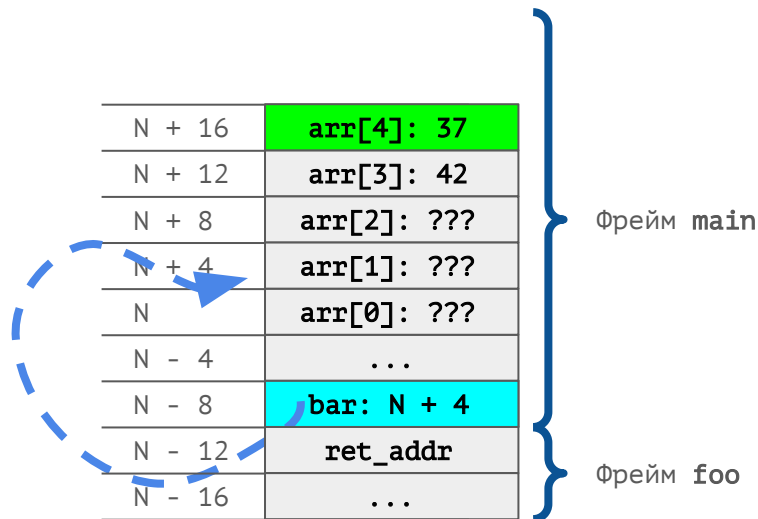
```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(&arr[1]);  
    printf("%d\n", arr[3]);  
}
```



Передача массива в функцию



```
void foo(int bar[5]) {  
    bar[3] = 37;  
}  
  
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(&arr[1]);  
    printf("%d\n", arr[3]);  
}
```



Массив - это указатель

Массив - это только адрес в памяти, от которого отсчитываются индексы; даже длина не является частью массива, и её невозможно из него получить

Можно взять адрес какого-нибудь элемента массива и представить массивом, тогда индексы будут отсчитываться от него

Здесь внимательный слушатель может вспомнить про то, что отрицательные индексы бывают полезны

Операция `[]` применима и к указателям (с такой же семантикой)

Передача массива в функцию

```
void foo(int bar[5]) {  
    bar[3] = 37;  
}
```

Вот это число вообще ни на
что не влияет

```
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int bar[]) {  
    bar[3] = 37;  
}
```

Лучше писать вот так

```
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int* bar) {  
    bar[3] = 37;  
}
```

Или даже вот так

```
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int* bar) {  
    bar[3] = 37;  
}
```

Или даже вот так

```
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Передача массива в функцию

```
void foo(int* bar) {  
    *(bar + 3) = 37;  
}
```

Или даже вот так

```
int main() {  
    int arr[5];  
    arr[3] = 42;  
    foo(arr);  
    printf("%d\n", arr[3]);  
}
```

Возврат массива из функции

Осуществляется так же по указателю - значению адреса нулевого элемента

Возврат массива из функции

Осуществляется так же по указателю - значению адреса нулевого элемента

Если вернуть адрес локального массива (автоматического типа хранения), вы получите адрес объекта, чьё время жизни уже закончилось; его использование провоцирует **UB**

То же самое касается адреса любого другого локального объекта

Возврат массива из функции

Осуществляется так же по указателю - значению адреса нулевого элемента

Если вернуть адрес локального массива (автоматического типа хранения), вы получите адрес объекта, чьё время жизни уже закончилось; его использование провоцирует **UB**

То же самое касается адреса любого другого локального объекта

Можно возвращать адреса глобальных массивов (статического типа хранения); также мы вернёмся к этой теме, когда будем изучать динамическую память

Массивы разных длин

```
void printArr(int* arr) {  
    for (size_t i = 0; i < 5; i++) {  
        printf("%d", arr[i]);  
        if (i != 4) {  
            printf(" ");  
        }  
    }  
}  
  
int main() {  
    int arr[5]; ...  
    printArr(arr);  
}
```

Массивы разных длин

```
void printArr(int* arr) {
    for (size_t i = 0; i < 5; i++) {
        printf("%d", arr[i]);
        if (i != 4) {
            printf(" ");
        }
    }
}

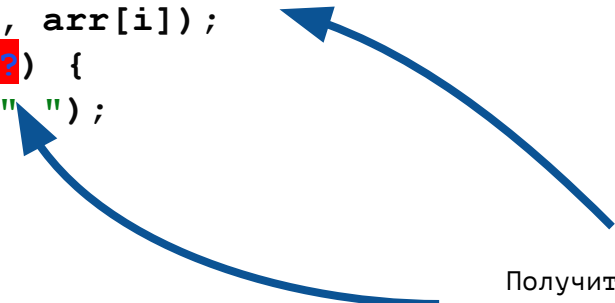
int main() {
    int arr[5]; ...
    printArr(arr);

    int mas[7]; ...
    printArr(mas);
}
```

Массивы разных длин

```
void printArr(int* arr) {  
    for (size_t i = 0; i < ???; i++) {  
        printf("%d", arr[i]);  
        if (i != ???) {  
            printf(" ");  
        }  
    }  
}
```

```
int main() {  
    int arr[5]; ...  
    printArr(arr);  
  
    int mas[7]; ...  
    printArr(mas);  
}
```



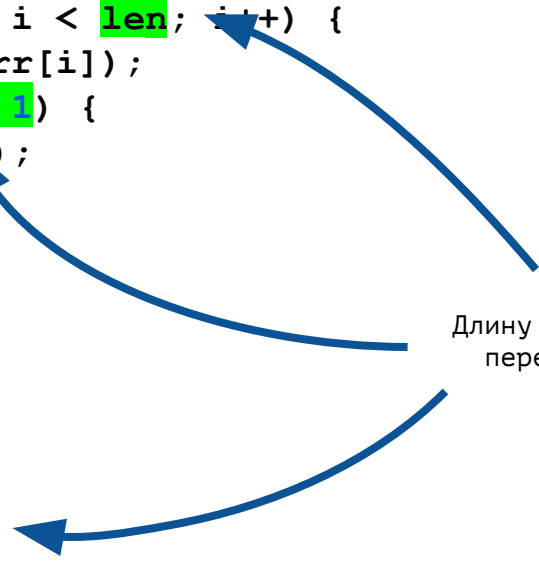
Получить 5 или 7 из `arr`
невозможно никаким образом

Массивы разных длин

```
void printArr(int* arr, size_t len) {  
    for (size_t i = 0; i < len; i++) {  
        printf("%d", arr[i]);  
        if (i != len - 1) {  
            printf(" ");  
        }  
    }  
}
```

```
int main() {  
    int arr[5]; ...  
    printArr(arr, 5);  
  
    int mas[7]; ...  
    printArr(mas, 7);  
}
```

Длину массива нужно
передавать явно



Рассмотрим задачу

С клавиатуры вводится число N , а затем N чисел, которые нужно сохранить в массив

Рассмотрим задачу

С клавиатуры вводится число N , а затем N чисел, которые нужно сохранить в массив

Если возможные значения N ограничены сверху некоторым известным $MAXN$, можно выделить массив длины $MAXN$ и использовать его начальный отрезок

Рассмотрим задачу

С клавиатуры вводится число **N**, а затем **N** чисел, которые нужно сохранить в массив

Если возможные значения **N** ограничены сверху некоторым известным **MAXN**, можно выделить массив длины **MAXN** и использовать его начальный отрезок

MAXN может не быть известно, а даже если известно, это всё равно расход памяти впустую

VLA

Массив переменной длины (variable length array, VLA) - массив, длина которого является `run-time known` значением

VLA

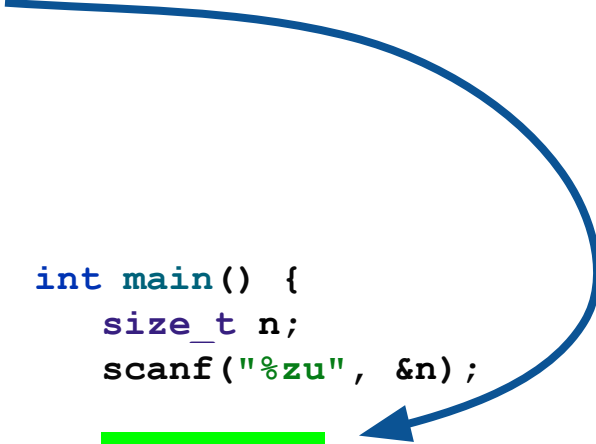
Массив переменной длины (variable length array, VLA) - массив, длина которого является `run-time known` значением

VLA не может быть создан в статической памяти (размер глобальных массивов обязан быть `compile-time known` значением), только в автоматической или динамической

VLA

```
int main() {  
    size_t n;  
    scanf("%zu", &n);  
  
    int arr[n];  
    for (size_t i = 0; i < n; i++) {  
        scanf("%d", &arr[i]);  
    }  
}
```

VLA



```
int main() {  
    size_t n;  
    scanf("%zu", &n);  
  
    int arr[n];  
    for (size_t i = 0; i < n; i++) {  
        scanf("%d", &arr[i]);  
    }  
}
```

VLA

У **VLA** долгая и сложная судьба - в C99 они появились и стали частью языка, в C11 их сделали необязательными, в C23 собираются вернуть, но с ограничениями

VLA

У **VLA** долгая и сложная судьба - в C99 они появились и стали частью языка, в C11 их сделали необязательными, в C23 собираются вернуть, но с ограничениями

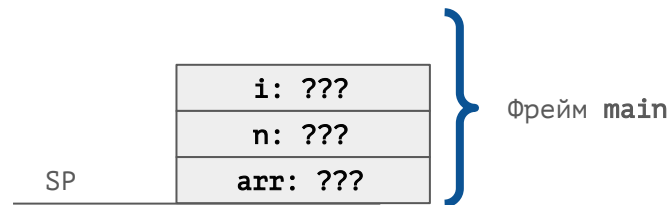
Связано это с проблемами реализации **VLA** на стеке (в автоматической памяти) и плохой репутацией в сообществе программистов

"So no. VLA's are not acceptable in the kernel. Don't do them. We're getting rid of them." Linus

Проблема с использованием локальных переменных



```
int main() {  
    size_t n;  
    scanf("%zu", &n);  
  
    int arr[n];  
    for (size_t i = 0; i < n; i++) {  
        scanf("%d", &arr[i]);  
    }  
}
```



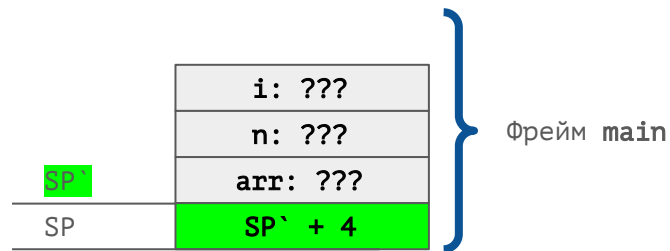
Проблема с использованием локальных переменных

Компилятор раскладывает переменные во фрейме функции так, как считает нужным, и использует их через регистр **SP** по известным (**compile-time known**) смещениям, как элементы своеобразного массива

Проблема с использованием локальных переменных



```
int main() {  
    size_t n;  
    scanf("%zu", &n);  
  
    int arr[n];  
    for (size_t i = 0; i < n; i++) {  
        scanf("%d", &arr[i]);  
    }  
}
```



Проблема с использованием локальных переменных

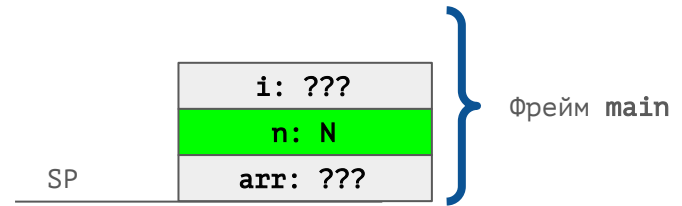
Компилятор раскладывает переменные во фрейме функции так, как считает нужным, и использует их через регистр **SP** по известным (**compile-time known**) смещениям, как элементы своеобразного массива

При выделении **VLA** во фрейме функции, регистр **SP** сдвигается вниз на **run-time known** значение

Проблема с использованием локальных переменных

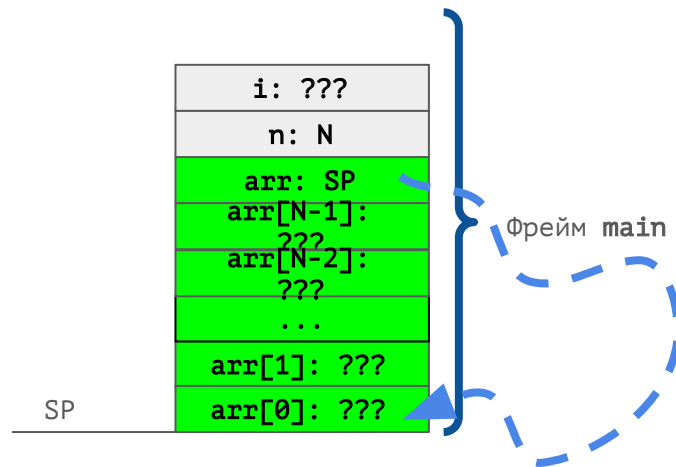


```
int main() {  
    size_t n;  
    scanf("%zu", &n);  
  
    int arr[n];  
    for (size_t i = 0; i < n; i++) {  
        scanf("%d", &arr[i]);  
    }  
}
```



Проблема с использованием локальных переменных

```
int main() {  
    size_t n;  
    scanf("%zu", &n);  
  
    int arr[n];  
    for (size_t i = 0; i < n; i++) {  
        scanf("%d", &arr[i]);  
    }  
}
```



Проблема с использованием локальных переменных

Компилятор раскладывает переменные во фрейме функции так, как считает нужным, и использует их через регистр **SP** по известным (**compile-time known**) смещениям, как элементы своеобразного массива

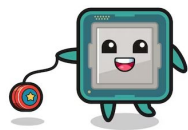
При выделении **VLA** во фрейме функции, регистр **SP** сдвигается вниз на **run-time known** значение

Локальные переменные теперь размещаются на **run-time known** (неизвестном компилятору) расстоянии от **SP**; есть разные техники работы с ними, но каждая из них в любом случае является более сложной и менее эффективной, чем работа с фреймом фиксированного размера

Проблема с переполнением стека

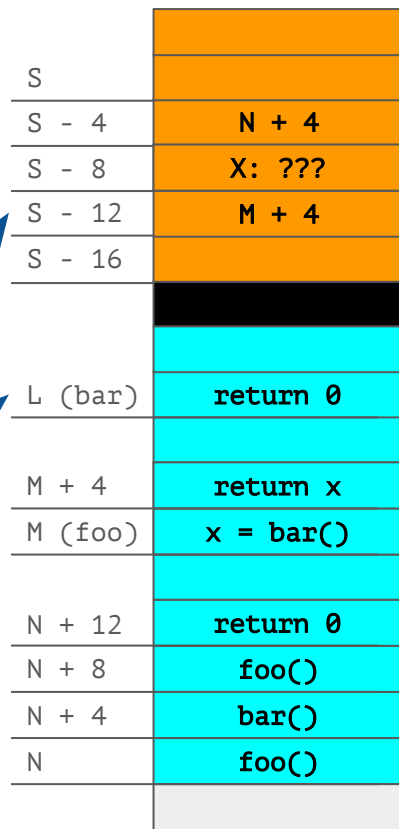
С помощью **VLA** очень легко сломать защиту от **UB** при переполнении стека

Напоминание



SP

PC



Фрейм main
Фрейм foo
Фрейм bar

Автоматическая память

Статическая память

Guard page



Проблема с переполнением стека

С помощью **VLA** очень легко сломать защиту от **UB** при переполнении стека

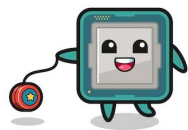
Чтобы получить лучшее **UB** в мире (мгновенный развал программы), нужно попасть в **guard page**
- попробовать разыменовать адрес, находящийся в ней

Проблема с переполнением стека

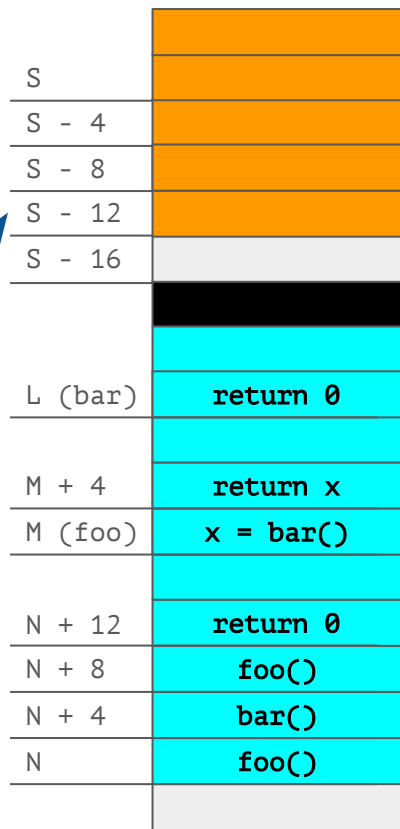
С помощью **VLA** очень легко сломать защиту от **UB** при переполнении стека

Чтобы получить лучшее **UB** в мире (мгновенный развал программы), нужно попасть в **guard page**
- попробовать разыменовать адрес, находящийся в ней

Находясь достаточно близко к **guard page** (4 Кб, то есть 1024 элемента типа **int**) и выделив достаточно большой **VLA**, можно получить адрес под **guard page**



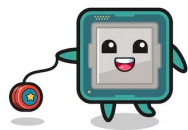
SP



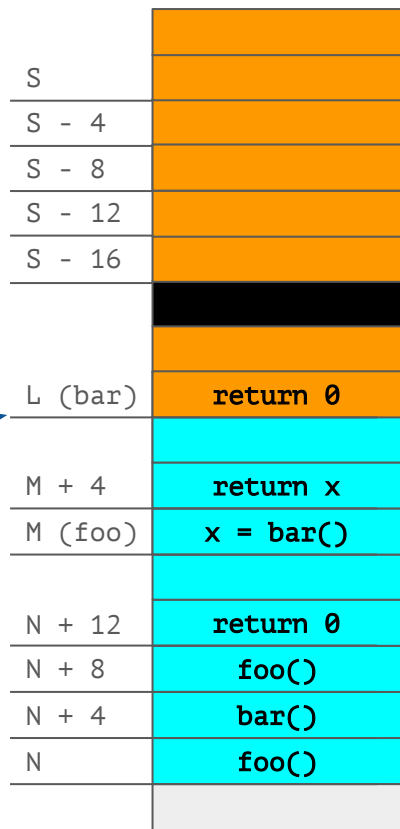
Фрейм main
Фрейм foo
Фрейм bar

Guard page





SP



Фрейм main

Фрейм foo

Фрейм bar

Guard page



VLA

VLA на стеке действительно сомнительная техника, но знать их и уметь пользоваться нужно (пара практических задач используют их)

Мы ещё вернёмся к этой теме в процессе изучения динамической памяти и многомерных массивов