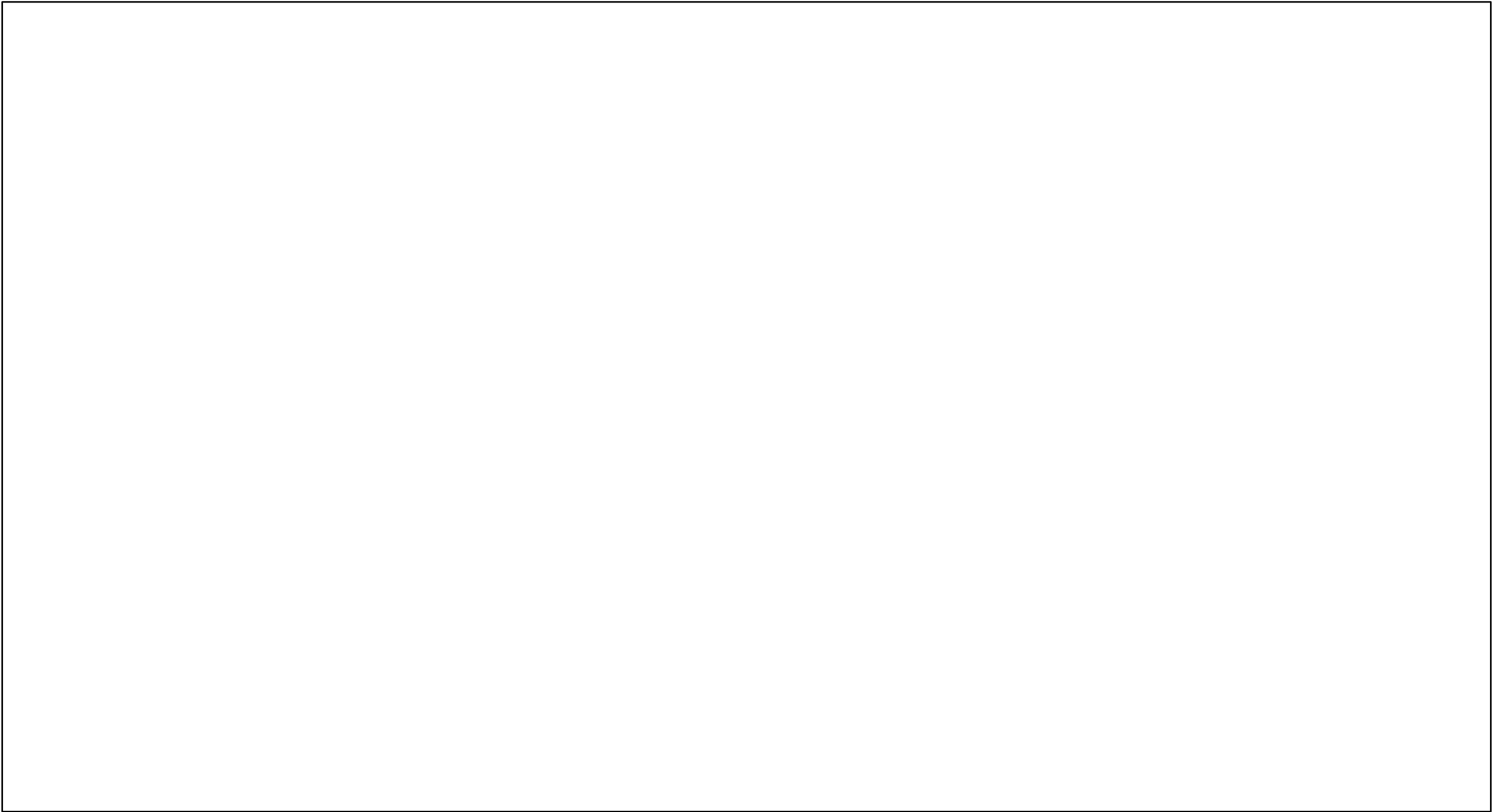


GENESIS

Минимум про массивы и указатели. Виртуальная память и разновидности памяти.

К. Владимиров, Syntacore, 2023
mail-to: konstantin.vladimirov@gmail.com



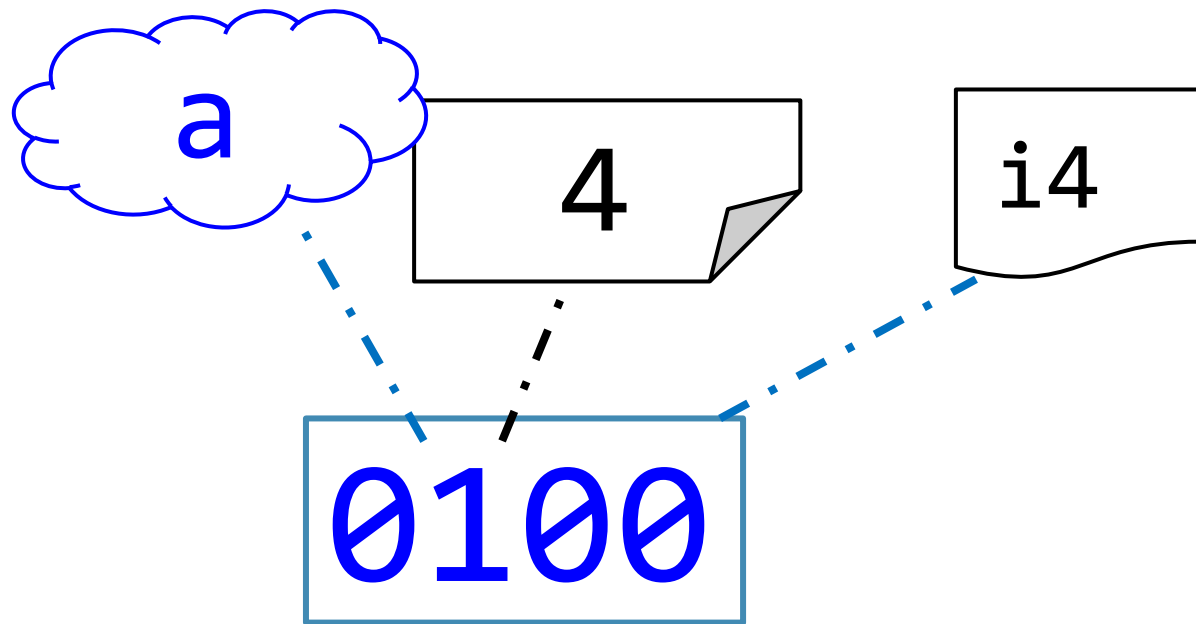
0100

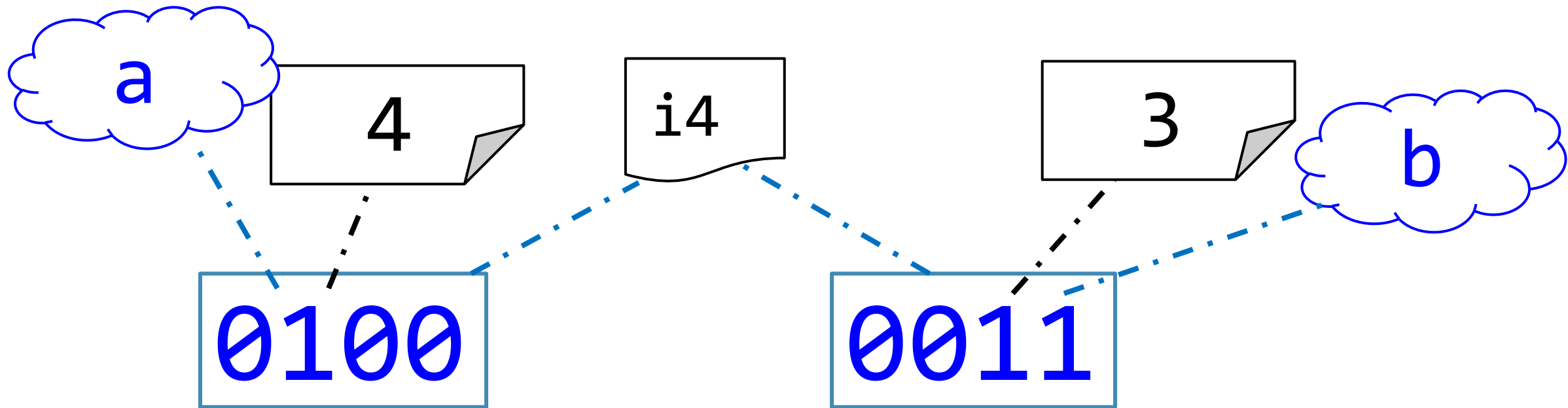
a

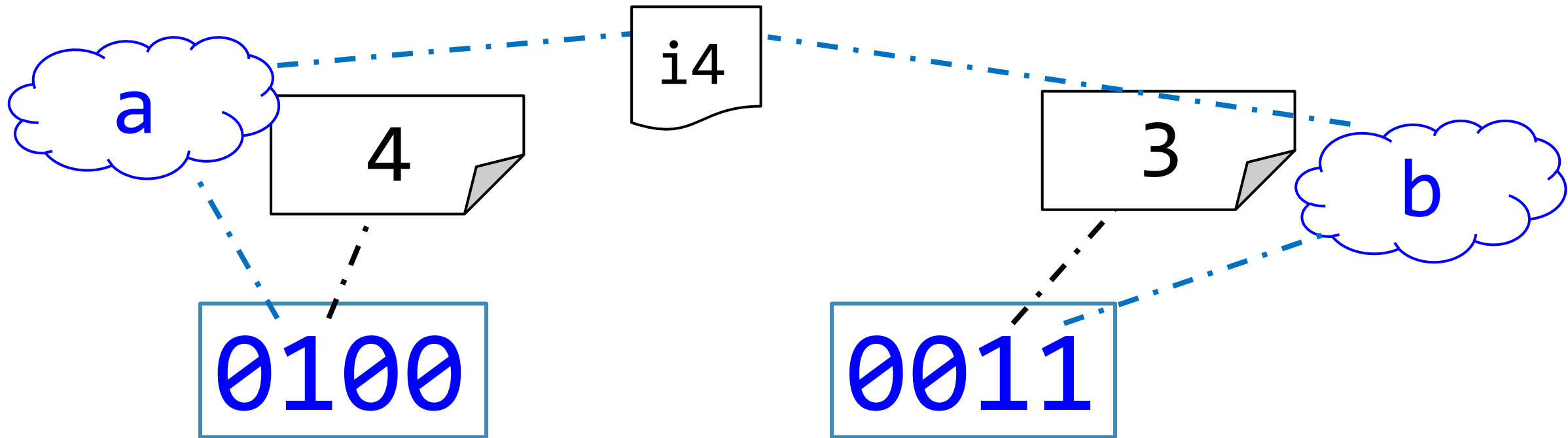
4

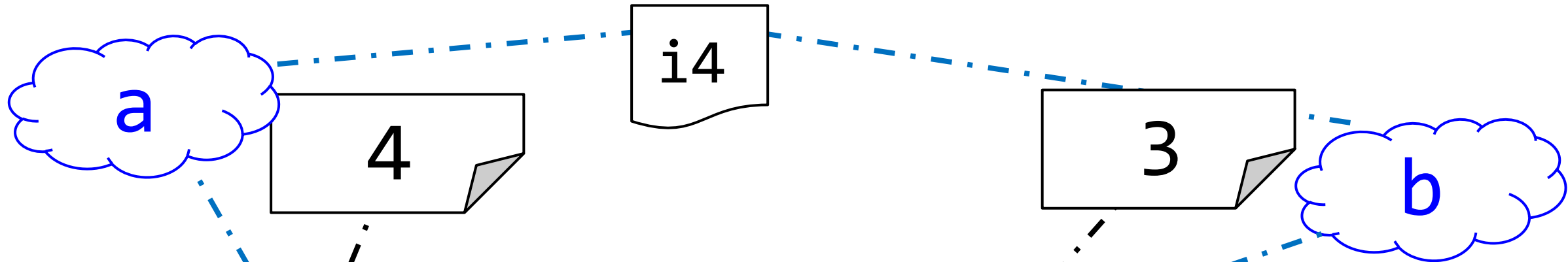
$[-7 \div 8]$

0100

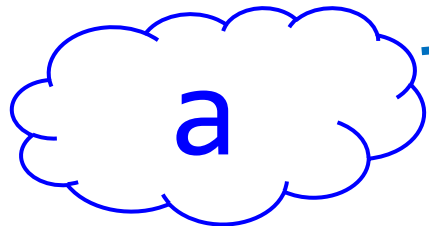




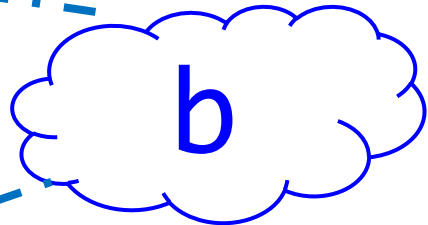




1100100000111001100111



i4



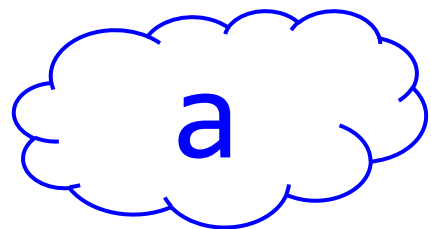
1100100001110011001111



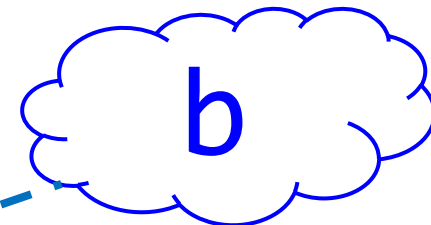
&a



&b



i4*

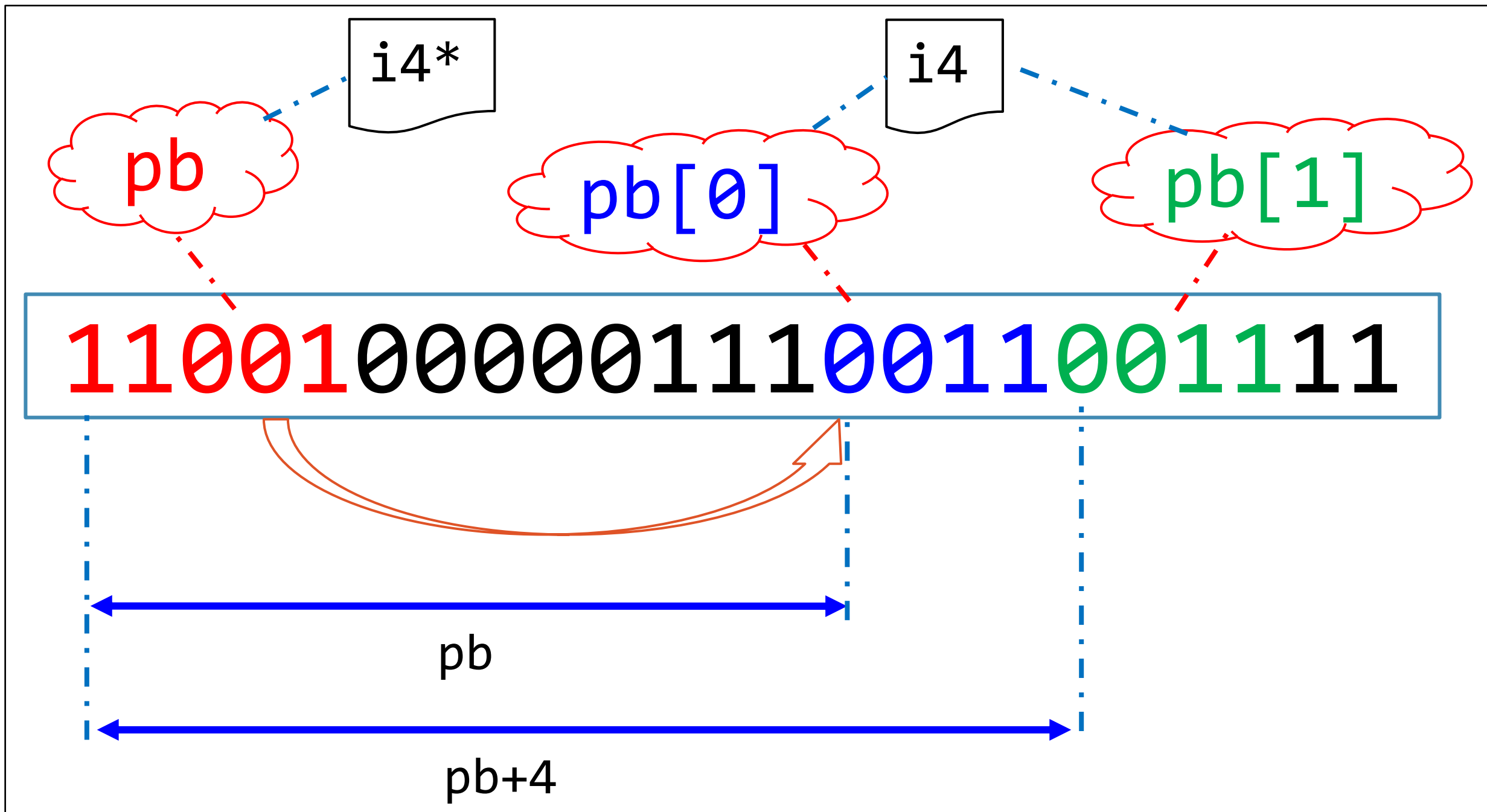


11001000001110011001111

&a

points-to

&b



Минимум про указатели

- На указатель можно взять указатель

```
int *px = &x; int **ppx = &px; int ***pppx = &ppx;
```

- К указателям можно прибавлять и вычитать целые числа

```
px = py + 2; px -= 3;
```

- Разыменование указателя мало чем отличается от его индексации

```
assert (*x == x[0]); assert(*(x+i) == x[i]);
```

- Поскольку сложение коммутативно, нет разницы между $x[2]$ и $2[x]$ просто второй способ записи никто не использует
- Минимальная адресуемая ячейка в C это char

arr[0]

arr[2]

arr[3]

1100100001110011001111

arr[5]

Минимум про массивы

- Название массива можно употреблять как адрес его первого элемента

```
int arr[5]; assert(arr == &arr[0]);
```

- К массиву нельзя ничего прибавить или присвоить, только к элементам

```
arr += 3; // ошибка, но 'arr[1] += 3' ok
```

- Указатель можно использовать для доступа в массиве

```
int *p = arr; p += 3; p -= 1; assert(*p == arr[2]);
```

- Выход за границы массива это серьёзная ошибка

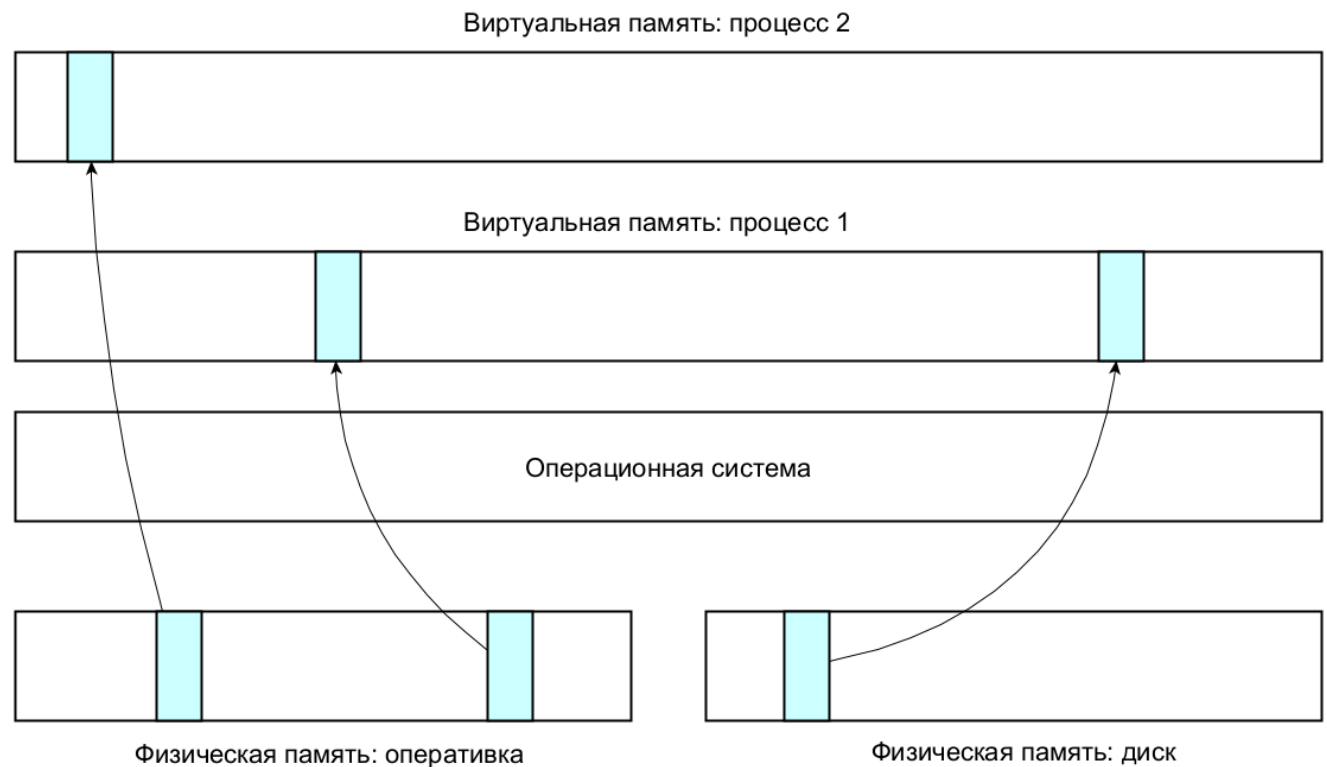
```
p += 10; printf("%d\n", *p); // печатает что угодно
```

Обсуждение: модель и реальность

- В абстрактной модели языка всё выглядит хорошо.
- Но как эта модель связана с реальностью?

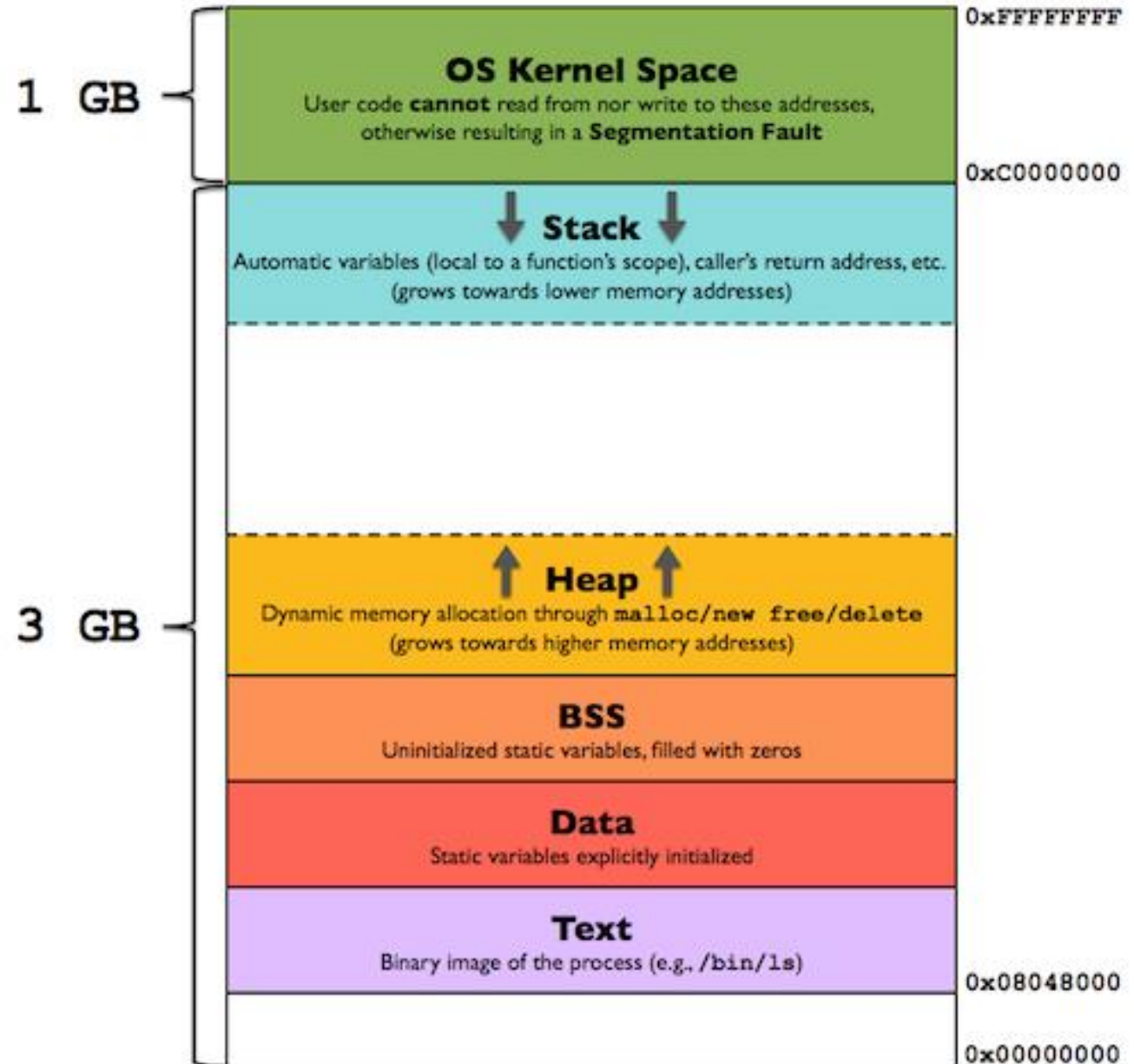
Виртуальная память

- Вашей 64-битной программе выделен весь диапазон адресов от 0 до $2^{64} - 1$.
- Такой же диапазон выделен каждому процессу.
- Связь виртуальной и физической памяти обеспечивает аппаратура и **операционная система**.



Виды памяти

- Runtime stack (стек исполнения)
 - локальные переменные
 - аргументы функций
 - служебная информация (адреса возврата)
- Глобальные данные
 - глобальные переменные
 - тела функций
 - всё, что видно для objdump
- Heap (куча)
 - всё самое интересное



Три вида памяти в вашей программе

- Программа, содержащая все три вида переменных

```
int x[100];  
int main () {  
    int y[100] = {0};  
    int *pz = calloc(100, sizeof(int));  
    free(pz);  
    return 0;  
}
```

Три вида памяти в вашей программе

- Глобальная память: переменная выделяется в data/rodata

`int x[100];`  глобальный массив

```
int main () {  
    int y[100] = {0};  
    int *pz = calloc(100, sizeof(int));  
    free(pz);  
    return 0;  
}
```

Три вида памяти в вашей программе

- Куча: переменная явно выделяется вызовом `calloc` или `malloc`

```
int x[100];
```

```
int main () {
```

```
    int y[100] = {0};
```

```
    int *pz = calloc(100, sizeof(int));
```



массив в куче

```
    free(pz);
```

```
    return 0;
```

```
}
```

Три вида памяти в вашей программе

- Куча: переменная явно освобождается вызовом free

```
int x[100];
```

```
int main () {
```

```
    int y[100] = {0};
```

```
    int *pz = calloc(100, sizeof(int));
```

```
    free(pz);
```

```
    return 0;
```

```
}
```



явное освобождение

Три вида памяти в вашей программе

- Стек: переменная выделяется в **стековом фрейме**

```
int x[100];
```

```
int main () {
```

```
    int y[100] = {0};
```



массив на стеке

```
    int *pz = calloc(100, sizeof(int));
```

```
    free(pz);
```

```
    return 0;
```

```
}
```

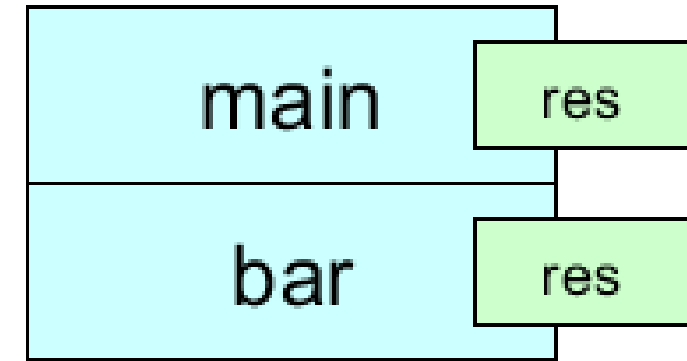
Почему стек называется стеком

```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```

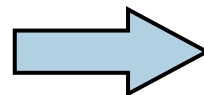


Почему стек называется стеком

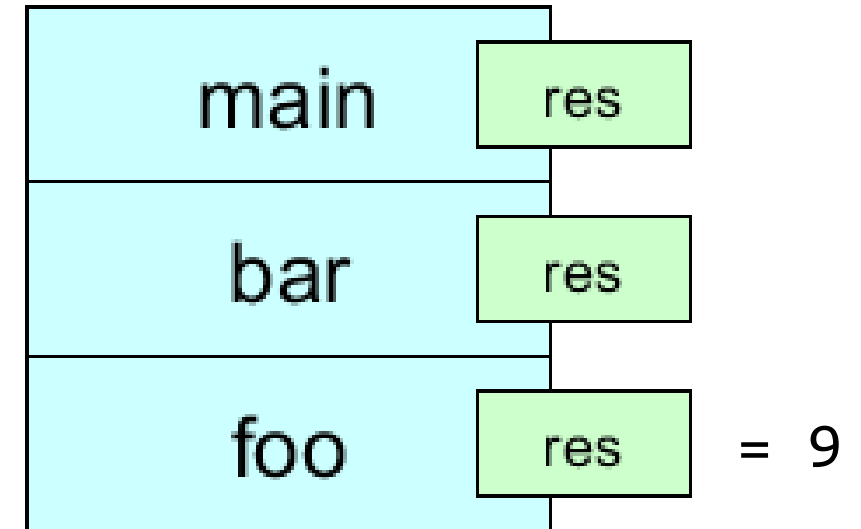
```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



Почему стек называется стеком

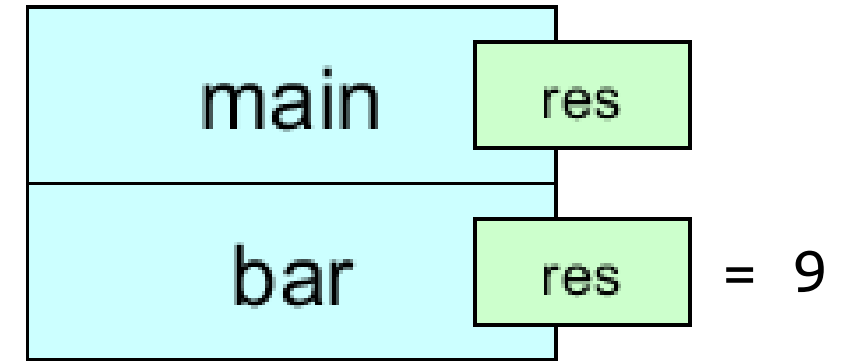
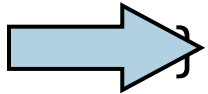


```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



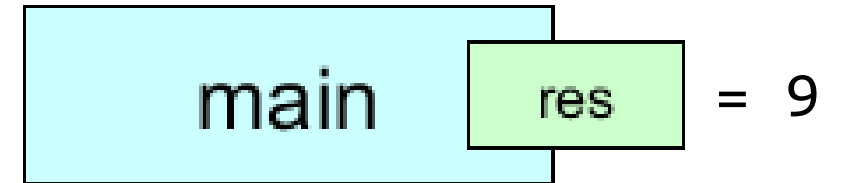
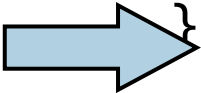
Почему стек называется стеком

```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



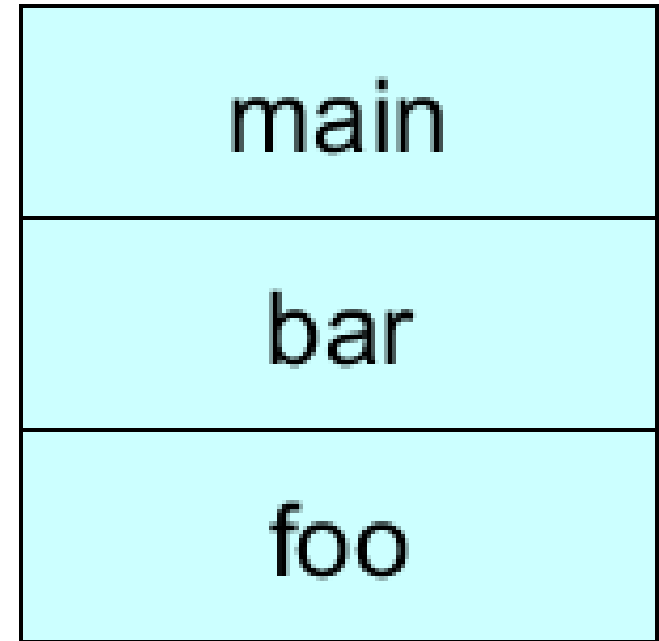
Почему стек называется стеком

```
int foo (int x, int y) {  
    int res = x + y;  
    return res;  
}  
  
int bar (int x) {  
    int res = foo (x, x * 2);  
    return res;  
}  
  
int main() {  
    int res = bar(3);  
    return res;  
}
```



Почему стек называется стеком

- Вызовы функций образуют естественную LIFO структуру (собственно стек)
- Этот стек также называется стеком исполнения (**runtime stack**)
- Все переменные, которые определены локально внутри функции, занимают её **стековый фрейм**
- Они также называются **автоматическими переменными**



Проблемы с временем жизни

- Функция возвращает адрес автоматической переменной

```
int *foo(int x, int y) {  
    int res = x + y;  
    return &res;  
}
```

- Этот адрес используется, но того места на стеке, куда он указывал, уже не существует

```
int main() {  
    int *p = foo(0, 42);  
    printf("%d\n", *p); // oops!  
}
```

- Никогда так не делайте

Работа с кучей: malloc и free

- Функция возвращает адрес в куче

```
int *foo(int x, int y) {  
    int *res = malloc(sizeof(int));  
    *res = x + y;  
    return res;  
}
```

- Этот адрес может быть свободно использован в вызывающей функции

```
int main() {  
    int *p = foo(0, 42);  
    printf("%d\n", *p); // ok  
    free(p);  
}
```

Контроль утечек: valgrind

```
$ gcc -g memleak.c  
$ valgrind ./a.out
```

```
==957== LEAK SUMMARY:  
==957==      definitely lost: 4 bytes in 1 blocks
```

```
$ valgrind --leak-check=full ./a.out
```

```
==985== 4 bytes in 1 blocks are definitely lost in loss record 1  
==985==      at 0x4848899: malloc  
==985==      by 0x109184: foo (memleak.c:5)  
==985==      by 0x1091D2: main (memleak.c:12)
```

Минимум о динамической памяти

- Выделение `malloc`(размер) и `calloc`(количество, размер элемента)
- Специальный тип `void*` означает "указатель на нетипизированную память" приводится к любому указателю

```
void *mem = malloc(10 * sizeof(double)); // 10 doubles
double *pd = (double *) mem;
int *pi = (int *) calloc(1000, sizeof(int));
```

- С возвращённым указателем, можно работать как с массивом

```
assert (*pi == pi[0]); pi[100] = 2; pd[3] = 4.0;
```

- Освобождение `free`(указатель) при этом `free(NULL)` это ок
- ```
free(mem); // или free(pd) но не вместе
```



# Проверка результата

- Исчерпание кучи гораздо менее болезненно, чем исчерпание стека. В этом случае malloc просто вернёт NULL и это можно явно проверить.

```
int *foo(int x, int y) {
 int *res = malloc(sizeof(int));
 assert (res != NULL); // или более сложная обработка
 *res = x + y;
 return res;
}
```

- Допустимо не приводить явно, например строка выше эквивалентна

```
int *res = (int *) malloc(sizeof(int));
```

# Problem M1: исследование кучи

- Напишите программу которая будет выделять всё больше памяти в куче с помощью malloc.
- Сделайте побольше итераций.
- На какой итерации malloc вернёт NULL?

# Problem M2: исследование стека

- Напишите программу, которая будет рекурсивно вызывать функцию, создающую большой массив на стеке (например 10000 байт)
- За каждый уровень рекурсии печатайте на экран следующее число
- Сколько стека вы сможете использовать до возникновения проблем?

# Область видимости и время жизни

- Глобальная переменная: область видимости везде время жизни всегда.
- Переменная на стеке: область видимости в пределах блока, время жизни в пределах блока.

```
int foo () {
 int x;
 int *py;
 {
 int y = 5;
 py = &y;
 }
 x = *py; // BOOM
}
```

конец времени жизни y

конец времени жизни x

# Область видимости и время жизни

- Глобальная переменная: область видимости везде время жизни всегда
- Переменная в куче: область видимости в пределах блока, время жизни до явного освобождения

```
int foo () {
 int x;
 int *py;
 {
 int* pz = malloc(sizeof(int)); *pz = 5;
 py = pz;
 }
 x = *py; // ОК
}
```

 утечка памяти под py

# Статические переменные

- Статическая переменная это глобальная переменная с ограниченной областью видимости

```
int foo() {
 static int x = 0; // время жизни: всегда
 x += 1;
 printf("%d\n", x);
}
```

```
foo(); // на экране 1
foo(); // на экране 2
foo(); // на экране 3
```