3. Алгоритмические языки программирования (на примере C++)

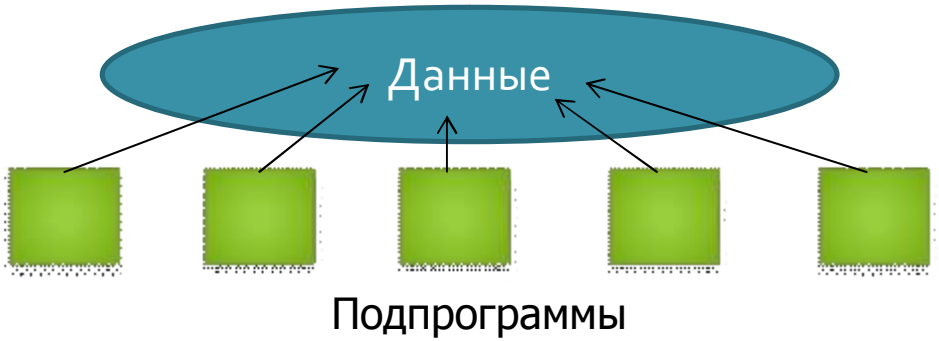
3.1. Языки высокого уровня: общая характеристика.
3.2. Лексические основы языка высокого уровня (C++).
3.3. Данные как объекты памяти ЭВМ.
3.4. Операторы C++.

3.1. Языки высокого уровня: общая характеристика

История развития языков высокого уровня

По мнению специалистов языки программирования прошли несколько поколений своего развития:

1. Первое поколение: 1954-1958 годы. FORTRAN, ALGOL-58 и др. Предназначены были в основном для упрощения программирования математических вычислений. Программы состояли из подпрограмм и глобальных данных, доступных всем подпрограммам.

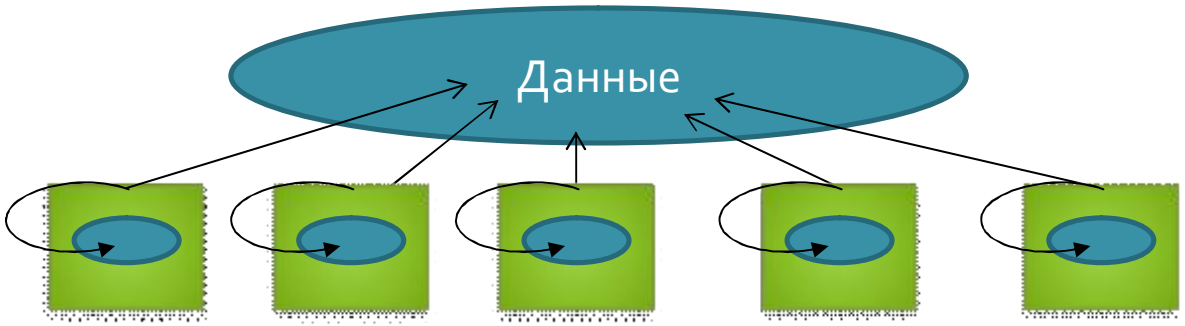


3.1. Языки высокого уровня: общая характеристика

2. Второе поколение: 1959 -1961 годы. FORTRAN II, ALGOL-60, COBOL, Lisp и др. Развитие алгоритмических абстракций. Появились:

- процедуры со своими локальными параметрами;
- раздельная компиляция;
- обработка списков, указатели и др.

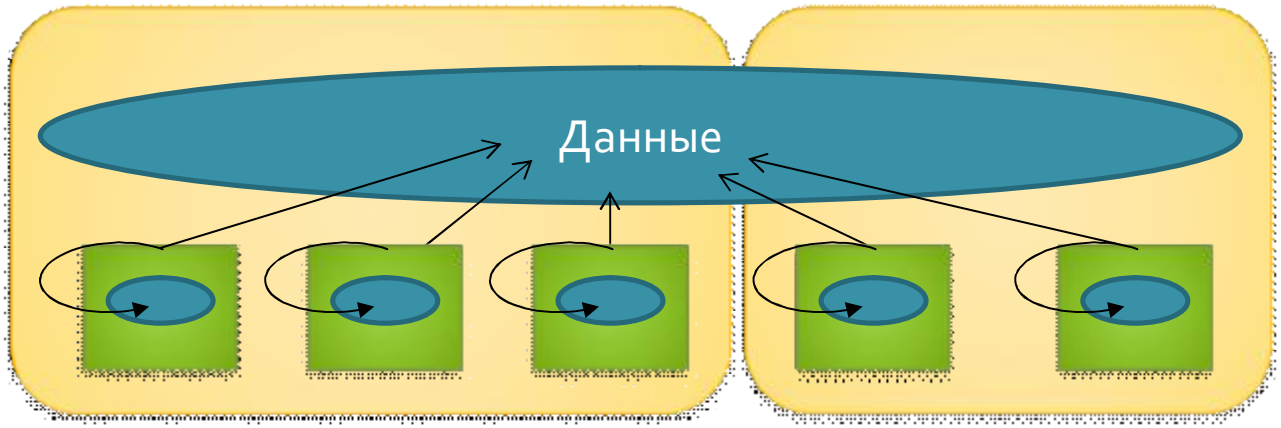
Языки второго поколения позволяли создавать универсальные подпрограммы и библиотеки.



3.1. Языки высокого уровня: общая характеристика

3. Третье поколение: 1962 -1970 годы. PL/1, ALGOL-68, Pascal, Simula и др. Абстрактные типы данных. Появились:

- модули как элемент разделения подпрограмм;
- абстрактные типы данных (структуры).

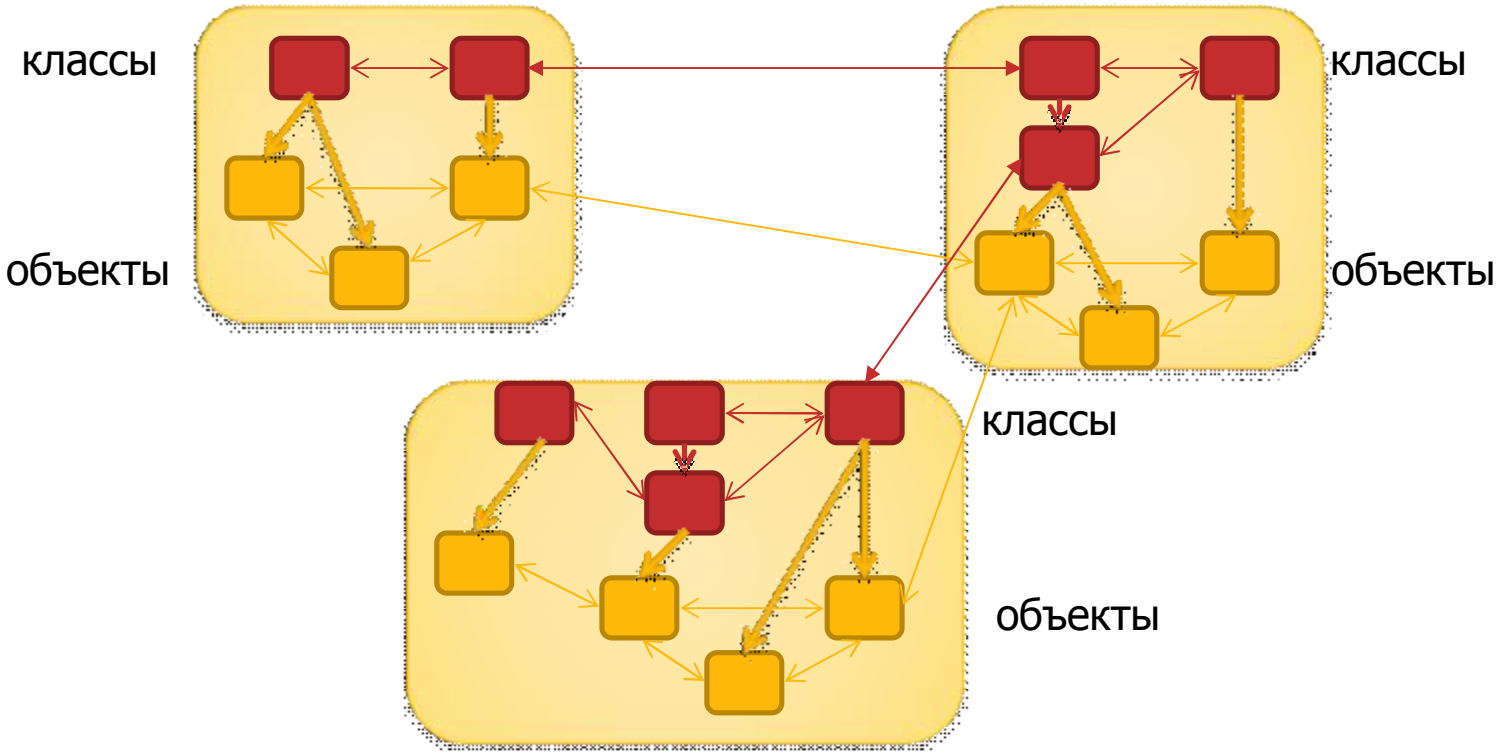


Подпрограммы и модули

3.1. Языки высокого уровня: общая характеристика

4. Четвертое поколение: 1970 -1980 годы. Smalltalk, Object Pacsal, C++, CLOS, Ada, Eiffel и др. Объектно-ориентированные языки программирования.

Основным элементом языков служит модуль, составленный из логически связанных классов и объектов.



3.1. Языки высокого уровня: общая характеристика

Разработка программных продуктов с помощью языков высокого уровня

Преобразование исходного кода программы, написанной на языке высокого уровня, в машинные коды называется *трансляцией*.

Имеется три основных способа трансляции:

- интерпретирование;
- ассемблирование;
- компилирование.

3.1. Языки высокого уровня: общая характеристика

При *интерпретировании* программа преобразуется в машинные коды «на лету», т.е. в ходе выполнения. Интерпретируемая программа обрабатывается *программой-интерпретатором*. По сути, работает именно программа-интерпретатор, а интерпретируемая программа предоставляет ей инструкции.

Примеры: Java-апплеты, выполняемые виртуальной машиной Java, программы на языке Matlab, выполняемые самой программой Matlab, программы на Visual Basic for Application (VBA), выполняемые в Microsoft Office.

Преимущества:

- гибкость (программу можно легко модифицировать);
- межплатформенная совместимость;
- безопасность.

Недостатки:

- низкая скорость выполнения;
- необходимость в интерпретаторе.

3.1. Языки высокого уровня: общая характеристика

Ассемблируемые и *компилируемые* программы транслируются на машинный язык до их использования. Поэтому они работают быстрее интерпретируемых, но не обладают их преимуществами – гибкостью, совместимостью и безопасностью.

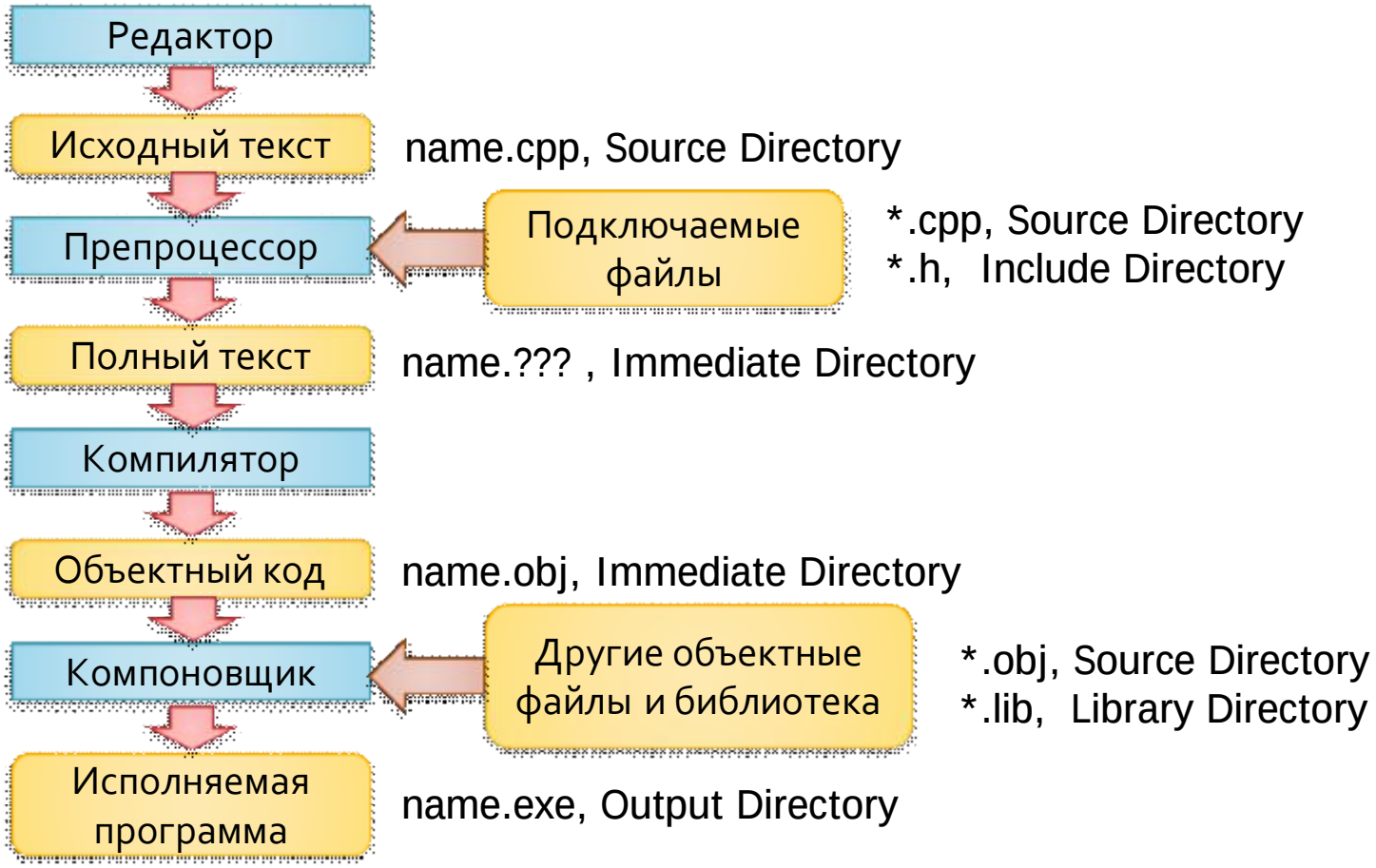
Отличие ассемблера (программы-транслятора с языка ассемблера) от компилятора состоит в способе преобразования исходного кода.

В языке ассемблера программист записывает эквивалент каждой машинной инструкции, следовательно, ассемблер выполняет строго фиксированные действия по трансляции команд и поэтому является достаточно простой программой.

Компиляторы обладают более «творческими» возможностями, самостоятельно решая, какие именно машинные команды создавать и как использовать ресурсы (память, регистры) для реализации тех шагов, которые заложены в программе на языке высокого уровня.

3.1. Языки высокого уровня: общая характеристика

Процесс создания компилируемого приложения (C++)



3.1. Языки высокого уровня: общая характеристика

Препроцессор выполняет множество функций, связанных с подготовкой исходного текста к компиляции (*Подбельский, Язык C++*):

- все системно-зависимые обозначения (например, системно-зависимый индикатор конца строки) перекодируются в стандартные коды;
- каждая пара из символов '\' и "конец строки" убираются, и тем самым следующая строка исходного файла присоединяется к строке, в которой находилась эта пара символов;
- в тексте распознаются директивы препроцессора, а каждый комментарий заменяется одним символом пустого промежутка;
- выполняются директивы препроцессора и производятся макроподстановки;
- ESC-последовательности в символьных константах и символьных строках, например, '\n' заменяются на их эквиваленты (на соответствующие числовые коды);
- смежные символьные строки конкатенируются, т.е. соединяются в одну строку.

3.1. Языки высокого уровня: общая характеристика

Директивы препроцессора (начинаются с #):

#define – замены в тексте (определение макросов и констант):

#define K, 40 – везде далее в тексте программы «K» будет заменено на 40

#define max(a,b) (a < b ? b : a) – определение макроса

#include – включение текстов из файлов:

#include **<**file1.h**>** - включение файла file1.h из директории INCLUDE;

#include **“**file2.h**”** - включение файла file1.h из директории пользователя

3.1. Языки высокого уровня: общая характеристика

Другие директивы препроцессора (*Подбельский, Язык C++*):

Директива #undef отменяет действие команды #define, которая определила до этого имя препроцессорного идентификатора.

Директива #if и ее модификации #ifdef, #ifndef совместно с директивами #else, #endif, #elif позволяют организовать условную обработку текста программы. Условность состоит в том, что компилируется не весь текст, а только те его части, которые так или иначе выделены с помощью перечисленных директив.

Директива #line позволяет управлять нумерацией строк в файле с программой. Имя файла и начальный номер строки указываются непосредственно в директиве #line.

Директива #error позволяет задать текст диагностического сообщения, которое выводится при возникновении ошибок.

Директива #pragma вызывает действия, зависящие от реализации.

Директива # ничего не вызывает, так как является пустой директивой, т.е. не дает никакого эффекта и всегда игнорируется.

3.1. Языки высокого уровня: общая характеристика

Компоновка

Объектные модули, связываемые компоновщиком (linkerом) могут, вообще говоря, быть образованы различными компиляторами или ассемблером. То есть, исходный текст для этих модулей может быть написан на различных языках программирования. При этом возникает проблема интерфейса или взаимодействия модулей, связанная с

- соглашениями об именах – разные компиляторы и ассемблеры по разному называют переменные и подпрограммы (различают или нет регистры, вводят различные префиксы и т.д.) в объектных модулях;
- порядком передачи параметров процедурам и возврата результата (как передаются параметры: через стек или регистры, в каком порядке, как возвращаются результаты).

Все эти вопросы должны быть решены программистом (заданием опций компилятора).

3.1. Языки высокого уровня: общая характеристика

Пример: программа на C должна вызывать подпрограмму на ассемблере:

x = AbsValue(y);

Выполняемый при вызове код на ассемблере:

push Y; занести в стек Y

call _AbsValue; вызвать подпрограмму

;(компилятор добавляет префикс _)

add sp,2; очистка стека (в C/C++ - дело вызывающей стороны!)

mov X, ax; сохранить результат в X (в C/C++ - результат ; передается через регистр A!)

3.1. Языки высокого уровня: общая характеристика

Подпрограмма на ассемблере:

```
_AbsValue PROC near
        push bp
        move bp, sp
        mov ax, [bp+4]; ax = значение параметра
        cwd; dx = 0 при ax>0 или -1 (0xffff) при ax<0
        xor ax,dx; ax = ax при dx==0 или ~ax при dx==0xffff
        sub ax,dx; ax = ax-0 при dx==0 или ax = ax+1при dx==-1
        pop bp
        ret
_AbsValue ENDP
```

3.2. Лексические основы языка высокого уровня (C++)

Лексемы (составляющие) языка C++:

- *идентификаторы (имена переменных, функций, типов);*
- *константы;*
- *знаки операций;*
- *разделители.*

3.2. Лексические основы языка высокого уровня (C++)

Идентификаторы – последовательности из букв латинского алфавита, десятичных цифр, а также символов подчеркивания, начинающихся не с цифры.

Прописные и строчные цифры различаются!

Служебные слова – идентификаторы, зарезервированные для специального использования:

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	typeid
const	goto	short	union
continue	if	signed	unsigned
default	inline	sizeof	virtual
delete	int	static	void
do	long	struct	volatile
			while

3.2. Лексические основы языка высокого уровня (C++)

Константы – представляют изображение фиксированного числового, символьного или строкового значения.

Целые константы бывают десятичными, восьмеричными и шестнадцатеричными*:*

Диапазоны значений констант			*Тип данных*
десятичные	*восьмеричные*	*шестнадцатеричные*	
от 0 до 32767	от 00 до 077777	от 0x0000 до 0x7FFF	int
-	от 0100000 до 0177777	от 0x8000 до 0xFFFF	unsigned int
от 32768 до 2147483647	от 0200000 до 017777777777	от 0x10000 до 0x7FFFFFFF	long
от 2147483648 до 4294967295	от 020000000000 до 037777777777	от 0x80000000 до 0xFFFFFFFF	unsigned long
> 4294967295	> 037777777777	> 0xFFFFFFFF	ошибка

3.2. Лексические основы языка высокого уровня (C++)

Вещественные константы

12.5, 13., .12, 1.15e-2

автоматически принимают тип **double.**

Программист может изменить тип константы, применяя суффиксы l(L), u(U), f(F):

64 – int	0.5 – double
64L – long	0.5F – float
64UL – unsigned long	0.5L – long double

Тип данных	*Размер, бит*	*Диапазон значений*
float	**32**	**от 3.4E-38 до 3.4E+38**
double	**64**	**от 1.7E-308 до 1.7E+308**
long double	**80**	**от 3.4E-4932 до 1.1E+4932**

3.2. Лексические основы языка высокого уровня (C++)

Перечисления – тип, определяющий набор целых констант, обозначаемых произвольными идентификаторами. Например, определение

```
enum DaysOfWeek {Sun, Mon, Tue, Wed, Thurs=10, Fri, Sat};
```

DaysOfWeek задает набор констант, соответствующих дням недели. По существу это обычные целочисленные константы (типа int), которым приписаны уникальные и удобные для использования обозначения. Значения констант по умолчанию начинаются с нуля и последовательно увеличиваются на единицу: Sun = 0, Mon = 1 и т.д.

Перечисления используются, например, для перегрузки функций. В этом случае несколько функций, имеющих одинаковые имена, будут различаться типами формальных параметров, в качестве которых используются перечислимые типы.

3.2. Лексические основы языка высокого уровня (C++)

Символьные константы – один или два символа, заключенные в **‘ ‘**, имеют тип char или int: **‘a’, ‘ab’**.

Строковые константы – последовательность символов, заключенная в **“ “**:**“String”**.

Строки размещаются в памяти, причем компилятор автоматически добавляет в конец строки нулевой байт.

Символьные и строковые константы могут включать в себя

ESC-последовательности – управляющие символы, начинающиеся с ****, например, перевод строки **\n**:

cout << “It is string.\nIt is new string!”;

It is string.

It is new string!

3.2. Лексические основы языка высокого уровня (C++)

ESC- последовательности

Изображение	*Внутренний код*	*Обозначаемый символ (название)*	*Реакция или смысл*
\a	0x07	bel (audible bell)	Звуковой сигнал
\b	0x08	bs (backspace)	Возврат на шаг (забой)
\f	0x0C	ff (form feed)	Перевод страницы (формата)
\n	0x0A	lf (line feed)	Перевод строки (новая строка)
\r	0x0D	cr (carriage return)	Возврат каретки
\t	0x09	ht (horizontal tab)	Табуляция горизонтальная
\v	0x0B	vt (vertical tab)	Табуляция вертикальная
\\	0x5C	\ (backslash)	Обратная косая черта
\'	0x27	' (single quote)	Апостроф (одинарная кавычка)
\"	0x22	" (double quote)	Двойная кавычка
\?	0x3F	? (question mark)	Вопросительный знак
\000	000	Любой (octal number)	Восьмеричный код символа
\xhh	0xhh	Любой (hex number)	Шестнадцатеричный код символа

3.2. Лексические основы языка высокого уровня (C++)

Знаки операций

- обеспечивают формирование и вычисление выражений;
- интерпретируются в зависимости от контекста:

  a*b; ***//*** бинарная операция умножения

  p;** **// унарная операция обращения по адресу

  c*=5; ***//*** «часть» знака операции ***=**: **c = c*5**

  int *h; ***//*** а это вообще не знак операции, а разделитель!
- могут быть перегружены (переопределены) для объектов классов:

  cout << 10;

  // «настоящий вызов»:

  // **cout.operator<<(10);**

  // где cout – объект класса ostream

3.2. Лексические основы языка высокого уровня (C++)

Унарные операции:

& получение адреса

***** обращение по адресу

- изменение знака

~ поразрядное инвертирование

! логическое НЕ (из «неноль» делает ноль, из ноля – единицу)

++ инкремент (увеличение на единицу)

-- декремент (уменьшение на единицу)

Операции инкремента и декремента имеют префиксную и постфиксную формы:

```
int a = 5; cout << --a<< '\n'; // 4
cout << a++ <<'\n'; // 4
cout << a; // 5
```

sizeof вычисление размера переменной в байтах

3.2. Лексические основы языка высокого уровня (C++)

Бинарные операции:

+,- аддитивные

***,/,%** (остаток от деления) мультипликативные

```
int a = 5, b = a%3;
cout << b; // 2
```

<<,>> сдвига

```
int a = 5, b = a<<2, c = a>>1;
//  0...0101-> 0…010100, 0…0101-> 0…0010
cout << b << ' ,' << c; // 20,2
```

&, |, ^ поразрядные конъюнкция (И), дизъюнкция (ИЛИ) и исключающее ИЛИ

```
int a = 5, b = 3;
cout << a&b << ' ,' << a|b << ' ,' << a^b; // 1,7,6
```

3.2. Лексические основы языка высокого уровня (C++)

<,>,<=,>=,==,!= операции отношения (результат TRUE/FALSE)

&&, || логические И и ИЛИ (результат TRUE/FALSE)

=,*=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^= присваивание

E1 op= E2 эквивалентно E1 = E1 op E2

3.2. Лексические основы языка высокого уровня (C++)

Операции выбора компонентов

.(точка) прямой выбор (по имени объекта и компонента)

-> выбор через указатель на объект и имя компонента

```
struct S {int i, float f;};
S s, *p = &s;
s.i = 13;
p->f =1.4; //аналогично (*p).f = 1.4
```

.* выбор по имени объекта и указатель на компонент

->* выбор через указатель на объект и указатель на компонент

:: - указание области видимости

- *используются редко*

3.2. Лексические основы языка высокого уровня (C++)

Скобки

() вызов функции fun(a,b)

Вызов функции – это результат применения к ее имени операции «круглые скобки»!

[ ] обращение к элементу массива M[i]

*Фактически конструкция M[i] эквивалентна *(M+i).*

M – константный указатель на первый элемент массива, i – целочисленный индекс. Сложение производится по правилам работы с указателями, т.е. фактически к адресу первого элемента прибавляется i, помноженное на размер элемента массива в байтах.

3.2. Лексические основы языка высокого уровня (C++)

Условная операция

x < 0 ? –x:x; ***//*** вычисление модуля

Операция приведения типа

float b = 64.4;

cout << (int)b; ***//*** 64

// - «обычная» форма

cout << char(b); ***//***A

// **-** «функциональная» форма, ограниченное использование

Операция запятая

int d;

cout << (d=3, d*2); ***//6***

3.2. Лексические основы языка высокого уровня (C++)

Операции динамического распределения памяти new и delete

int *p; // объявление указателя

p = new int(14); //создание динамической переменной и ее

delete p;

p = new int[10]; //создание массива из 10 целых типа int

delete [] p; // удаление динамического массива

3.2. Лексические основы языка высокого уровня (C++)

Ранги операций и их ассоциативность

Ранги операций определяют порядок их выполнения, ассоциативность – «направление» выполнения

Ранг	*Операции*	Ассоциативность
1	() [] -> :: .	→
2	! ' ~ + - ++ -- & * (тип) sizeof new delete тип() (функциональное преобразование типа)	←
3	.* ->*	→
4	* / % (мультипликативные бинарные операции)	→
5	+ - (аддитивные бинарные операции)	→
6	<< >>	→
7	< <= >= >	→
8	== !=	→
9	&	→
10	^	→
11	|	→
12	&&	→
13	||	→
14	?: (условная операция)	←
15	= *= /= %= += -= &= ^= |= <<= >>=	←
16	, (операция "запятая")	→

3.2. Лексические основы языка высокого уровня (C++)

Пример:

int a=1, b = 2, *p = &b, M[ ] = {3,4,5};

a += *p + M[1] - M[2]/2;

1) a += *p + 4 - M[2]/2;

2) a += *p + 4 - 5/2;

3) a += 2 + 4 - 5/2;

4) a += 2 + 4 - 2;

5) a += 6 - 2;

5) a += 4; // a = 1+4 = 5

3.2. Лексические основы языка высокого уровня (C++)

Разделители (знаки пунктуации)

() :

- изменяет порядок выполнения операций

  a =5*(3+2);
- используется при описании и определении функций

  float fun(int k);
- используется для описания операторов условий и циклов

  if(x>1)…

  while (x>0)…

  for (int i = 0; i<5; i++) …

3.2. Лексические основы языка высокого уровня (C++)

{} :

- определяет тело функции, класса

  int fun() {…} class C {…};

- определяет составной оператор блок

  if (a<1) {a+=1; x = 2;}

  if (a<1) {int i = 0; a+=1; x = 2;…}

- используется для оформления списков инициализации

  int M[ ] = {1,2,3,4};

,(запятая) разделяет элементы списков

; разделяет операторы

: определение метки

L: x+=2;

goto L;

3.2. Лексические основы языка высокого уровня (C++)

… (многоточие) определяет переменный список параметров функции

fun(int a, …);

***** используется при описании указателей

int *p;

= используется в описаниях при инициализации

int a = 10;

директивы препроцессора

#include <iostream.h>

& используется при описании ссылок

int& fun(int& a);

Порядок использования (чтения) разделителей аналогичен порядку использования (чтения) знаков операций.

3.2. Лексические основы языка высокого уровня (C++)

Порядок использования (чтения) разделителей аналогичен порядку использования (чтения) знаков операций.

Примеры:

int *M[10];

1) **int *M[10];** M - массив из 10-ти

2) **int *M[10];** указателей

3) **int *M[10];** на int

***M[2] = 4;**

1) ***M[2]=4;** обращаемся к 3-ему элементу массива

2) ***M[2]=4;** читаем адрес и обращаемся по нему

3) ***M[2]=4;** записываем туда 4

3.2. Лексические основы языка высокого уровня (C++)

int (*M)[10];

1) **int (*M)[10];** M – указатель на

2) **int (*M)[10];** массив из 10 элементов

3) **int (*M)[10];** типа int

(*M)[2] = 4;

1) **(*M)[2] = 4;** обращаемся по адресу из M и получаем массив

2) **(*M)[2] = 4;** обращаемся к 3-ему элементу этого массива

3) **(*M)[2] = 4;** записываем туда 4

3.3. Данные как объекты памяти ЭВМ

Объект памяти – объект, размещенный в памяти, только такой объект может находиться в левой части операции присваивания.

Для каждого объекта памяти задается ***тип***, который определяет:

- количество памяти, отводимой объекту;
- интерпретацию содержимого памяти;
- совокупность разрешенных операций.

Типизация – контроль типов компилятором, заключается в запрете использовать переменные одних типов вместо других и применять операции над объектами не разрешенные для данного типа.

3.3. Данные как объекты памяти ЭВМ

Типы данных:

- скалярные:
  - базовые (char, int, long и т.д.);
  - перечислимые константы;
  - указатели;
- агрегатные:
  - массивы;
  - структуры;
  - классы;
  - объединения;
- функции.

3.3. Данные как объекты памяти ЭВМ

Правила описания производных типов

Из базовых типов с помощью разделителей и ключевых слов struct, unit, class задаются производные типы: агрегатные и функции. При этом порядок использования разделителей аналогичен порядку использования знаков операций. Примеры:

```
int * M[10]; // массив указателей на данные типа int
int (*M)[10]; //указатель на массив с данными типа int
int* (*M)[10]; //указатель на массив указателей на данные типа int
int* fun(char); //прототип функции, принимающей данные типа char
                // и возвращающей указатель на данные типа int
int (*fun)(char); //указатель на функцию, принимающую данные типа
                  // char и возвращающей указатель на данные типа
    int
```

3.3. Данные как объекты памяти ЭВМ

Определение и описание объектов

Перед использованием объектов они должны быть определены или по крайней мере описаны.

Определение объекта по сути есть выделение ему памяти.
Общий вид:

[<s>] [<m>] <тип> <имя1> [=<инициализатор1>] ,

<имя2> [=<инициализатор2>] ,…;

s – спецификатор (static, register,…)

m – модификатор, например, const

инициализатор (необязателен), например

=1, = {1,2,3} (массив), ={ “John”, 234} (структра)

3.3. Данные как объекты памяти ЭВМ

Описание объекта делает имена переменных или сигнатуры функций известными компилятору (сами определения будут сделаны позднее или в другом модуле).

Примеры:

```
extern int var; //определение внешней переменной

int fun(float, char*); //прототип функции
```

В C/C++ принято помещать объявления общих переменных и функций в файлы заголовков (h-файлы) и подключать их в модулях, где они нужны, директивой include.

3.3. Данные как объекты памяти ЭВМ

Спецификация «пользовательских» типов

По настоящему пользовательские типы вводятся как классы и структуры, однако в наследство от языка С языку С++ предоставлена возможность задания «синонимов» посредством ключевого слова **typedef** (определить тип).

Пусть, например, необходимо описать список из 25 человек, каждый из которых имеет фамилию, имя, отчество:

char *list [25][3];

- массив из 25 массивов из трех указателей на char. Пусть, например, 13 член списка – Иванов Иван Иванович.

list[12][0]= “Ivanov”;

list[12][1] = “Ivan”;

list[12][2] = “Ivanovich”;

3.3. Данные как объекты памяти ЭВМ

Можно описать список более информативно, введя новое имя типа:

typedef char * FIO[3];

- FIO – «новый тип» -массив из трех указателей на char.

FIO list[25];

list – массив из 25 элементов типа FIO, т.е. из трех указателей типа char .

3.3. Данные как объекты памяти ЭВМ

Преобразования типов

Часто при выполнении вычислений и записи выражений требуется преобразование типов операндов:

float a = 2.5*4;

//2.5 (double)*4(int) = 10 (float)

Преобразования типов могут быть ***безопасными***, например,

char→int, int, float →double, если при их вычислении не изменяется ни интерпретация, ни точность представления.

Безопасные преобразования всегда обратимы.

Опасные преобразования:

unsigned int →int:

long→float; // может привести к потери точности

Опасные преобразования могут привести к потере точности.

3.3. Данные как объекты памяти ЭВМ

Часть преобразований выполняется компилятором самостоятельно. Это касается безопасных преобразований, а также опасных, если в данном контексте они не приводят к потере информации. Иначе компилятор может выдать предупреждение или даже сообщение об ошибке.

В других случаях преобразование типа должно выполняться программистом явно с помощью операторов преобразования типа

(тип)X; - каноническая форма

тип(X); - функциональная форма, применяется только для простых типов

Пример: вывести на экран число 155 как char, int, float:

```
cout << (char)155 << 155 << (float)155;
```

3.3. Данные как объекты памяти ЭВМ

Кроме типа, каждый объект характеризуется рядом ***атрибутов***:

- класс памяти;
- область действия идентификатора;
- видимость;
- продолжительность существования;
- тип компоновки.

3.3. Данные как объекты памяти ЭВМ

Класс памяти определяется местом размещения объекта (сегмент данных, стек, регистры).

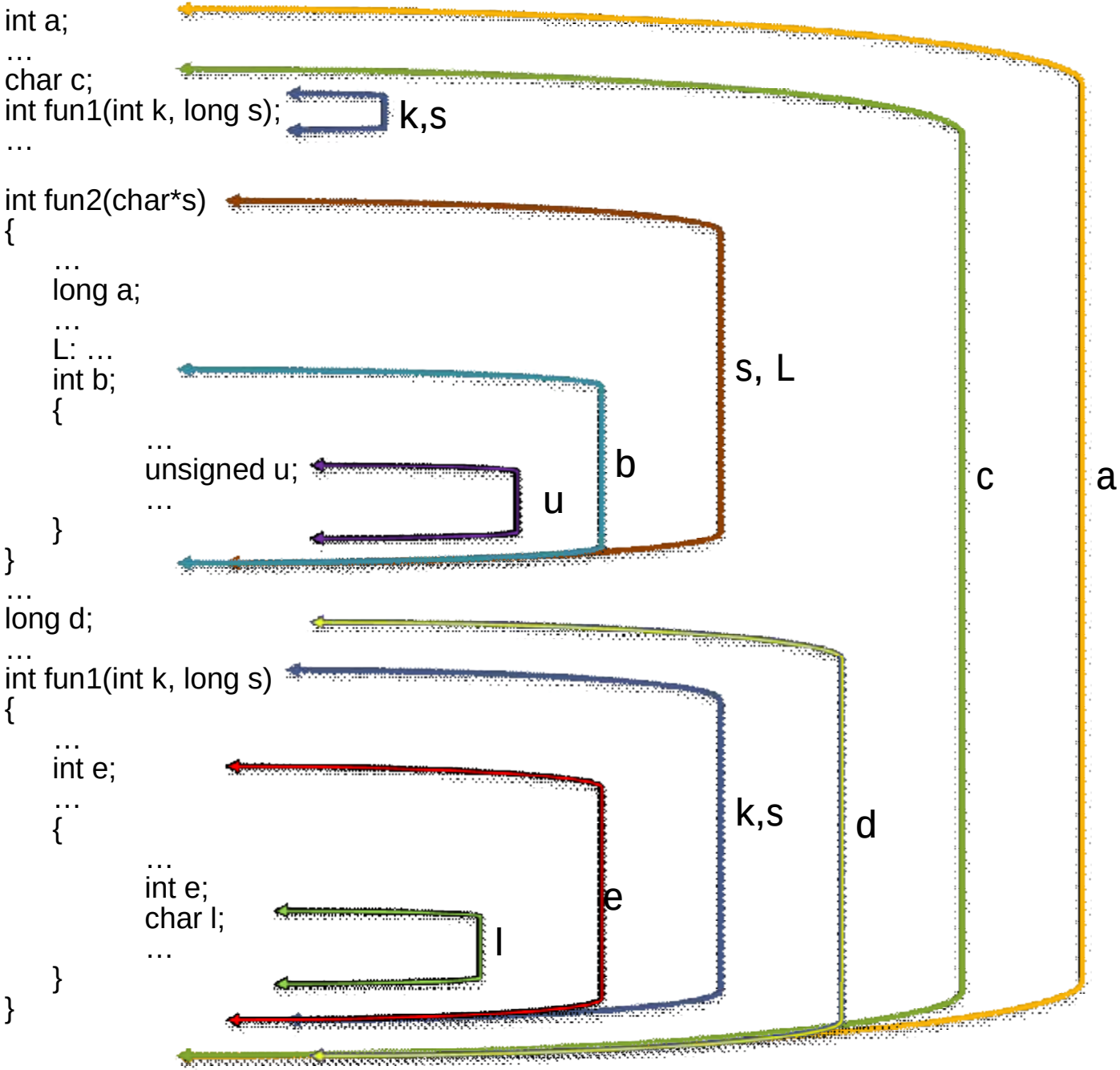
В основном класс памяти определяется компилятором по контексту (т.е. по месту определения переменных или функций):

- глобальные переменные – сегмент данных. Такие переменные называются *статическими*;
- локальные переменные функций – стек или регистры. Такие переменные называются *автоматическими*;
- *динамические* переменные – сегмент «кучи» (heap).

Повлиять на класс памяти можно с помощью спецификаторов *auto*, *register*, *static*. Однако эти спецификаторы не имеют «силы приказа» (исключение – *static:* внутри функции можно задать статическую переменную видимую только этой функции) .

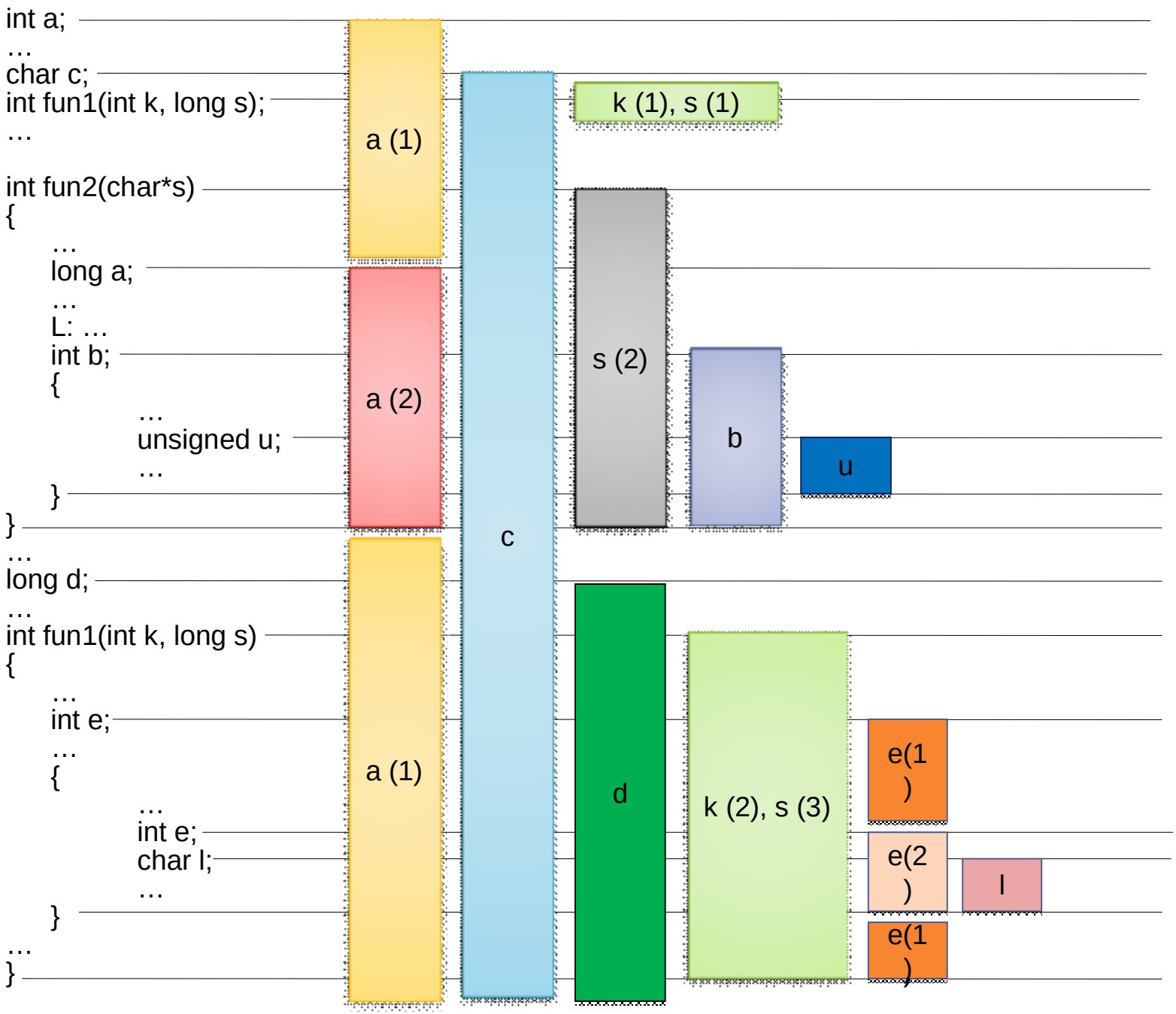
3.3. Данные как объекты памяти ЭВМ

Область действия идентификатора – часть программы, в которой по нему может быть получен доступ к ***какому-либо*** объекту



3.3. Данные как объекты памяти ЭВМ

Видимость объекта – часть программы, в которой по идентификатору возможен доступ ***именно к этому*** объекту



3.3. Данные как объекты памяти ЭВМ

Изменение видимости возможно с помощью операции ::

```
int a =10;
fun()
{
    float a = 20;
    cout << a; //20
    cout << '\n' << ::a; //10
}
```

3.3. Данные как объекты памяти ЭВМ

Продолжительность существования – период в течение которого идентификатору соответствует конкретный объект в памяти.

Продолжительность:

- локальная;
- статическая;
- динамическая.

3.3. Данные как объекты памяти ЭВМ

Локальная продолжительность существования характерна для локальных объектов, т.е. объектов, описанных внутри блока или функции (за исключением объектов, объявленных как static). Они всегда уничтожаются при выходе из блока или функции. Объекты с локальной продолжительностью, как правило, размещаются в стеке или регистрах (по возможности) и должны быть явно инициализированы, иначе их значения непредсказуемы.

Статическая продолжительность существования определена для глобальных объектов (переменных, определенных вне функций и для самих функций). Такие объекты «живут» пока выполняется программа (и даже в exe-файле!). Локальные переменные также могут быть объявлены статическими с помощью ключевого слова static. Статические переменные по умолчанию инициализируются нулевыми значениями (иначе непонятно, что писать в exe-файл).

3.3. Данные как объекты памяти ЭВМ

Примеры статической и локальной продолжительности существования:

```
int a =10; // статическая
fun()
{
    float a = 20; // локальная
    static double b; // статическая, но видна только в fun()
}
```

3.3. Данные как объекты памяти ЭВМ

Динамическая продолжительность существования характеризуется тем, что объекты создаются и уничтожаются с помощью специальных операторов (new, delete) или функций (malloc(), free() из alloc.h) в специальной области памяти – *куче*.

Куча построена в виде связного списка блоков памяти фиксированной длины.

Когда программа «создает» динамическую переменную, выполняется специальный код, отсоединяющий один или более блоков от связного списка и возвращающий адрес новой переменной, который записывается в указатель, определенный самой программой.

Когда программа «уничтожает» динамическую переменную выполняется специальный код присоединяющий освободившиеся блоки памяти к связному списку кучи.

3.3. Данные как объекты памяти ЭВМ

Формат применения операторов new, delete:

указатель = new тип (инициализатор);

int *h = new int (10);

delete указатель;

delete h; // освобождается память, указатель не удаляется

При определении динамических массивов они не инициализируются, но зато необходимо указать их размер:

long *p = new long [3];

…

delete [ ] p; //освобождение памяти для всего массива

3.3. Данные как объекты памяти ЭВМ

Тип компоновки определяет соответствие идентификаторов, объектам в программе, размещенной в нескольких файлах (модулях).

Имя может соответствовать одному объекту, определенному в каком-либо модуле, либо нескольким объектам, размещенным в различных модулях.

Если объект определен в одном модуле, а используется в другом, то говорят о *внешнем связывании*, иначе говорят о *внутреннем связывании*. Внешнее связывание выполняет компоновщик, внутренне – компилятор.

Тип компоновки определяется компилятором по контексту и ключевым словам static и extern.

3.3. Данные как объекты памяти ЭВМ

Все локальные объекты по умолчанию являются объектами внутренней компоновки.

Все глобальные объекты по умолчанию являются объектами внешней компоновки. Изменить тип компоновки можно с использованием ключевого слова static.

Для того, чтобы объекты, определенные в другом файле были доступны в текущем, они должны быть в нем описаны (иначе компилятор выдает сообщение об ошибке). Для описания «внешних» функций применяются прототипы, «внешних» переменных - ключевое слово extern.

```
// ---------------------- Модуль 1 ----------------------
//P3-10-1.CPP - первый файл многомодульной программы
int K = 0;                // Для K - внешнее связывание
void counter(void)        // Для counter - внешнее связывание
{ static int K_IN = 0; // Для K_IN - внутреннее связывание
  K += ++K_IN;
}

// ---------------------- Модуль 2 ----------------------
//P3-10-2.CPP - второй (основной) файл программы
#include <iostream.h>
void main(void)
{ void counter(void);  // Прототип - внешнее связывание
  void display(void);  // Прототип - внешнее связывание
  // K - локальный объект - внутреннее связывание:
  for (int K = 0; K < 3; K++)
     { cout << "\nПараметр цикла K = " << K;
       counter();       // Изменяет свою K_IN и внешнюю K
       display();
     }
}

// ---------------------- Модуль 3 ----------------------
//P3-10-3.CPP - третий файл программы
#include <iostream.h>
void display(void)        // Для display - внешнее связывание
{ extern int K;           // Для K - внешнее связывание
 ,static int K_IN = 0; // Для K_IN - внутреннее связывание
  cout << "\nВнешнее K = " << K++ <<
          " Внутреннее K_IN из функции display = " <<
          K_IN++;
}
```

Результат выполнения:

```
Параметр цикла K = 0
Внешнее K = 1 Внутреннее K_IN из функции display = 0
Параметр цикла K = 1
Внешнее K = 4 Внутреннее K_IN из функции display = 1
Параметр цикла K = 2
Внешнее K = 8 Внутреннее K_IN из функции display = 2
```

3.4. Операторы C++

Простые и составные операторы

Все «простые» операторы заканчиваются точкой с запятой.

```
a  = 2; // простой оператор
```

Составной оператор:

```
{
    a*=a;
    b = a;
}
```

Блок (содержит определение локальной переменной):

```
{
    int c = 1;
    a+=2;
    b = a+c;
}
```

3.4. Операторы C++

Операторы выбора

Синтаксис:

if (<выражение>) <оператор1> [else <оператор2>]

выражение - скалярное или указатель, если не равно нулю, выполняется оператор1, иначе оператор2

оператор1 и оператор2 могут быть:

- простыми;
- составными;
- блоками.

[else <оператор2>] может быть опущено.

Допустима вложенность конструкций выбора, при этом

***else** всегда относится к ближайшему **if**!*

3.4. Операторы C++

Пример:

```
float x;
…
if (x>0)
    if (x>1) cout <<“x>1”;
else cout << “x<=0”;
```

- логическая ошибка: при x = 0.5 выдает «x<=0». Правильное решение:

```
if (x>0)
    {if (x>1) cout <<“x>1”;}
else cout << “x<=0”;
```

3.4. Операторы C++

Переключатель:

Синтаксис:

```
switch (<выражение>)
{
    case <константа1>: <операторы1>
    case <константа2>: <операторы2>
    ....
    [default: <операторN>]
}
```

Выражение – обязательно целочисленное. По вычислению выражения происходит переход к метке с соответствующей константой ***и далее программа выполняется последовательно!*** Досрочного выхода из переключателя применяется оператор **break.** Если выражение не равно ни одной константе, осуществляется переход к метке **default** или выход из переключателя, если такой метки нет.

3.4. Операторы C++

Пример:

```
int x, y, z;
…
switch(x)
{
    case 0: y = 0;
    case 1: case2: x =2; z =1;
    case3: y = 3; break;
    case 4: y = 4; z = 0; break;
    default: y = 5;
}
```

3.4. Операторы C++

Прогон (x = 0):

```
switch(x)
{
    case 0: y = 0;
    case 1: case2: x =2; z =1;
    case3: y = 3; break;
    case 4: y = 4; z = 0; break;
    default: y = 5;
}
```

Итог: x = 2, y = 3, z = 1.

Очевидно, что то же самое получится при x = 1 и x = 2!

3.4. Операторы C++

Прогон (x = 3):

```
switch(x)
{
    case 0: y = 0;
    case 1: case2: x =2; z =1;
    case3: y = 3; break;
    case 4: y = 4; z = 0; break;
    default: y = 5;
}
```

Итог: x = 3, y = 3, z = ?.

3.4. Операторы C++

Прогон (x = 4):

```
switch(x)
{
    case 0: y = 0;
    case 1: case2: x =2; z =1;
    case3: y = 3; break;
    case 4: y = 4; z = 0; break;
    default: y = 5;
}
```

Итог: x = 4, y = 4, z = 0.

3.4. Операторы C++

Прогон (x = 5):

```
switch(x)
{
    case 0: y = 0;
    case 1: case2: x =2; z =1;
    case3: y = 3; break;
    case 4: y = 4; z = 0; break;
    default: y = 5;
}
```

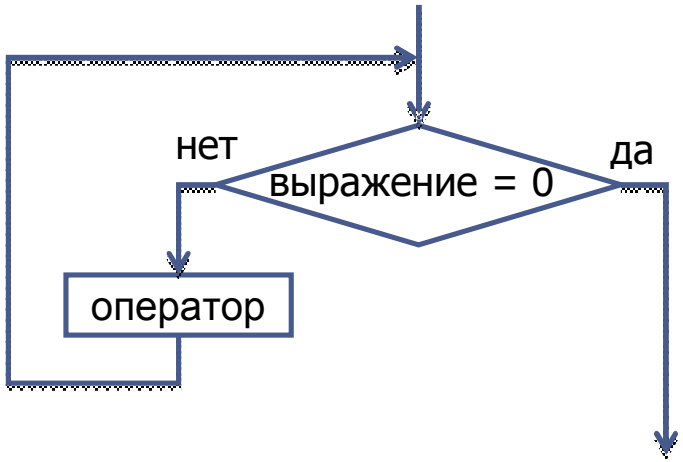
Итог: x = 5, y = 5, z = ?.

3.4. Операторы C++

Операторы цикла

Цикл с предусловием:

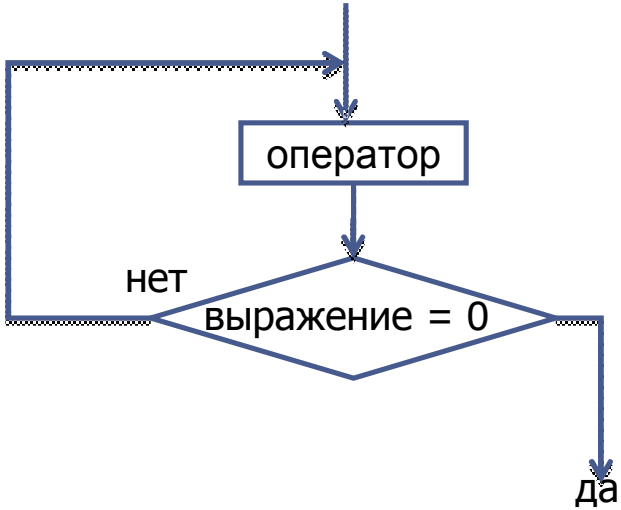
while (<выражение>) <оператор>




```
//вычислить длину строки:
char *str = “String”;
int  len = 0;
while (*strt++) len++;
```

Цикл с постусловием:

do <оператор> while (<выражение>);



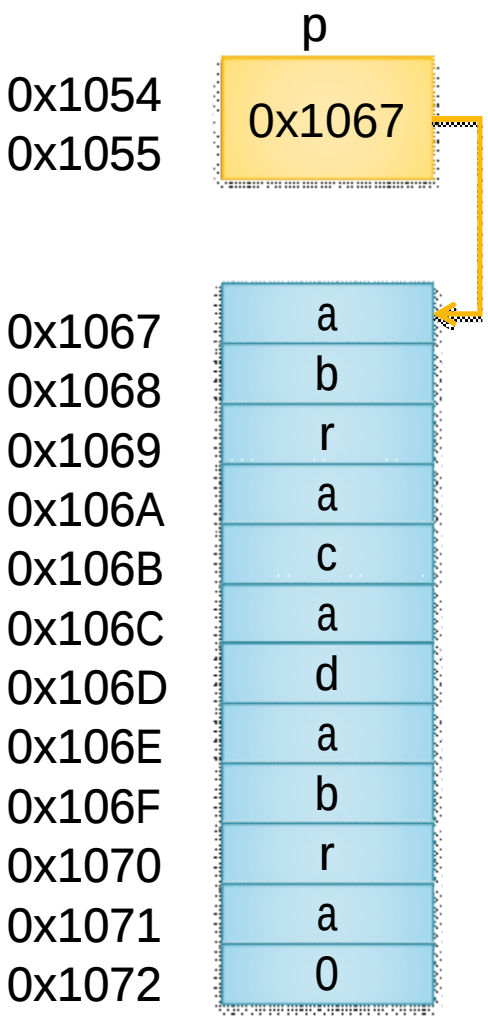

```
//копировать строки:
char *str = “String”;
char nstr[10], *pstr = nstr;
do *pstr++ =*str; while (*str++);
//или
do ; while (*pstr++ =*str++);
```

3.4. Операторы C++

Пример (цикл с предусловием)

Заменить в строке букву ‘a’ на букву ‘b’:

```
char * p;
…
 p = “Abracadabra”;
…
while (*p)
{
    if (*p==‘a’) *p = ‘b’;
    p++;
}
```

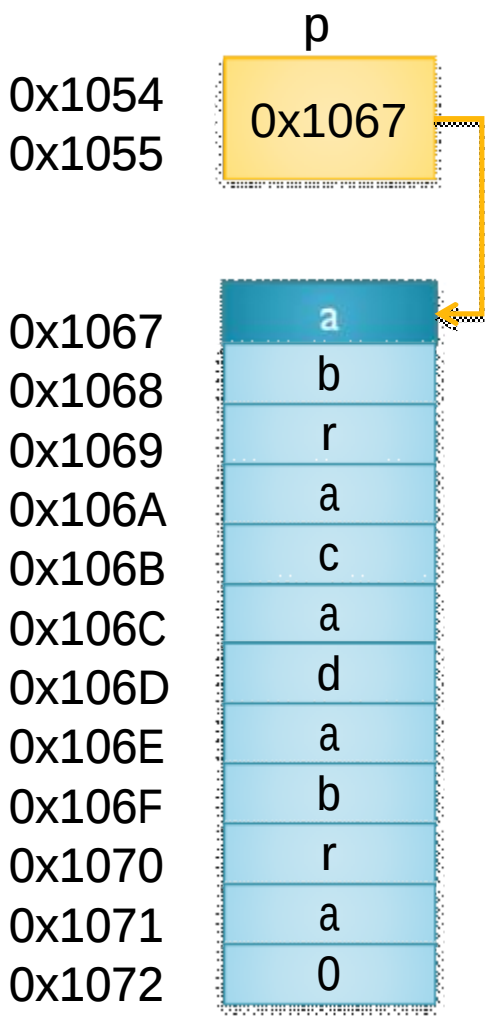


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

То, на что указывает p (символ ‘a’), не равно нулю

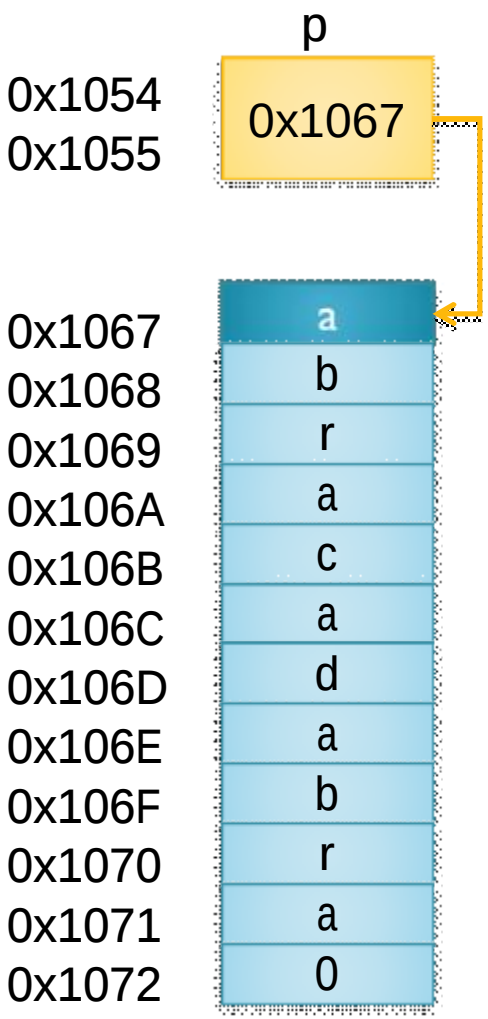


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p, равно символу ‘a’

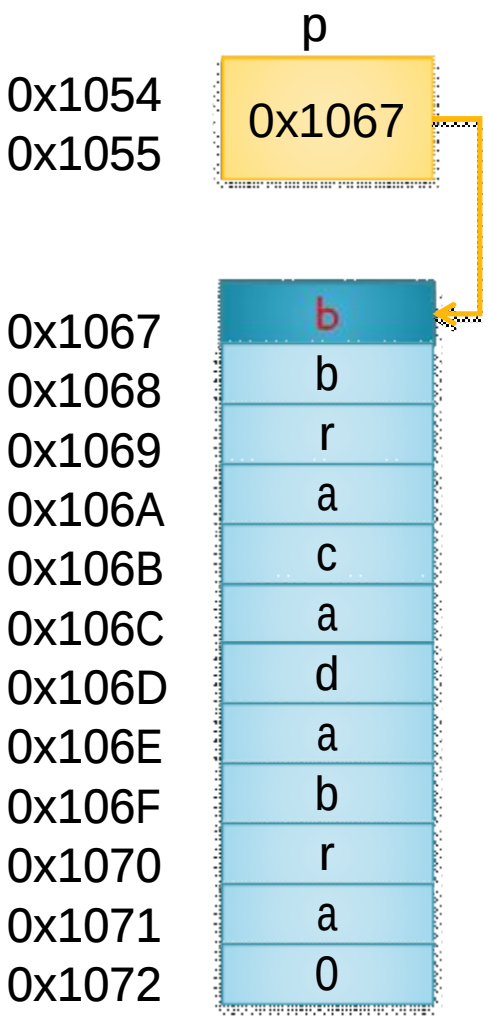


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Записываем по адресу, содержащемуся в указателе, код символа ‘b’

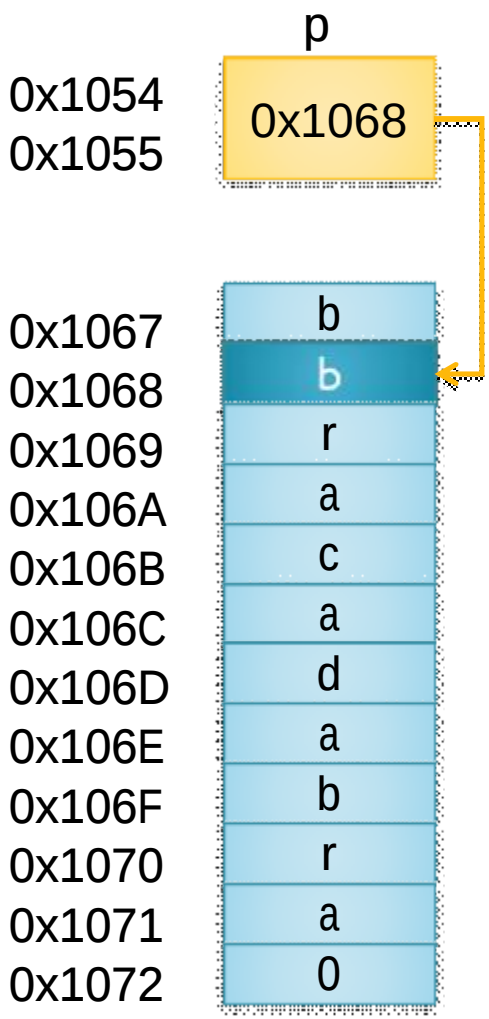


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

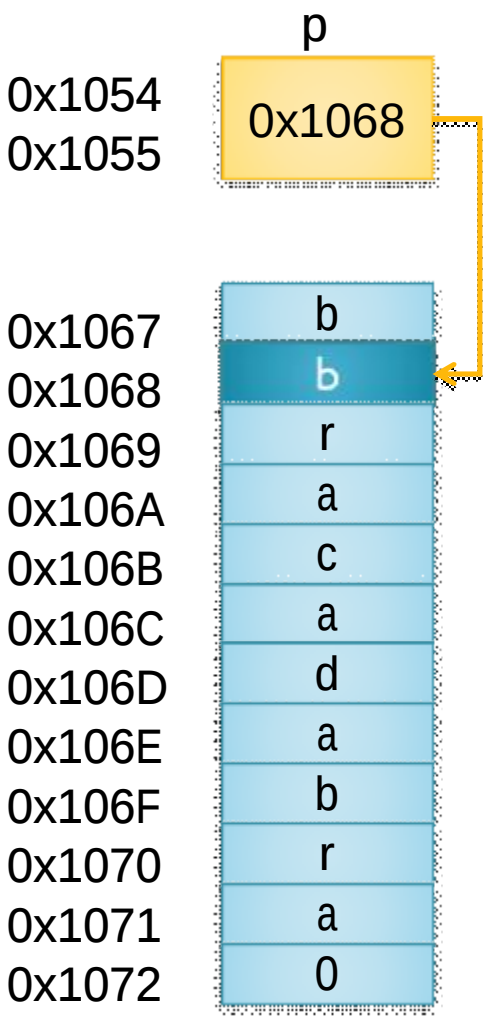


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p (символ ‘b’), не равно нулю

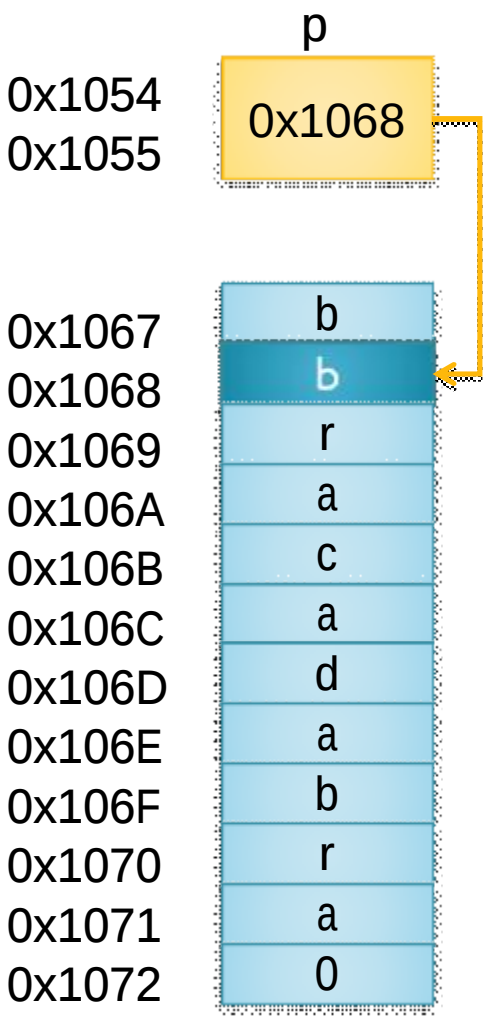


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

То, на что указывает p, не равно символу ‘a’

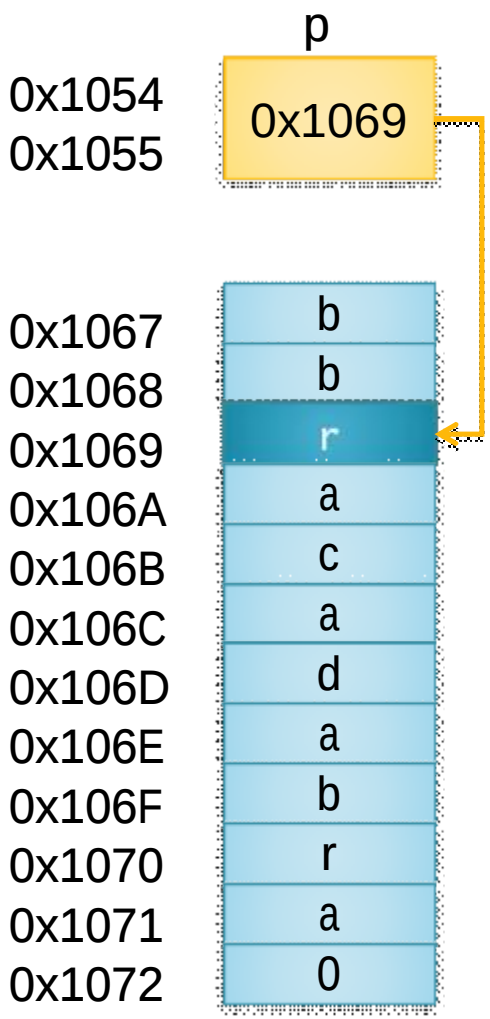


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

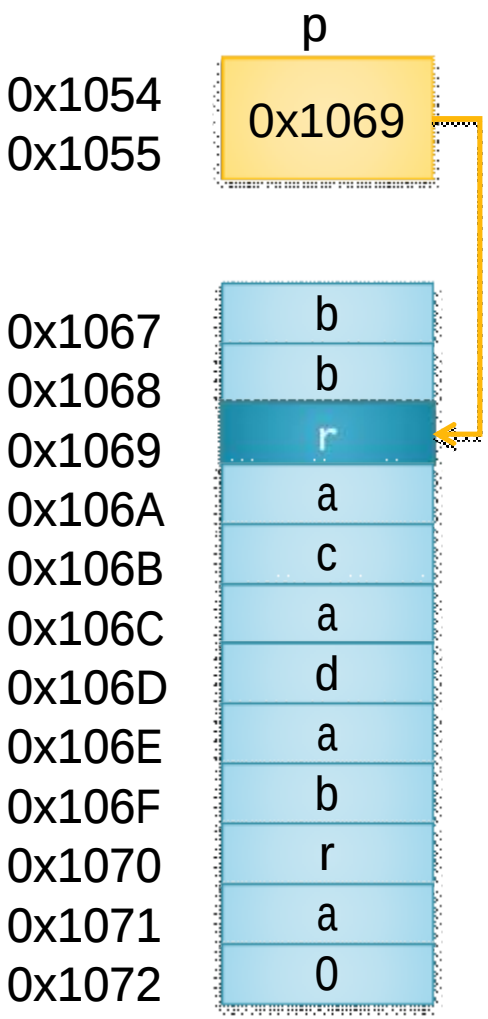


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p (символ ‘r’), не равно нулю

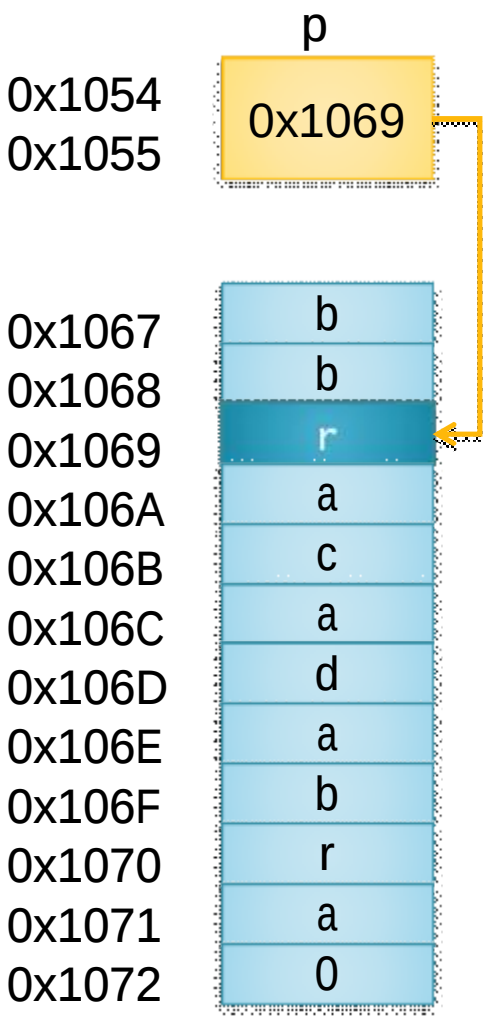


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

То, на что указывает p, не равно символу ‘a’

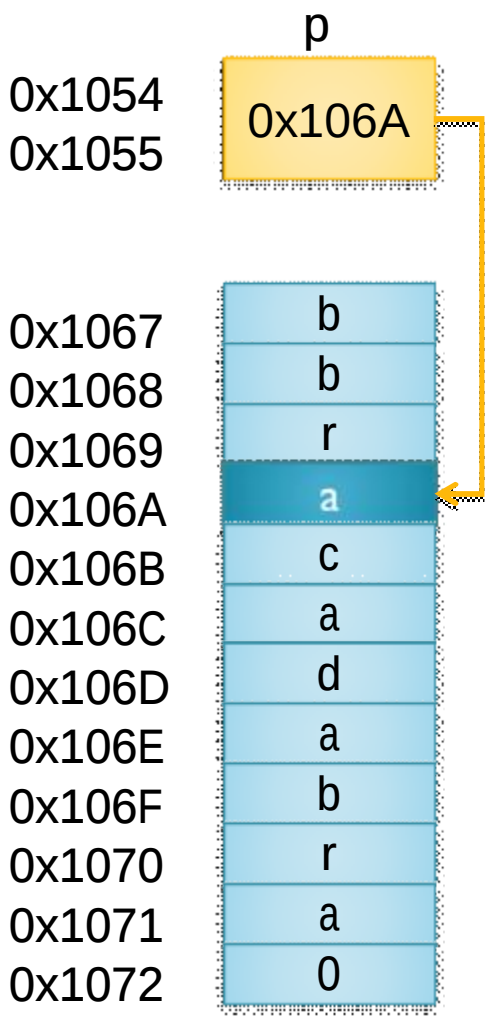


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

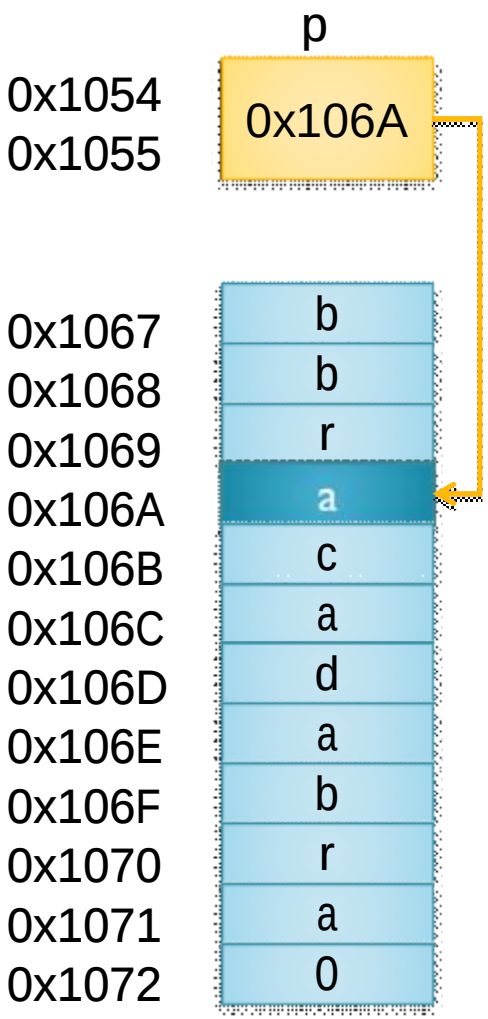


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p (символ ‘a’), не равно нулю

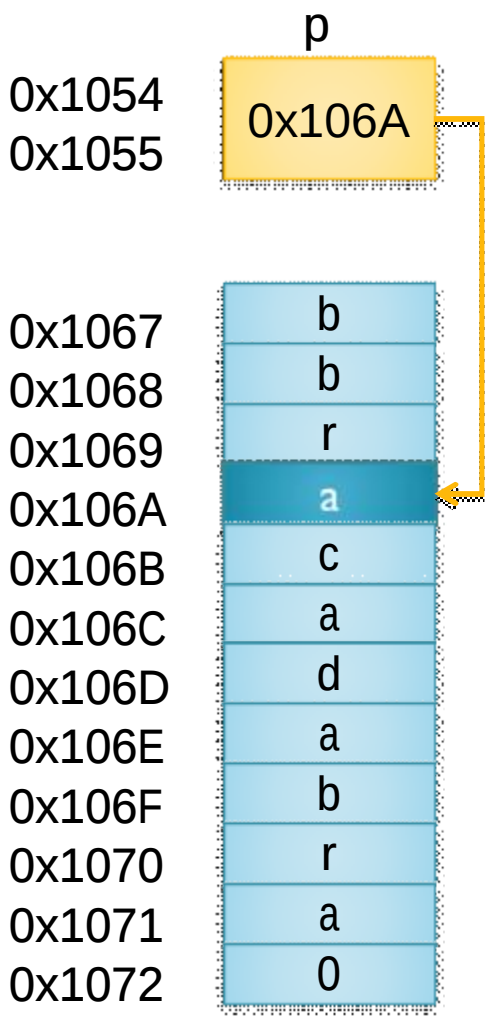


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p, равно символу ‘a’

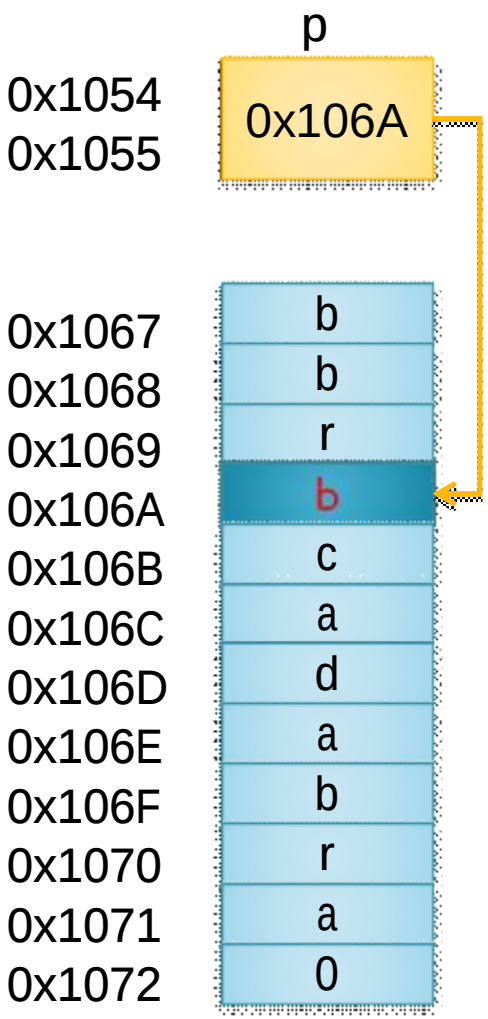


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Записываем по адресу, содержащемуся в указателе, код символа ‘b’

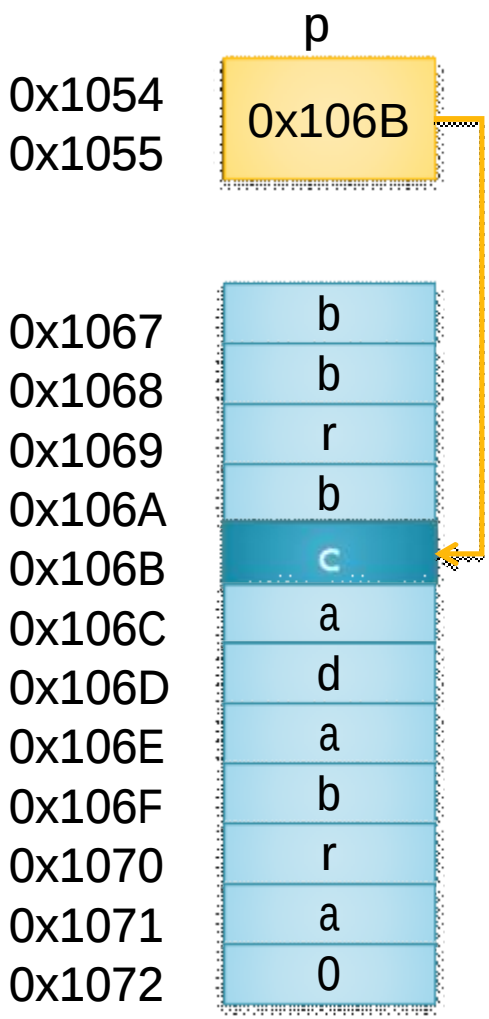


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

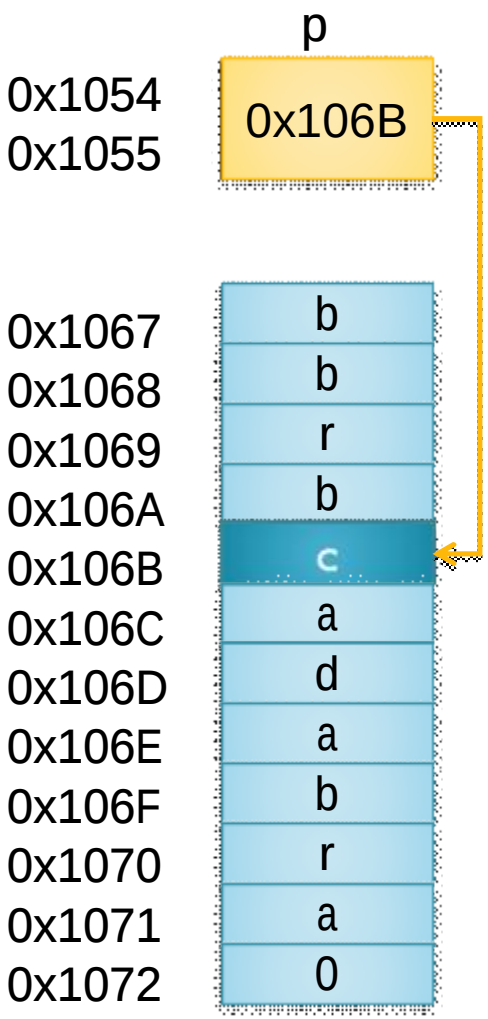


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p (символ ‘c’), не равно нулю

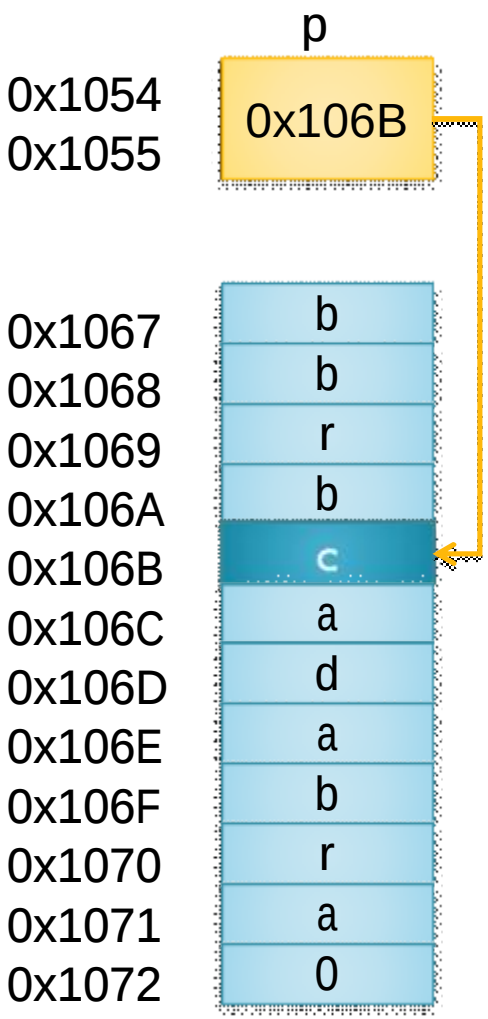


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

То, на что указывает p, не равно символу ‘a’

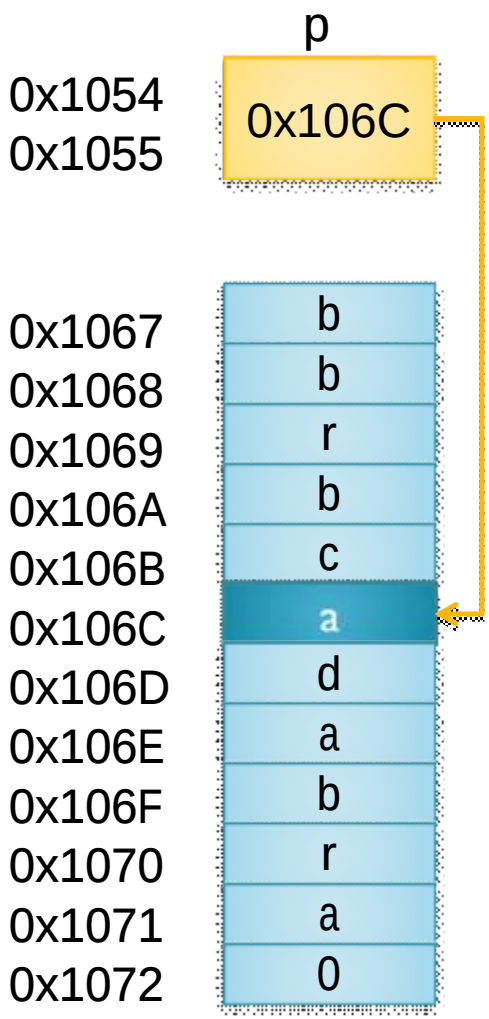


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

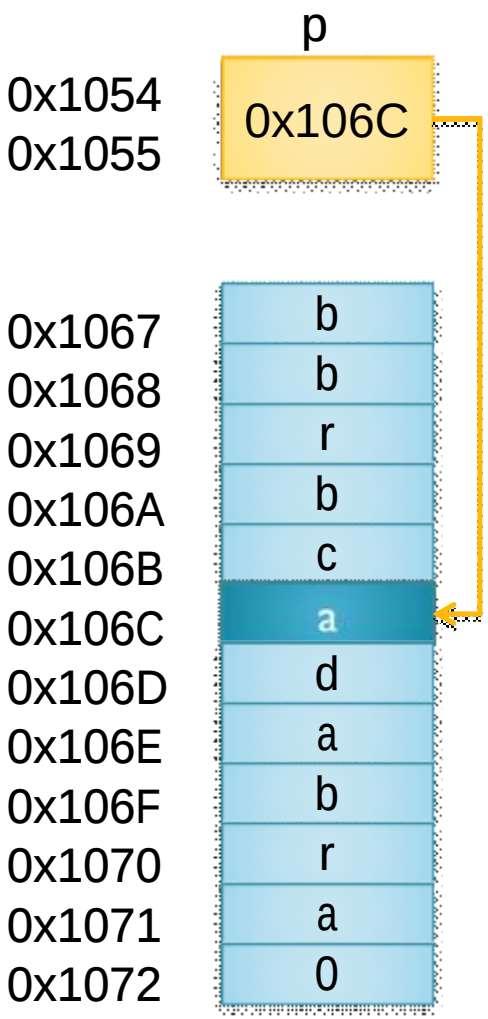


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

То, на что указывает p (символ ‘a’), не равно нулю

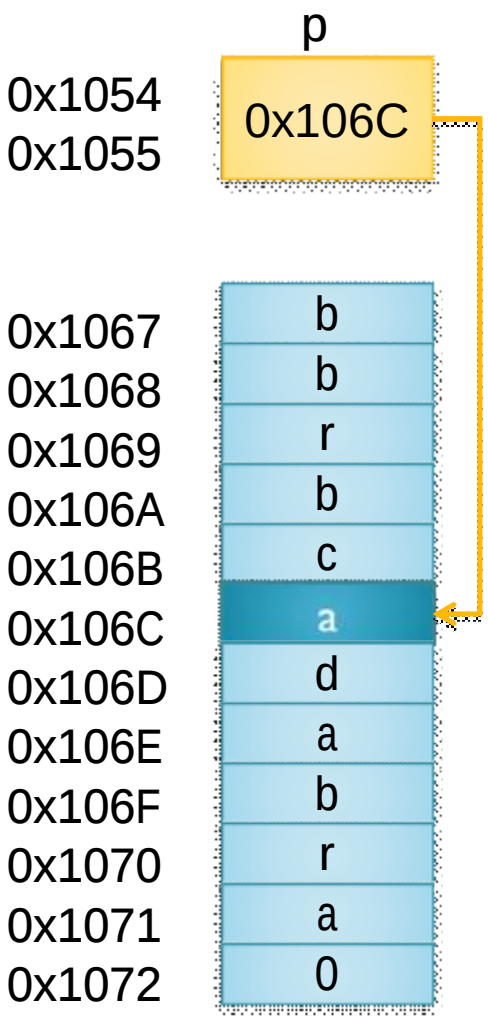


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

То, на что указывает p, равно символу ‘a’

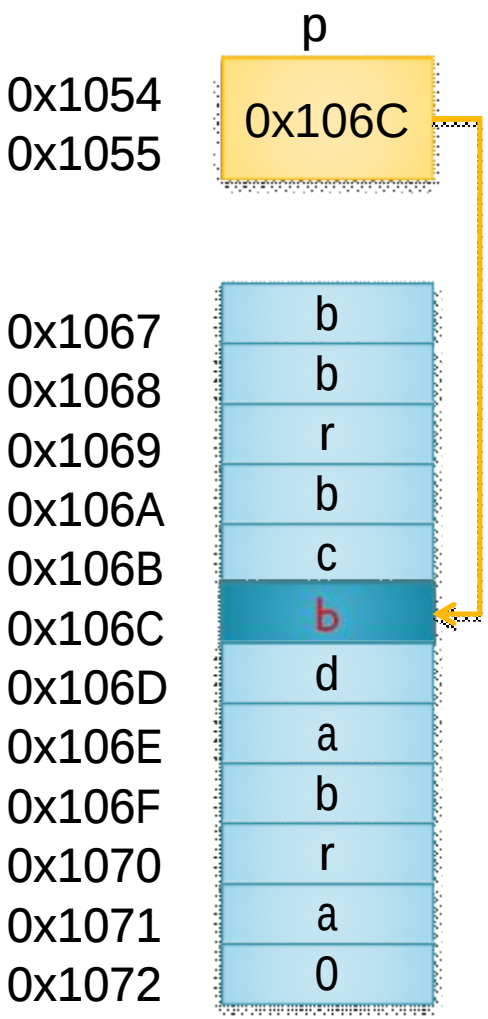


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Записываем по адресу, содержащемуся в указателе, код символа ‘b’

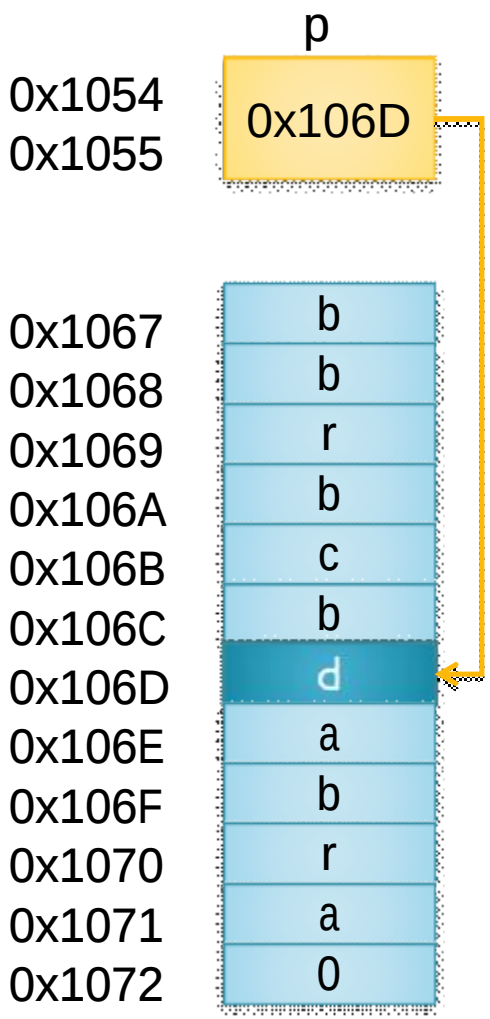


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

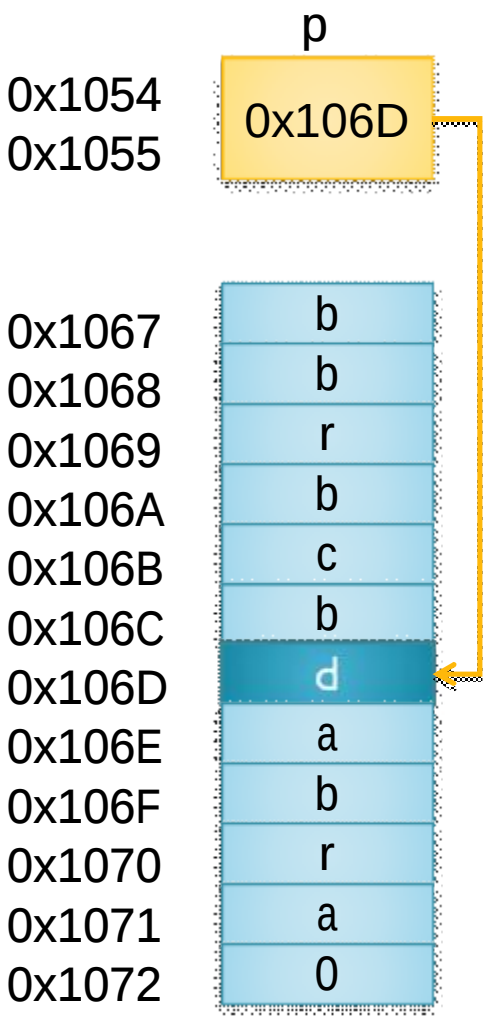


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

То, на что указывает p (символ ‘d’), не равно нулю

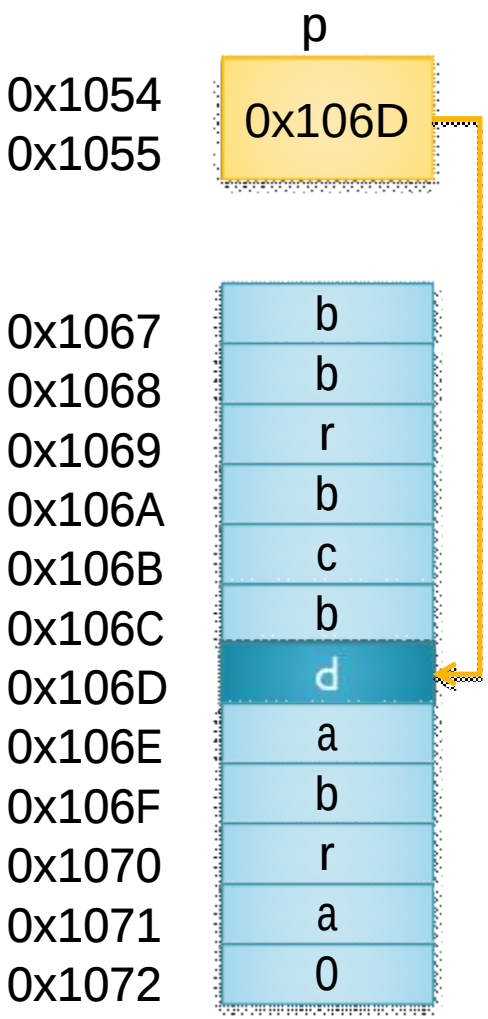


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

То, на что указывает p, не равно символу ‘a’

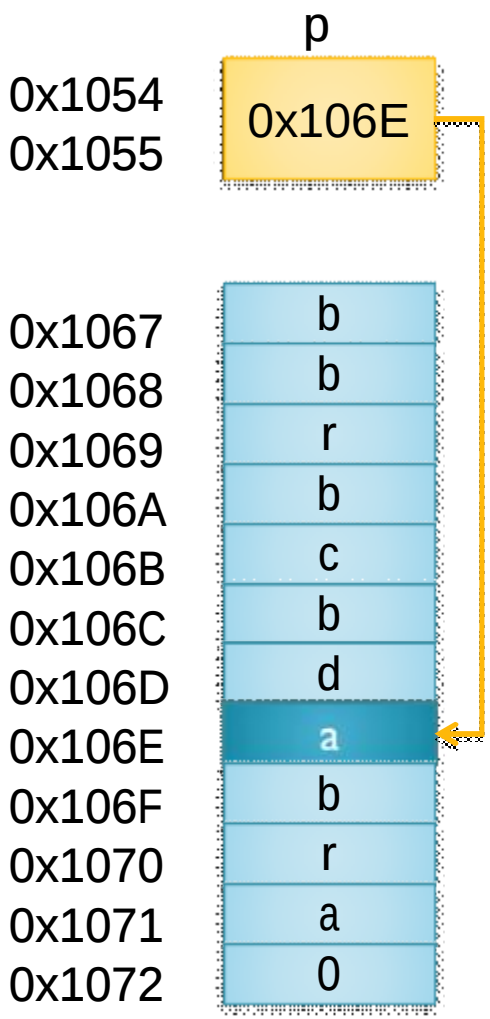


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

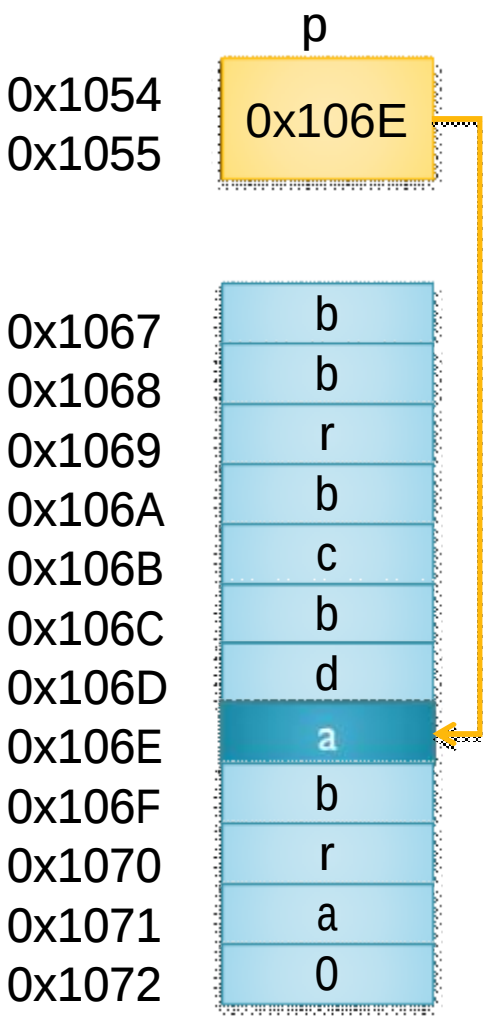


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p (символ ‘a’), не равно нулю

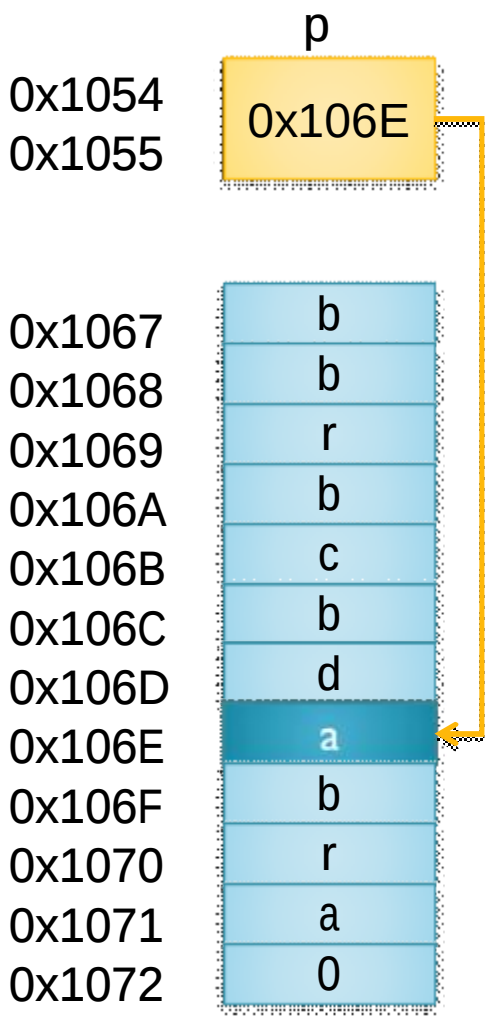


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p, равно символу ‘a’

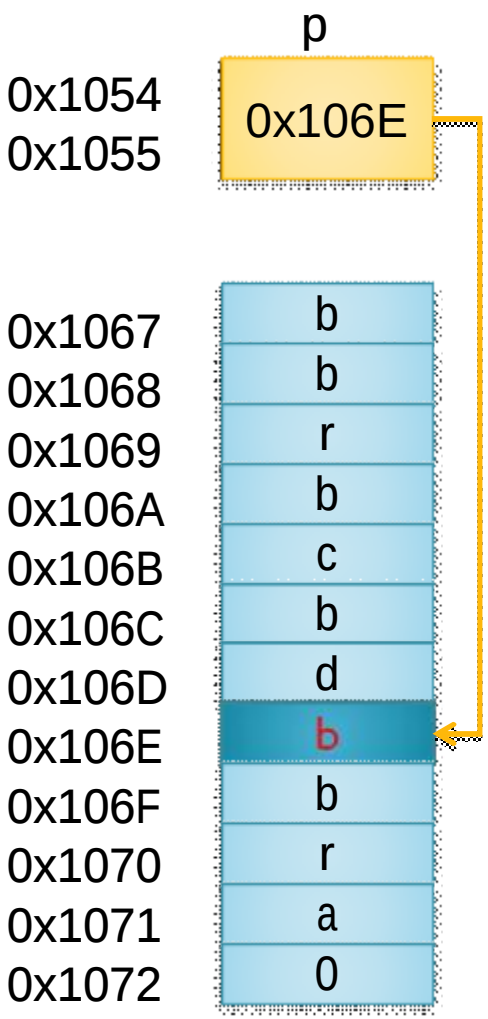


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

Записываем по адресу, содержащемуся в указателе, код символа ‘b’

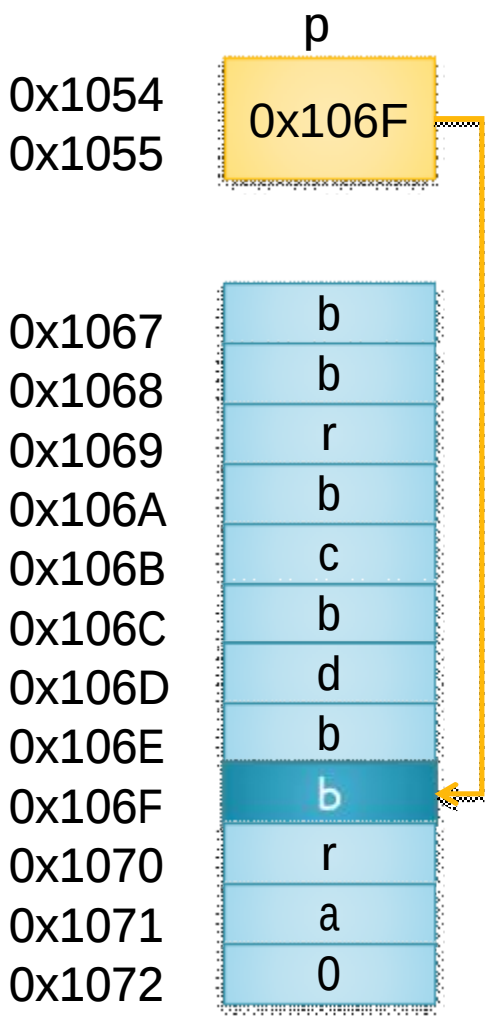


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

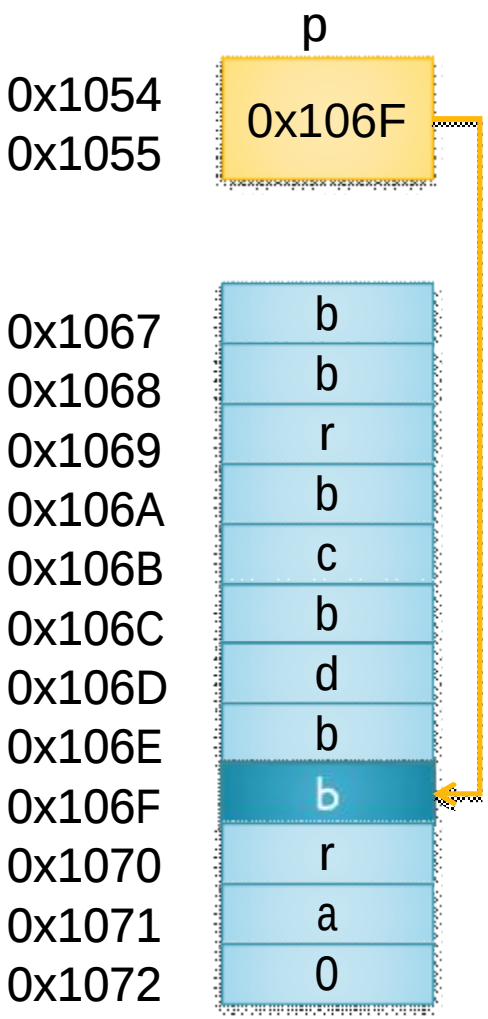


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

То, на что указывает p (символ ‘b’), не равно нулю

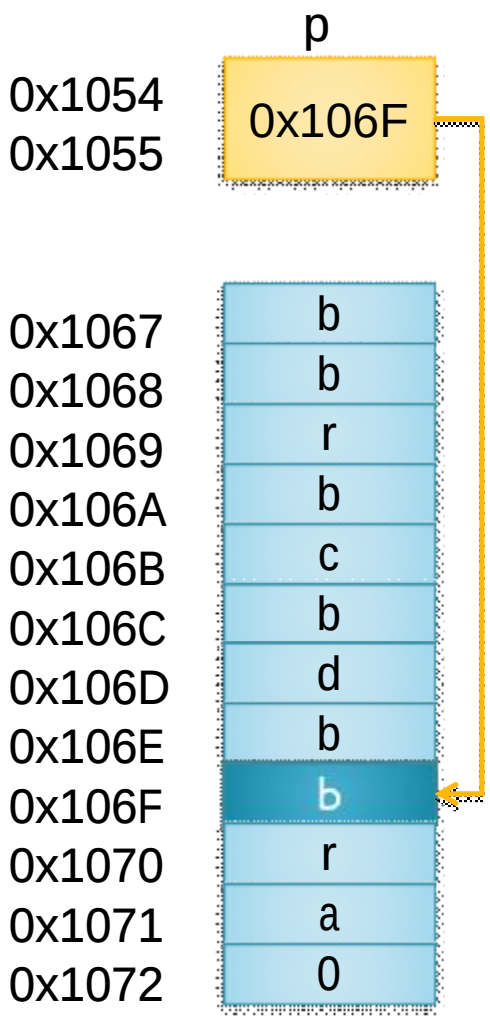


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

То, на что указывает p, не равно символу ‘a’

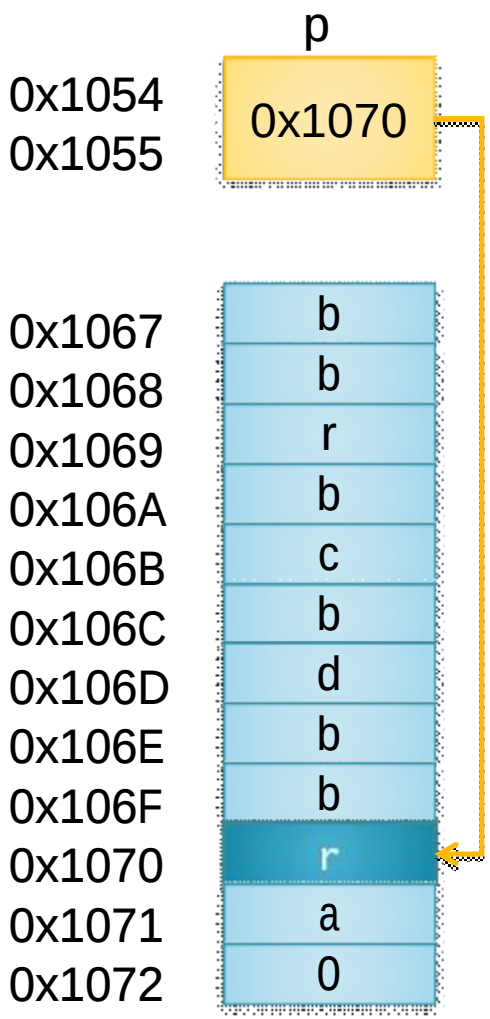


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

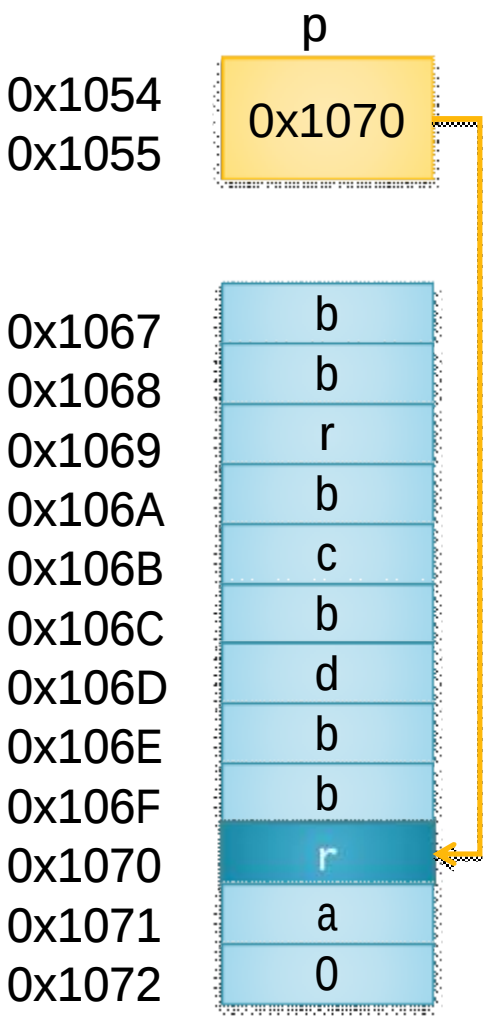


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

То, на что указывает p (символ ‘r’), не равно нулю

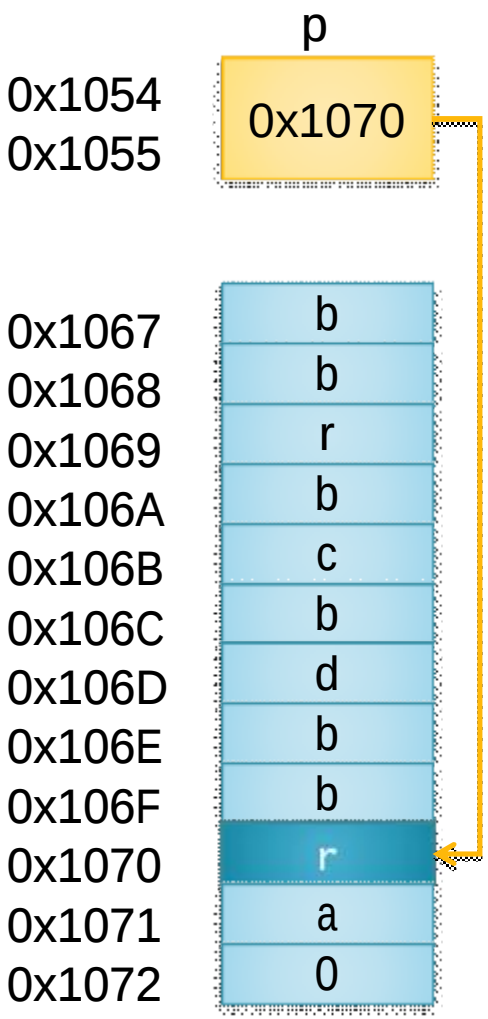


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

То, на что указывает p, не равно символу ‘a’

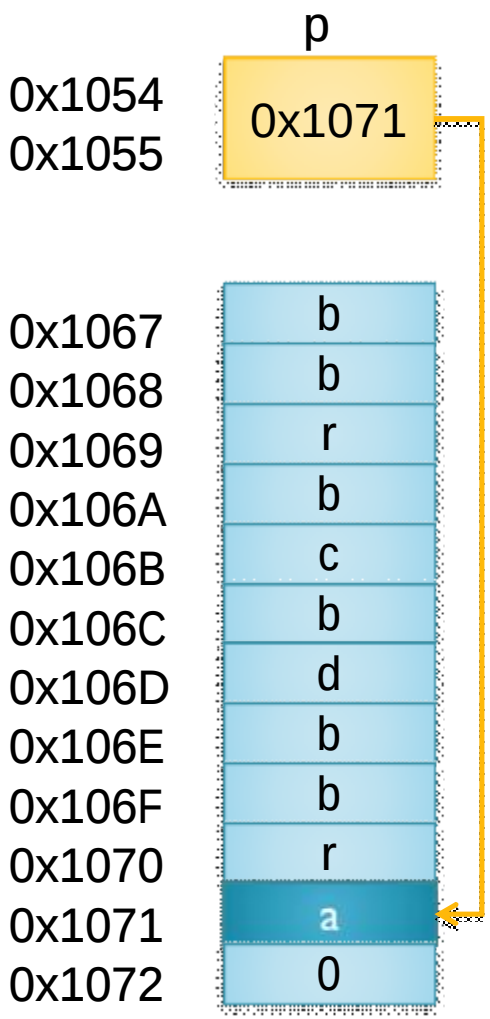


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

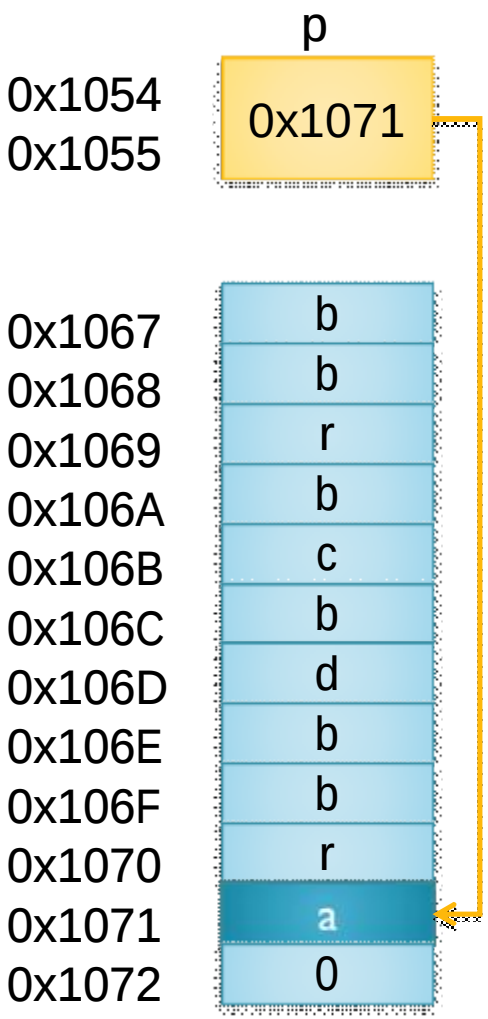


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

То, на что указывает p (символ ‘a’), не равно нулю

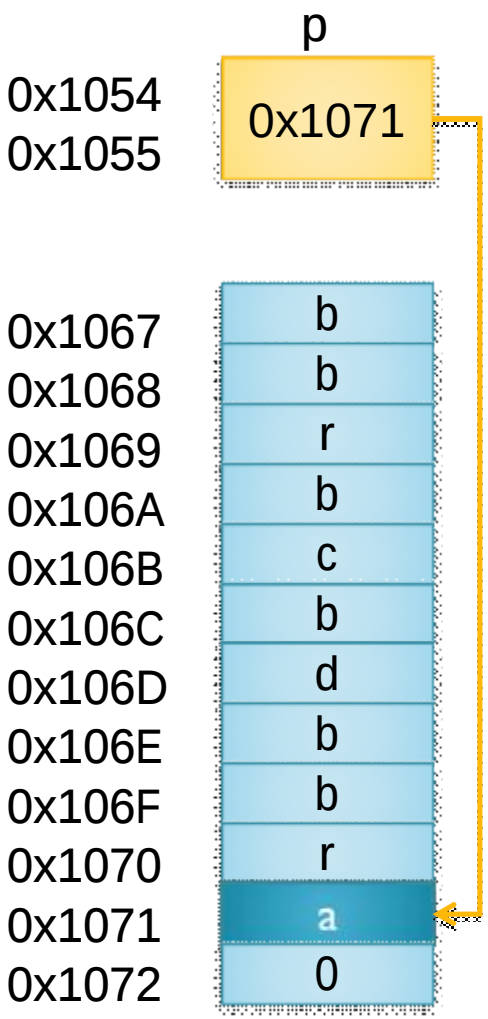


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p=='a')
        *p = 'b';
    p++;
}
```

То, на что указывает p, равно символу ‘a’

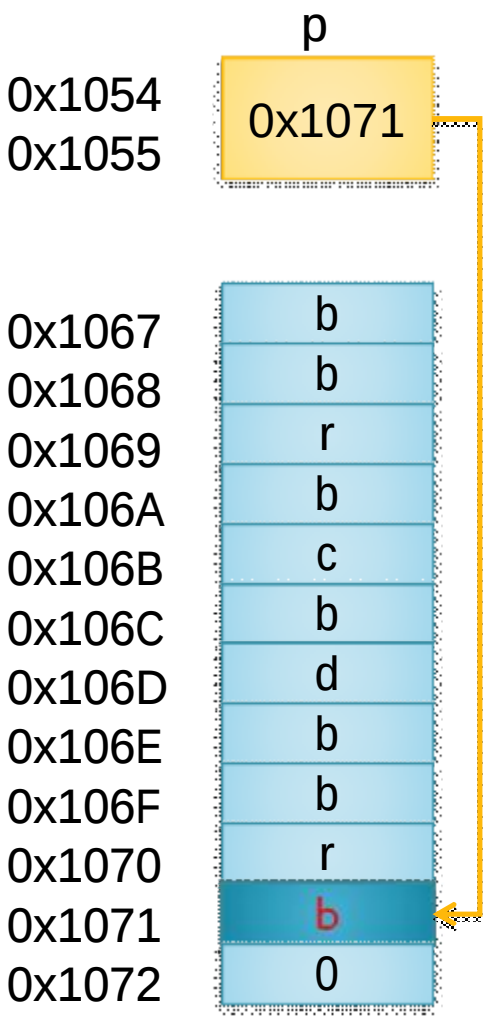


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Записываем по адресу, содержащемуся в указателе, код символа ‘b’

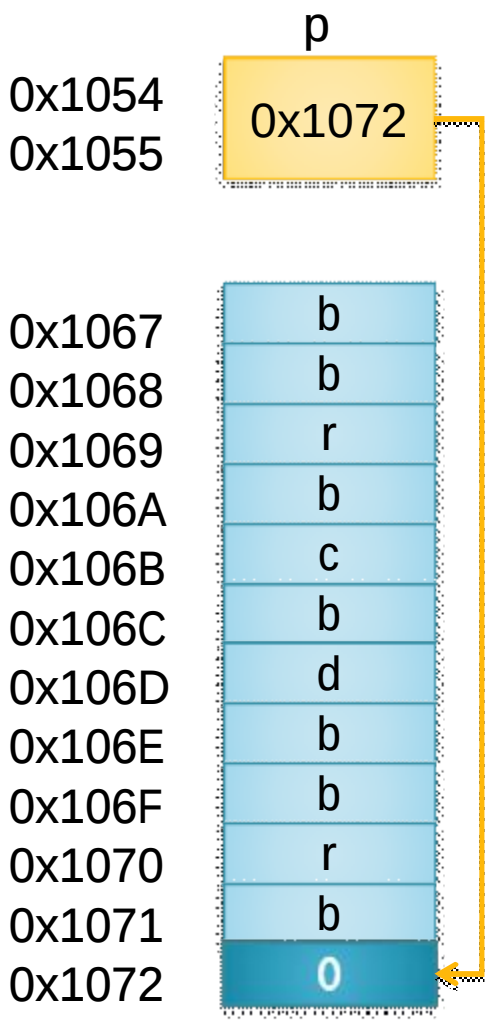


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
        *p = ‘b’;
    p++;
}
```

Увеличиваем указатель

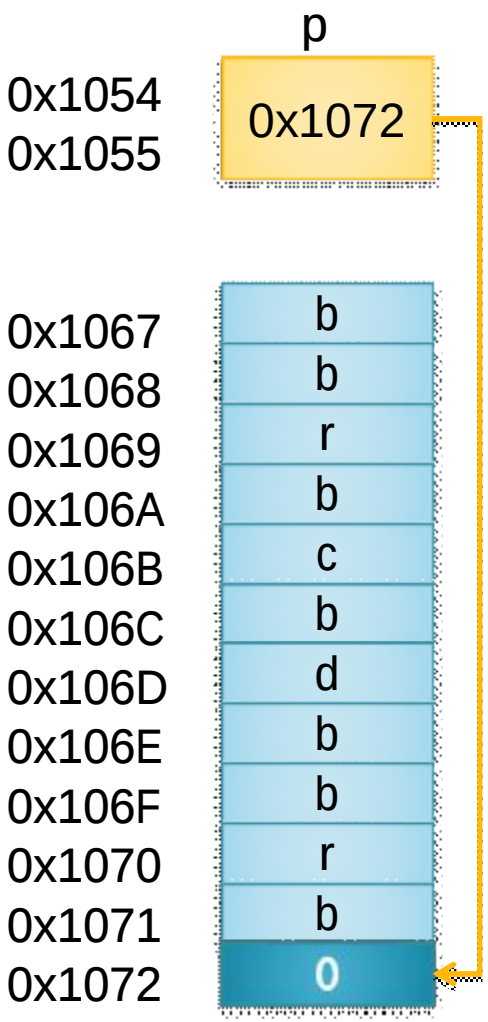


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

То, на что указывает p, равно нулю

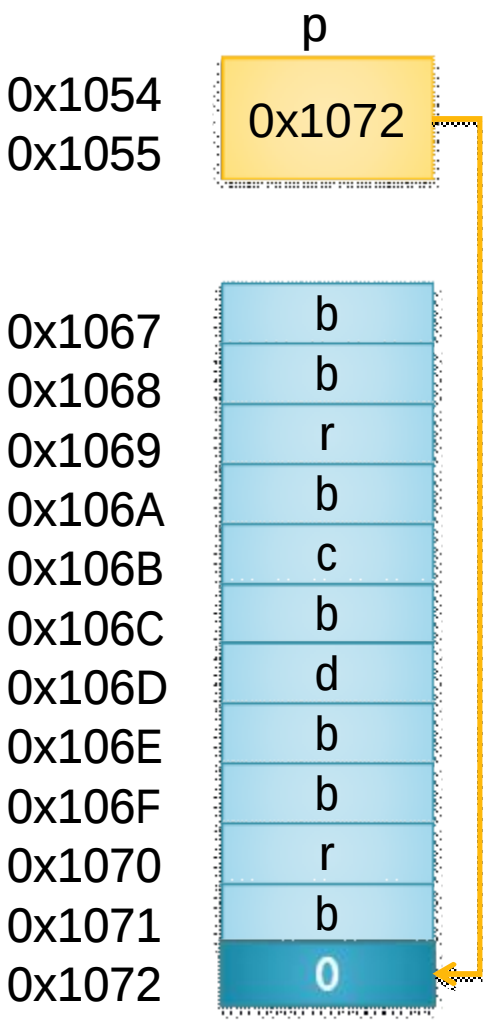


3.4. Операторы C++

Шаг №1

```
while (*p)
{
    if (*p==‘a’)
            *p = ‘b’;
    p++;
}
```

Выходим из цикла



3.4. Операторы C++

Итерационный цикл:

Синтаксис:

for ([<инициализатор>]; [<условие>];[<постоператоры>])

[<оператор>]

инициализатор – список определений, разделенный запятыми, *выполняется один раз при выходе в цикл*;

условие – проверяется перед началом каждой итерации, итерация выполняется, если условие истинно;

постоператоры – список выражений, разделенных запятыми, *вычисляются в каждой итерации после вычисления оператора*;

оператор – пустой, простой, составной или блок.

3.4. Операторы C++

Пример:

Вывод элементов массива на экран:

int M[ ] = {1,2,3,4};

1) можно так:

for (int i =0; i<4; i++) cout << ‘\n’ << M[i];

2) можно и так:

for (int i =0; i<4;) cout << ‘\n’ << M[i++]);

3) или даже так:

int i =0;

for (; i<4;); {cout << ‘\n’ << M[i]; i++;}

3.4. Операторы C++

Пример вложенных циклов.

В массиве

int M[ ] = {1,2,3,4,5,6,7,8,9};

записана матрица с тремя строками и тремя столбцами:

1 2 3

4 5 6

7 8 9

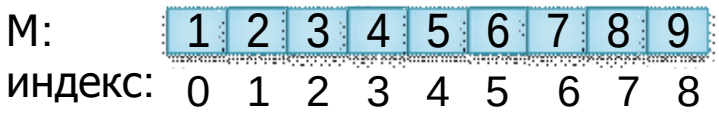
Вывести матрицу на экран.

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Объявляется переменная
i – счетчик строк

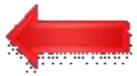


На экране:

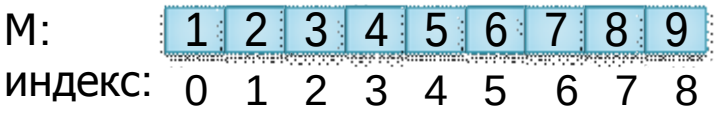
i: 0 j: ?

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



i<3 - истина

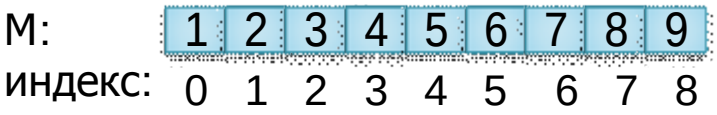


На экране:

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Объявляется переменная
j=0 – счетчик столбцов



На экране:

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –истина

M:

1	2	3	4	5	6	7	8	9

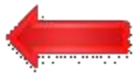
индекс: 0 1 2 3 4 5 6 7 8

i: 0 j: 0

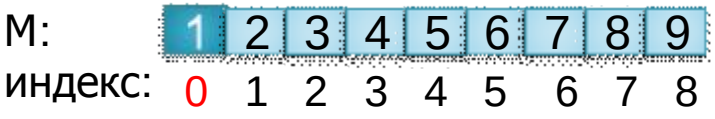
На экране:

3.4. Операторы C++

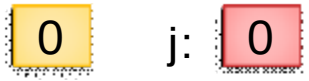
```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



Вывод на экран M[0] и табуляция



На экране:
1 |



3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

увеличиваем j

На экране:
1 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –истина

M: 1 2 3 4 5 6 7 8 9

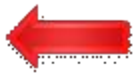
индекс: 0 1 2 3 4 5 6 7 8

На экране:
1 |

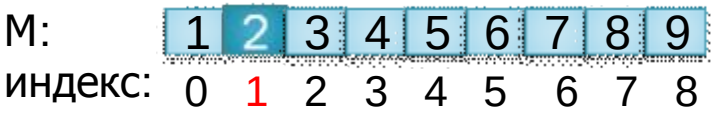
i: 0 j: 1

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



Вывод на экран M[1] и табуляция



На экране:
1 2 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

увеличиваем j

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

На экране:

1 2 |

i: 0 j: 2

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –истина

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

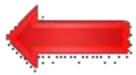
На экране:

1 2 |

i: 0 j: 2

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



Вывод на экран M[2] и табуляция

На экране:
1 2 3 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Увеличиваем j

На экране:
1 2 3 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –ложь

M:

1	2	3	4	5	6	7	8	9

индекс: 0 1 2 3 4 5 6 7 8

На экране:

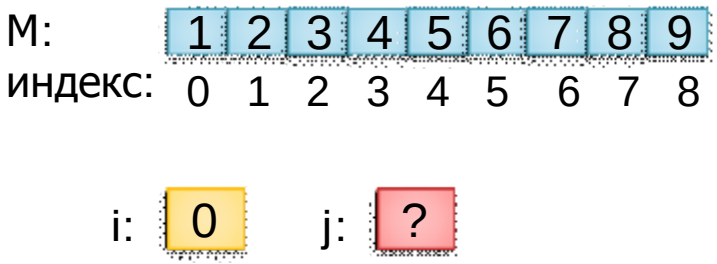
1 2 3 |

i: 0 j: 3

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

переход на следующую строку



i: 0 j: ?

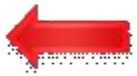
На экране:

1 2 3

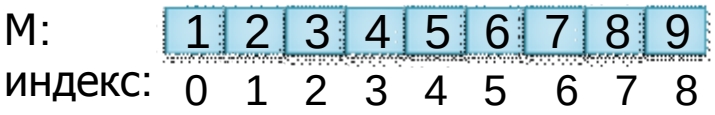
|

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



увеличиваем i



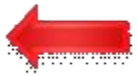
На экране:

1 2 3

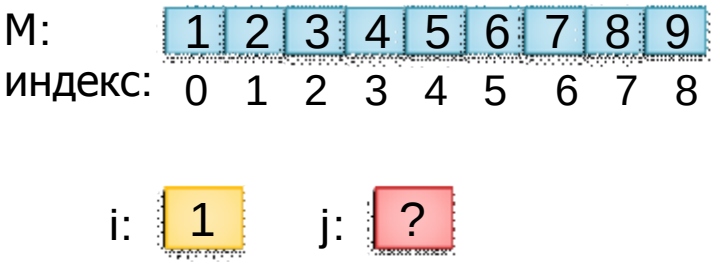
|

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



i<3 - истина



На экране:

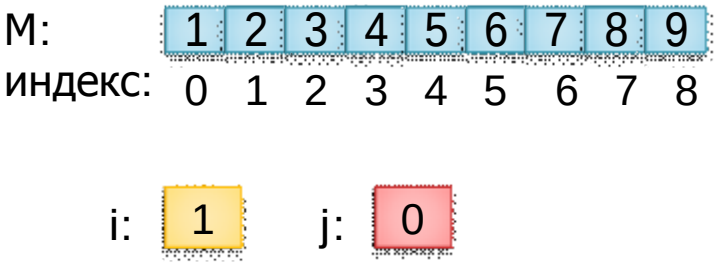
1 2 3

|

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Объявляется переменная
j=0 – счетчик столбцов

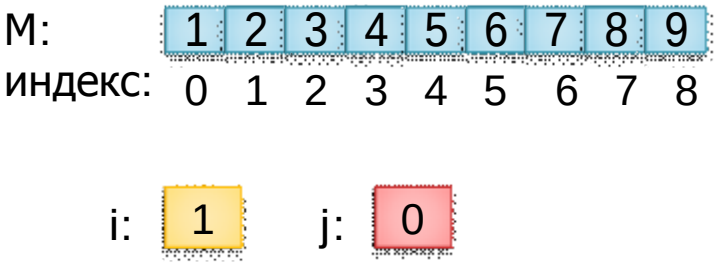


На экране:
1 2 3
|

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

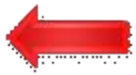
j<3 –истина



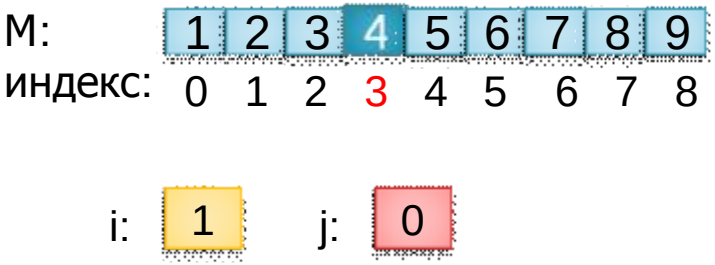
На экране:
1 2 3
|

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



Вывод на экран M[3] и табуляция



i: 1 j: 0

На экране:

1 2 3

4 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

увеличиваем j

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

i: 1 j: 1

На экране:

1 2 3

4 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –истина

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

i: 1 j: 1

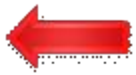
На экране:

1 2 3

4 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



Вывод на экран M[4] и табуляция

На экране:

1 2 3

4 5 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

увеличиваем j

M:

1	2	3	4	5	6	7	8	9

индекс: 0 1 2 3 4 5 6 7 8

i: 1 j: 2

На экране:

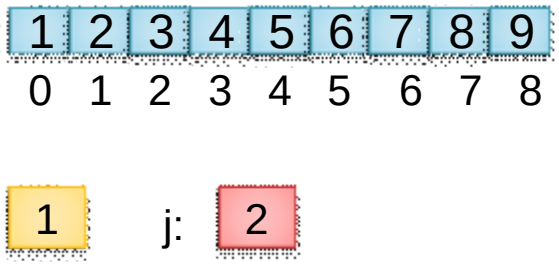
1 2 3

4 5 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –истина



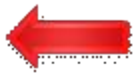
На экране:

1 2 3

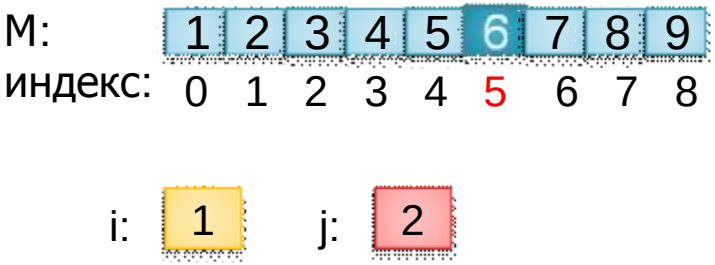
4 5 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



Вывод на экран M[5] и табуляция



i: 1 j: 2

На экране:

1 2 3

4 5 6 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Увеличиваем j

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

i: 1 j: 3

На экране:

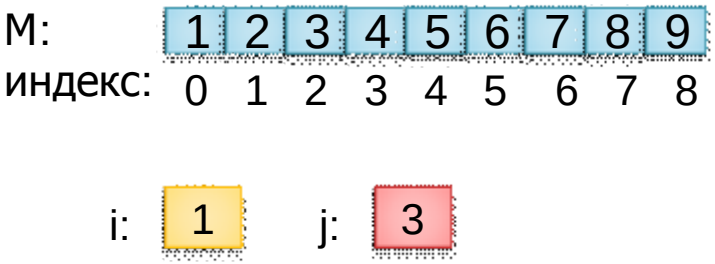
1 2 3

4 5 6 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –ложь



На экране:

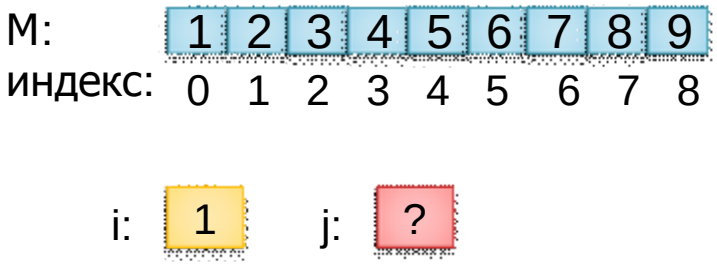
1 2 3

4 5 6 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

переход на следующую строку



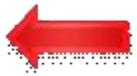
На экране:

1 2 3

4 5 6

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

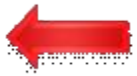


увеличиваем i

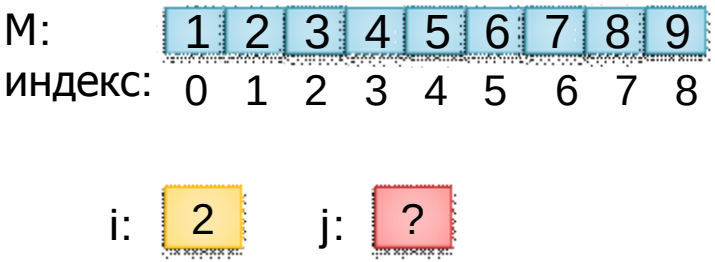
На экране:
1 2 3
4 5 6
|

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



i<3 - истина



На экране:

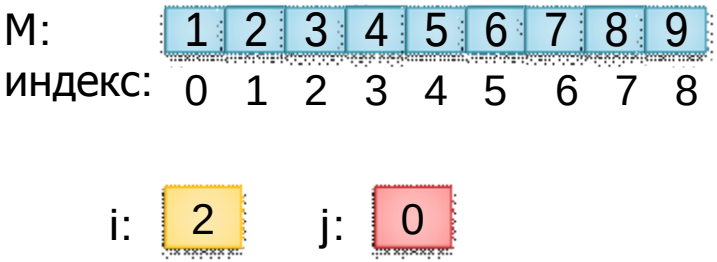
1 2 3

4 5 6

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Объявляется переменная
j=0 – счетчик столбцов

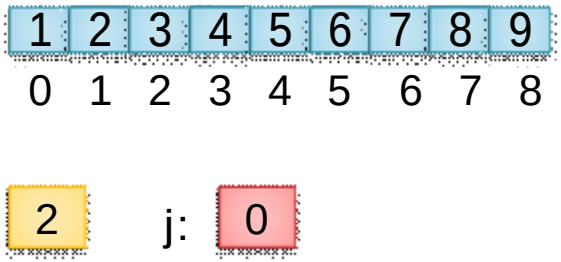


На экране:
1 2 3
4 5 6
|

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –истина



На экране:

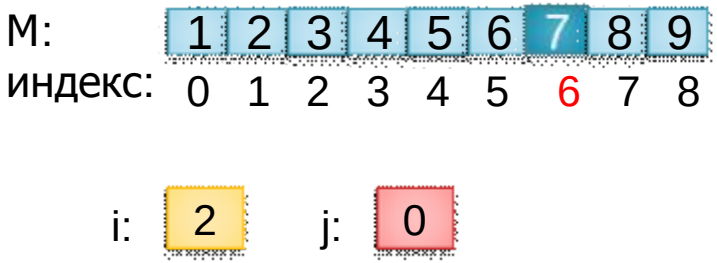
1 2 3

4 5 6

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Вывод на экран M[6] и табуляция



На экране:

1 2 3
4 5 6
7 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

увеличиваем j

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

i: 2 j: 1

На экране:

1 2 3

4 5 6

7 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –истина

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

i: 2 j: 1

На экране:

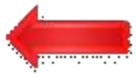
1 2 3

4 5 6

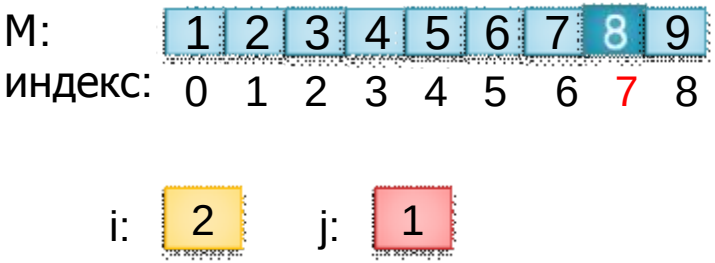
7 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



Вывод на экран M[7] и табуляция



На экране:

1 2 3
4 5 6
7 8 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

увеличиваем j

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

i: 2 j: 2

На экране:

1 2 3

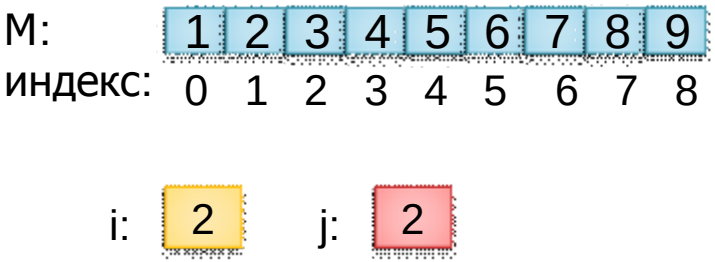
4 5 6

7 8 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –истина



На экране:

1 2 3
4 5 6
7 8 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Вывод на экран M[8] и табуляция

M: 1 2 3 4 5 6 7 8 9

индекс: 0 1 2 3 4 5 6 7 8

i: 2 j: 2

На экране:

1 2 3

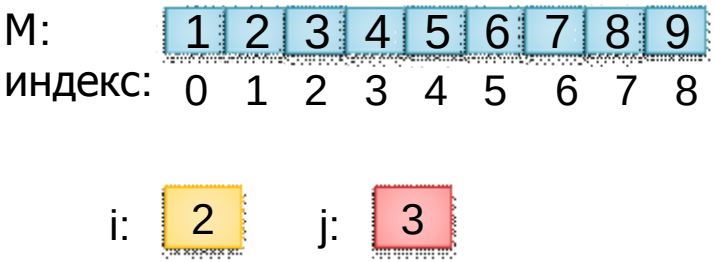
4 5 6

7 8 9

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Увеличиваем j



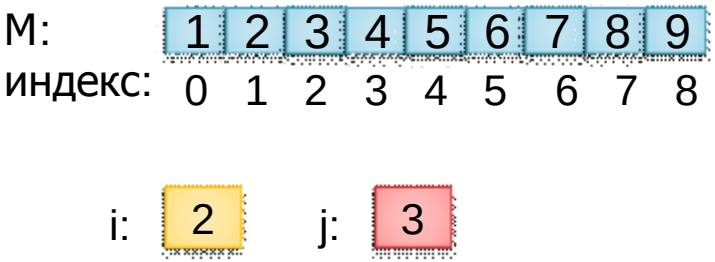
На экране:

1 2 3
4 5 6
7 8 9 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

j<3 –ложь



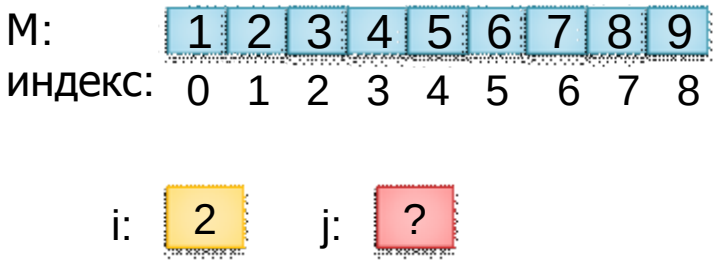
На экране:

1 2 3
4 5 6
7 8 9 |

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

переход на следующую строку

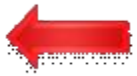


На экране:

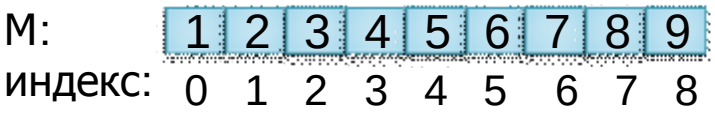
```
1   2   3
4   5   6
7   8   9
|
```

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



увеличиваем i

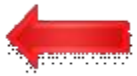


На экране:

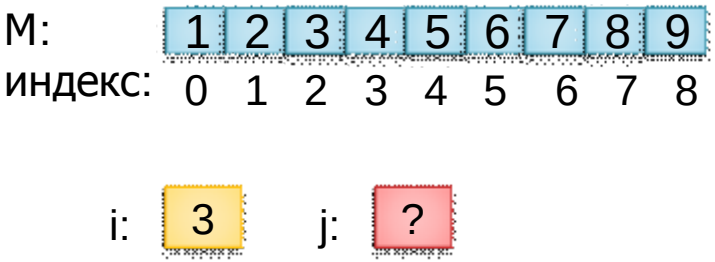
1 2 3
4 5 6
7 8 9

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```



i<3 - ложь



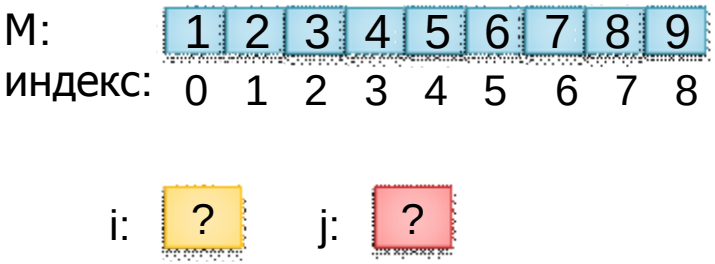
На экране:

```
1   2   3
4   5   6
7   8   9
|
```

3.4. Операторы C++

```
for (int i =0; i<3; i++)
{
    for (int j =0; j<3;j++)
            cout << M[i*3+j]<< '\t';
    cout << '\n';
}
```

Выходим из цикла



На экране:

```
1   2   3
4   5   6
7   8   9
|
```

3.4. Операторы C++

Операторы передачи управления:

- **goto** – безусловный переход к метке;
- **return** – выход из функции с возвратом результата;
- **break** – выход («досрочный») из цикла или переключателя;
- **continue** – переход к следующей итерации цикла.

3.4. Операторы C++

Пример использования break и continue.

```
// Выводить, пока не откажется пользователь, простые числа
//  начиная с 11 на экран
#include <iostream.h>
#include <conio.h> // для getch()
main()
{
   int i = 10;
   while(1) // бесконечный цикл
   {
      i++;
      if (!(i%2 && i%3 && i%5 && i%7))  continue;
      cout << i << '\t';
      char ans = getch();  //ожидает нажатия клавиши на клавиатуре и возвращает код
     if (ans==27)  break; // 27 – код клавиши ESC
   }
}
```

3.4. Операторы C++

Результат (прервано на 793):

11	13	17	19	23	29	31	37	41	43
47	53	59	61	67	71	73	79	83	89
97	101	103	107	109	113	121	127	131	137
139	143	149	151	157	163	167	169	173	179
181	187	191	193	197	199	209	211	221	223
227	229	233	239	241	247	251	253	257	263
269	271	277	281	283	289	293	299	307	311
313	317	319	323	331	337	341	347	349	353
359	361	367	373	377	379	383	389	391	397
401	403	407	409	419	421	431	433	437	439
443	449	451	457	461	463	467	473	479	481
487	491	493	499	503	509	517	521	523	527
529	533	541	547	551	557	559	563	569	571
577	583	587	589	593	599	601	607	611	613
617	619	629	631	641	643	647	649	653	659
661	667	671	673	677	683	689	691	697	701
703	709	713	719	727	731	733	737	739	743
751	757	761	767	769	773	779	781	787	793

3.4. Операторы C++

Функции

Формат определения:

[<тип возвращаемого значения>] имя([<список параметров>])

{

<тело функции>

}

- *тип возвращаемого значения*: если функция ничего не возвращает – void (или может быть опущен);
- *список параметров*: если не пуст, имеет формат:

  тип имя, тип имя, и т.д.
- *тело функции*:
  - если функция ничего не возвращает, может включать оператор **return** для «досрочного» выхода
  - если функция возвращает результат, **должно** включать оператор return <результат>;

3.4. Операторы C++

Пример:

Функция, возвращающая элемент массива, ближайший к заданному числу:

```
#include <math.h> // для fabs()
double nearest(double M[ ], int n, double num)
{
    double error = fabs(M[0]-num); //модуль разности
    int j = 0;
    for(int i = 1; i<n; i++)
    {
            double nerror = fabs(M[i]-num);
            if (nerror<error)    {error = nerror; j=i;}
    }
  return M[j];
}
```

3.4. Операторы C++

Вызов функции – результат применения операции () к ее имени. В скобках приводится список фактических параметров без указания их типов. При вызове компилятор проводит контроль соответствия типов фактических параметров типам формальных параметров, приведенным в определении функции или ее прототипе. Если типы не совпадают и не приводимы, фиксируется ошибка.

```
//Вызов функции nearest()
#include <iostream.h>
main()
{
    double M[]={2, 3.5, 4.1, 8, 2.67};
    cout << nearest(M,5,4);
}
```

Результат:

4.1

3.4. Операторы C++

Прототипы функций используются для того, чтобы сообщить компилятору о функции, которая еще не определена (до ее вызова) и будет определена позднее в этом же модуле, либо ее определение находится в другом модуле.

Прототип – это заголовок функции с точкой с запятой в конце. В прототипе необязательно указывать имена формальных параметров

```
//Прототип функции nearest()
double nearest(double M[ ], int n, double num);
//или так:
double nearest(double [ ], int, double);
```