

Ответы по ОАиПу

Вопросы 37-54

C'est la fin.

Это предварительная версия. Здесь косноязычие может
встретиться, како и очепятки,

и кри^Вое.

форматирование.

Содержание

37	Операции над указателями разного порядка	4
38	Арифметика указателей	4
39	Массивы переменных размеров. Аллокаторы памяти	6
40	Рекурсивные алгоритмы	8
41	Алгоритмы сортировки. Асимптотическая сложность	9
42	Функции языка C для работы со строками	11
43	Методы языка C++ для работы со строками	14
44	Декларация структур (struct) в C/C++. Отличия в декларации	14
45	Инициализация и доступ к элементам структуры. Выравнивание	16
46	Вложенные структуры и массивы структур	18
47	Указатели на структуры	19
48	Объединения и битовые поля	20
49	Локальные и глобальные переменные	22
50	Автоматические переменные	24
51	Внешние и статические переменные, особенности их реализации	24
52	Символические константы: #define. Включение файла: #include	27

53	Директивы препроцессора: <code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> , <code>#else</code> , <code>#endif</code>	30
54	Понятие алгоритма. Введение в алгоритмизацию	34

37 Операции над указателями разного порядка

38 Арифметика указателей

Указатель — переменная, значением которой является адрес памяти. В памяти указатель представляется как беззнаковое целое длины, равной длине машинного слова. *Одноименными* будем называть указатели, которые указывают на переменные одинакового типа. По стандарту арифметические операции нельзя совершать над указателями на `void`¹ и функции, хотя GCC разрешает эти операции в качестве расширения.

К арифметическим операциям над указателями относятся следующие операции:

1. **Сложение с числом.** К указателю можно прибавлять как положительные, так и отрицательные числа. Эта операция коммутативна. При прибавлении к указателю `a` числа `n`, значение адреса памяти, на который указывает указатель, увеличивается на `n*sizeof(int)`². Для удобства выражение `*(a + n)` можно записывать как `a[n]`. Пример (выведет ОК):

```
int n = 4;
if (((unsigned long) (a+n)) ==
    ((unsigned long) a) + (n*sizeof(int))) {
    std::cout << "OK\n";
}
```

2. **Инкремент и декремент** прибавляет и отнимает единицу к указателю (**не** к адресу!) по правилу, указанному выше, соответственно. При этом, как и с обычными числами, префиксные

¹точнее, на неполные типы (incomplete types — types that describe objects but lack information needed to determine their sizes)

²или не `int`, а нужного типа.

операторы возвращают измененное значение, а постфиксные — неизмененное.

Единственное, следует обратить внимание на приоритет оператора инкремента (декремента) и оператора разыменования *. При использовании как префиксного, так и постфиксного оператора сначала выполнится инкремент (декремент) и только потом — разыменование.

```
int c[2]{5, 10};
int *b = c;
// 5 (b до изменения указывает на c[0])
std::cout << *b++ << '\n';
// Теперь b указывает на c[1]
// 10 | 5 10
std::cout << *b << " | " << c[0] << ' '
               << c[1] << '\n';
```

3. **Вычитание числа из указателя** работает так же, как и прибавление к указателю числа, противоположного по знаку.
4. **Вычитание одноименных указателей** $a - b$ возвращает такое число n , что $a == b + n$. Число n имеет тип `ptrdiff_t` из `<ptrdiff_t>`, который является `typedef`'ом от какого-то³ обычного *знакового* целочисленного типа.

```
int a[20]{};
int *b = a;
int *c = a + 12;
// 12
std::cout << c - b << '\n';
// -12
std::cout << b - c << '\n';
```

Если результат вычитания настолько большой, что не может поместиться в `ptrdiff_t`, то UB.

³implementation-defined

39 Массивы переменных размеров. Алло- каторы памяти

Массив — это определённое число ячеек памяти, расположенных непосредственно друг за другом. Массив позволяет хранить несколько значений одинакового типа.

Поскольку число элементов массива переменной длины и, следовательно, его размер заранее неизвестны, память для него обычно выделяется в куче. Доступ к элементам массива при этом осуществляется через указатель на первый элемент массива при помощи арифметики указателей или оператора `[]`

Существует три различных подхода к выделению динамической памяти:

1. Функции `malloc`, `realloc`, `calloc`. Для первоначального выделения памяти можно использовать любую из этих функций. Для изменения размера выделенного участка памяти необходимо использовать функцию `realloc`. Она либо расширяет старый участок памяти, либо выделяет память заново, копирую при этом туда нужное число элементов (минимум от старого и нового размеров) и освобождая после этого старый участок.

```
// Выделение памяти
int *array = (int*) malloc(sizeof(int) * array_size);

// Изменение размера
array = (int*) realloc(sizeof(int) * new_array_size);

// Освобождение памяти
free(array);
```

2. Оператор `new` Язык C++ не предоставляет аналога функции `realloc` из C. Поэтому для изменения размера массива необ-

ходимо выделить память заново и вручную скопировать (переместить, если в массиве лежит что-то сложнее `int`ов) в новую область памяти существующие элементы массива.

```
// Выделение памяти
int *array = new int[array_size];

// Изменение размера
int *new_array = new int[new_array_size];
int copy_size = std::min(array_size, new_array_size);
for (int i = 0; i < copy_size; ++i) {
    new_array[i] = array[i]; // Или std::move(array[i]);
}
delete[] array;
array = new_array;

// Освобождение памяти
delete[] array;
```

Работа с памятью в стиле языка Си в некоторых случаях⁴ позволяет ускорить перевыделение памяти⁵, поскольку избегает копирования всех элементов массива, однако возлагает на программиста ответственность за ручной вызов деструкторов и `placement new`, если приходится работать с более сложными объектами.

Аллокатор — высокоуровневая абстракция над выделением и освобождением памяти, которая позволяет задать конкретный способ того, как будет выделяться память. Применение аллокаторов оправдано, когда программист знает, как именно его программа использует память. В таком случае аллокаторы позволяют сократить число системных вызовов и ускорить работу программы.

⁴Почти всегда?

⁵Также функции выделения памяти обычно устроены так, чтобы сократить число системных вызовов

40 Рекурсивные алгоритмы

Рекурсивная функция – такая функция, которая вызывает саму себя. Тривиальным примером рекурсивной функции может служить функция для вычисления чисел Фибоначчи, определяемых рекуррентным соотношением $f_n = f_{n-1} + f_{n-2}$, причем $f_1 = f_2 = 1$:

```
long FibRecursion(long n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return Fib(n - 1) + Fib(n - 2);
}
```

Математически доказуемо, что любой рекурсивный алгоритм можно реализовать с помощью цикла и обратно: любой циклический алгоритм можно реализовать с помощью рекурсии.

Обычно рекурсивные алгоритмы более наглядные и простые. Благодаря этому рекурсия нашла довольно широкое применение в различных алгоритмах (Quicksort, обход вершин графов).

Однако рекурсивные алгоритмы из-за накладных расходов на вызов функции обычно показывают худшую производительность⁶, чем циклические, несмотря на одинаковую асимптотику. Кроме того, для работы рекурсивных алгоритмов необходимо поддерживать стек вызовов. Но размер стека ограничен⁷, что накладывает ограничение на максимальную глубину рекурсии.

Из-за кривого дизайна (как в примере выше) выполнение функции может иметь экспоненциальную сложность. Например, для вычисления 10-го числа Фибоначчи эта функция два раза вычислит 9-ое число, для чего ей понадобится 4 раза вычислить 8-ое и т.д. Чтобы избежать повторных вычислений, применяют подход, называемый **мемоизацией**: после вычисления функции для заданного значения аргумента оно сохраняется в памяти, а при повторных запросах функция возвращает уже вычисленное значение. Применение

⁶и Time, и Space Complexity

⁷как?

мемоизации позволяет снизить алгоритмическую сложность до $O(n)$ (в данном примере), пожертвовав дополнительной памятью ⁸.

41 Алгоритмы сортировки. Асимптотическая сложность

Сортировка — процесс расположения элементов массива (последовательности) в определенном порядке, удобном для работы. Если отсортировать массив чисел в порядке убывания, то первый элемент всегда будет наибольшим, а последний наименьшим. Сортировки имеют важное прикладное значение. Например, с отсортированными данными иногда можно работать более эффективно, чем с неупорядоченными (бинарный поиск имеет сложность $O(\log n)$, а линейный — $O(n)$).

Алгоритм сортировки называется **устойчивым** (stable), если он сохраняет порядок следования элементов с совпадающим значением ключа — признака, по которому происходит сравнение.

Алгоритмическая сложность многих алгоритмов сортировки может зависеть от входных данных ⁹.

Можно доказать, что асимптотическая сложность сортировки, основанной на сравнениях, не может быть лучше, чем $O(n \cdot \log n)$. При этом существуют алгоритмы сортировки, которые используют знания о природе сортируемых данных и имеют сложность $O(n)$ в среднем, однако они могут быть неприменимы в общем случае.

Ниже приведены и кратко описаны некоторые алгоритмы сортировки. Для простоты будем считать, что мы сортируем массивы чисел по возрастанию.

Пузырьковая сортировка. Самый дурацкий алгоритм сортировки. Выполняет проходы по массиву до тех пор, пока массив не будет

⁸Поскольку рекурсия использует стек вызовов глубины n и сохраняет n значений, Space Complexity не изменится и составит $O(n)$, что все еще хуже циклического алгоритма.

⁹Здесь и далее, если не указано иное, за n принимается размер массива

отсортирован. Если во время прохода встретится пара элементов, которые непосредственно идут друг за другом и имеют неверный порядок, то они меняются местами. Time Complexity — $O(n^2)$, Space Complexity — $O(1)$.

Сортировка выбором. В первом проходе выбирает наименьший элемент массива и меняет его местами с первым. На следующем этапе проходит по массиву начиная со второго элемента и выбирает наименьший элемент из этой части массива, после чего меняет его местами со вторым элементом массива. Повторяет этот шаг до тех пор, пока не останется один элемент. Массив отсортирован. Time Complexity — $O(n^2)$, Space Complexity — $O(1)$.

Сортировка вставками. Перебираются элементы в неотсортированной части массива. Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находиться. Time Complexity — $O(n^2)$ в худшем случае и $O(n)$, если массив уже отсортирован. Space Complexity — $O(1)$.

Сортировка вставками имеет довольно маленькую константу, благодаря чему используется в функции `std::sort` для сортировки небольших массивов или маленьких частей больших массивов как часть следующего алгоритма.

Быстрая сортировка.

Сортировка слиянием. Разделяет исходный массив на два равных подмассива, после чего рекурсивно сортирует их по отдельности и объединяет. Массивы разделяются до тех пор, пока в них не останется одного элемента.

Алгоритм сортировки таков:

1. Если в массиве 1 элемент — завершиться.
2. Найти середину массива.
3. Посортировать первую половину.

4. Посортировать вторую половину.

5. Объединить массив.

Алгоритм объединения массивов:

1. Циклично проходим по двум массивам.

2. В объединяемый ставим тот элемент, что меньше.

3. Двигаемся дальше, пока не дойдем до конца обоих массивов.

Time Complexity: $O(n \log n)$, Space Complexity: $O(n)$.

На степике также упоминается **сортировка Шелла** и **пирамидальная сортировка** (она же Heapsort или сортировка кучей), но их суть кратко описать довольно затруднительно.

42 Функции языка C для работы со строками

Строки используются для представления текстовой информации. В языке C строки рассматриваются как массивы символов (`char`), заканчивающиеся специальным зарезервированным символом с кодом 0.

Прототипы функций для работы со строками в языке C находятся в заголовочном файле `<string.h>` (в языке C++ для работы с C-строками — `<cstring>`).

`size_t strlen(const char *s)` - определяет длину строки `s` без учёта нуль-символа.

Копирование строк

`char *strcpy(char *dst, const char *src)` — выполняет побайтное копирование символов из строки `src` в строку `dst`. Возвращает указатель `dst`. Программист должен удостовериться, что `dst` указывает на участок памяти достаточного размера.

`strncpy(s1,s2, n)` - выполняет побайтное копирование `n` символов из строки `s2` в строку `s1`. возвращает значения `s1` Конкатенация строк
`char* strcat(char *dst, const char *src)` - объединяет строку `src` со строкой `dst`. Результат сохраняется в `dst`.
`strncat(s1,s2,n)` - объединяет `n` символов строки `s2` со строкой `s1`. Результат сохраняется в `s1`

Сравнение строк

`strcmp(s1,s2)` - сравнивает строку `s1` со строкой `s2` и возвращает результат типа `int`: 0 –если строки эквивалентны, `>0` – если `s1<s2`, `<0` – если `s1>s2` С учётом регистра

`strncmp(s1,s2,n)` - сравнивает `n` символов строки `s1` со строкой `s2` и возвращает результат типа `int`: 0 –если строки эквивалентны, `>0` – если `s1<s2`, `<0` – если `s1>s2` С учётом регистра

`strcmp(s1,s2)` - сравнивает строку `s1` со строкой `s2` и возвращает результат типа `int`: 0 –если строки эквивалентны, `>0` – если `s1<s2`, `<0` – если `s1>s2` Без учёта регистра

`strnicmp(s1,s2,n)` - сравнивает `n` символов строки `s1` со строкой `s2` и возвращает результат типа `int`: 0 –если строки эквивалентны, `>0` – если `s1<s2`, `<0` – если `s1>s2` Без учёта регистра

Обработка символов

`isalnum(c)` - возвращает значение `true`, если `c` является буквой или цифрой, и `false` в других случаях

`isalpha(c)` - возвращает значение `true`, если `c` является буквой, и `false` в других случаях

`isdigit(c)` - возвращает значение `true`, если `c` является цифрой, и `false` в других случаях

`islower(c)` - возвращает значение `true`, если `c` является буквой нижнего регистра, и `false` в других случаях

`isupper(c)` - возвращает значение `true`, если `c` является буквой верхнего регистра, и `false` в других случаях

`isspace(c)` - возвращает значение `true`, если `c` является пробелом, и `false` в других случаях

`toupper(c)` - если символ `c`, является символом нижнего регистра, то функция возвращает преобразованный символ `c` в верхнем регистре, иначе символ возвращается без изменений.

Функции поиска

`strchr(s,c)` - поиск первого вхождения символа `c` в строке `s`. В случае удачного поиска возвращает указатель на место первого вхождения символа `c`. Если символ не найден, то возвращается ноль.

`strcspn(s1,s2)` - определяет длину начального сегмента строки `s1`, содержащего те символы, которые не входят в строку `s2`

`strspn(s1,s2)` - возвращает длину начального сегмента строки `s1`, содержащего только те символы, которые входят в строку `s2`

`strprbk(s1,s2)` - Возвращает указатель первого вхождения любого символа строки `s2` в строке `s1`

Функции преобразования

`atof(s1)` - преобразует строку `s1` в тип `double`

`atoi(s1)` - преобразует строку `s1` в тип `int`

`atol(s1)` - преобразует строку `s1` в тип `long int` Функции стандартной библиотеки ввода/вывода `<stdio>`

`getchar(c)` - считывает символ `c` со стандартного потока ввода, возвращает символ в формате `int`

`gets(s)` - считывает поток символов со стандартного устройства ввода в строку `s` до тех пор, пока не будет нажата клавиша `ENTER`

Функции для работы с Си-строками никогда не выделяют память, если оказывается, что размер буфера недостаточен. Это может привести к неопределенному поведению и ошибкам сегментации. О выделении достаточного размера памяти должен заботиться программист, который вызывает функцию.

43 Методы языка C++ для работы со строками

44 Декларация структур (struct) в C/C++. Отличия в декларации

Структура — производный тип данных, который представляет какую-то определенную сущность. Для определения структуры используется ключевое слово `struct`:

```
struct ИмяСтруктуры {  
    поля_структуры;  
};
```

Поля структуры — это переменные, доступ к которым можно получать через объект структуры с помощью оператора `.` или через указатель на объект структуры через оператор `->`.

В языке C, в отличие от C++, объявленная таким образом структура будет доступна под именем `struct ИмяСтруктуры` (в C++ — просто `ИмяСтруктуры`). Чтобы не писать слово `struct`, можно объявить псевдоним для типа структуры с помощью ключевого слова `typedef`. В C++ так тоже можно делать для обратной совместимости с Си. Как в C++, так и в C для полей структур можно задавать значения по умолчанию.

В языке C++ во всех структурах неявно объявляется конструктор и деструктор по умолчанию (если они не объявлены явно). Конструктор вызывается при объявлении (и/или инициализации) объекта структуры (также при вызове оператора `new`, а деструктор — когда объект покидает область видимости или вызывается оператор `delete`.)

```
#include <iostream>  
#include <cstdint>
```

```
struct Vector2 {
```

```

    float x;
    // Значение по умолчанию
    float y = 0;
};

typedef struct Vector2 vector2_t;

// typedef можно писать и сразу. Название структуры можно опускать
typedef struct {
    size_t size;
    char* str;
} string_t;

typedef struct Segment {
    Vector2 a;
    Vector2 b;
} segment_t;

int main() {
    // Обращение в стиле Си
    struct Vector2 a = {1.0, -3.0};
    vector2_t b = {1.0, 3.4};
    // Обращение в стиле C++
    Vector2 c = {-2.0, -0.4};

    // Анонимная структура. У нее нет названия,
    // но в остальном она работает как обычная структура.
    struct {
        double x;
        double y;
    } point = {.x = a.x, .y = a.y};
    // В строчке выше используется designated initializer,
    // который позволяет указывать названия полей, которые
    // инициализируются. (перед названием поля для этого)
    // ставится точка.

```

```

// В Си это было с незапамятных времен, а в C++
// стандартизировано лишь в C++20

// Обращение к полю x
std::cout << b.x + a.x + c.x << '\n';

// Использование псевдонима и динамического выделения памяти
segment_t *segment = new segment_t;
// Обращение через указатель на структуру
segment->a = a;
segment->b = c;
delete segment;
return 0;
}

```

45 Инициализация и доступ к элементам структуры. Выравнивание

Определение понятия структуры см. выше.

Инициализация

Инициализация структуры в языках C и C++ отличается. Так, в языке C++ присутствует конструктор по умолчанию — метод структуры, который неявно вызывается компилятором при создании объекта структуры. Он же вызывается (но уже фактически явно) и при выделении памяти с помощью оператора `new`. В языке C ничего подобного нет.

Конструктор по умолчанию в языке C++ инициализирует все поля значениями по умолчанию, которые можно указывать явно. Если значения явно не указано, то поля простых типов наподобие `int` или указатели инициализируются нулями, если структура располагает-

ся в статической памяти или в куче¹⁰ ; и мусором, если структура объявлена на стеке.

```
typedef struct {
    int a = 42;
    int c;
} ExampleStruct;

static ExampleStruct e1;

int main() {
    ExampleStruct e2;
    // 42 42
    std::cout << e1.a << ' ' << e2.a << '\n';
    // 0 <мусор>
    std::cout << e1.c << ' ' << e2.c << '\n';
}
```

Для инициализации полей структур в Си используется перечисление значений в порядке объявления полей. Также допустима инициализация с явным указанием пар поле-значение:

```
typedef struct Example {
    int a;
    float f;
};

// Обычная инициализация
Example e1 = {0, 0.4};
// designated initializer
Example e2 = {.a = 0, .f = 0.4};
```

Оба этих вида инициализации также поддерживаются языком Си (вторая — начиная с C++11). Кроме того, начиная с C++11 поддерживается еще один вид инициализации:

¹⁰при выделении памяти с помощью оператора `new`. Особо искушенные последователи Культа также знают о `placement new`, который может проинициализировать любой участок памяти (в т. ч. выделенный с помощью `malloc`.)

Example e3{0, 3.14};

Выравнивание

Обычно процессоры эффективнее работают, когда данные выравнены определенным образом. Это значит, что их адрес должен иметь специфическое значение (обычно кратное какой-либо степени числа 2). Отметим, что все простые типы должны быть выравнены по своему размеру, то есть их адрес в памяти должен быть кратен размеру этого типа. Так, адрес 4-байтного `int` должен быть кратен числу 4, а адрес переменной типа `char` может быть любым.

Поэтому при создании структур компилятор может добавлять неиспользуемые байты — **паддинги**.

46 Вложенные структуры и массивы структур

Бессмысленный вопрос.

Поле структуры может являться любой полный тип и, в частности, другая структура.

Структура не является полным типом до конца ее объявления, поэтому она не может содержать саму себя в качестве поля, поскольку это привело бы к тому, что такая структура должна иметь бесконечный размер. Однако структура может содержать указатель или ссылку на себя.

```
struct Node {
    int value;
    // Ошибка компиляции: Field has incomplete type
    Node next;
};
```

```
struct RefNode {
    int value;
```

```

    Node &next;
};

struct PtrNode {
    int value;
    Node *next = nullptr
};

```

Вложенная структура имеет такое же выравнивание, как и ее первое поле.

Массив структур объявляется точно так же, как и массив простых типов. Для создания и освобождения динамических массивов лучше использовать операторы `new []` и `delete[]`, поскольку они вызывают конструкторы и деструкторы и позволяют корректно инициализировать и освободить память полей, которые имеют производные типы (например, `std::string`).

```

struct Vector2 {
    int x;
    int y;
};

Vector2 static_array[100];

Vector2 dyn_array = new Vector2[100];
delete[] dyn_array;

```

47 Указатели на структуры

Какой-то это бессмысленный вопрос. Тут даже не о чем говорить.

Вся общая теория указателей (арифметика указателей, разыменования) также применима к указателям на структуры. Для доступа к элементам структуры надо использовать оператор `->`.

Если структура объявлена на куче, то память также надо очищать вручную.

48 Объединения и битовые поля

Объединения

Объединение — группирование переменных, которые разделяют одну и ту же область памяти.

Объявление объединения (типа объединения или шаблона объединения) начинается с ключевого слова `union`.

```
union ИмяТипаОбъединения {  
    Тип1 переменная_1;  
    Тип2 переменная_2;  
    ...  
    ТипN переменная_n;  
};
```

Где **ИмяТипаОбъединения** — непосредственно имя новосозданного типа;

переменная_1, ..., переменная_n — имена переменных, которые являются полями объединения. Эти переменные могут быть разных типов;

Тип1, ..., ТипN — типы полей объединения.

Размер объединения равен размеру самого большого поля.

Объединение относится к определенному участку памяти, в котором может находиться объект одно из типов, которые есть в объединении. При попытке перезаписать данные другим типом новые данные записываются вместо старых, из-за чего старые данные не могут корректно удалиться. Поэтому в `union` без дополнительных плясок с бубном нельзя записать «умный» тип наподобие `std::string`.

При обращении к полю объединения записанные в память данные будут интерпретироваться как данные того типа, к которому относится переменная, к которой происходит обращение. Нетрудно догадаться, что обращение к неправильному типу может вызвать UB.

```
// Можно объявлять и анонимные union.
// Тогда их поля попадут в ту же область
// видимости, где и объявлено объединение.
union {
    float f;
    int i;
} united;
// Одно из возможных побитовых представлений NaN по IEEE754.
united.i = 0x7f800001;
// nan (на x86_64 работает. как на других архитектурах, хз)
std::cout << united.f << '\n';
```

Резюмируя:

1. Объединения можно использовать для хранения одного из заданных типов данных. Чтобы знать, какой именно тип хранится в объединении, надо хранить эту информацию отдельно.
2. Объединения можно использовать для побитового преобразования одного типа в другой

В C++ для более безопасного хранения нескольких типов в одном участке памяти можно использовать `std::variant`, а для побитового преобразования (начиная с C++20) — `std::bit_cast`.

Битовые поля

Битовое поле позволяет задать длину поля структуры в битах. То есть, они как бы позволяют получать целочисленные типы произвольной (но не более машинного слова) длины. Битовые поля объявляются точно так же, как и обычные, но после имени поля через двоеточие указывается его длина.

```
struct ИмяСтруктуры {
    unsigned <char|int|short|long|long long> имя_поля: длина;
};
```

Обратите внимание, что только битовые поля могут иметь только целочисленные типы. Желательно, чтобы они были `unsigned`. Хотя использование обычных (знаковых) чисел не запрещается, оно, вообще говоря, может привести к неожиданным результатам (отрицательные числа) и даже к UB^{проверить?}, потому что способ представления отрицательных чисел до C++20 не был стандартизирован. С C++20 все компиляторы обязаны использовать дополнительный код.

Максимальное число, которое может поместиться в битовое поле длины n , равно $2^n - 1$. Обычно, если несколько битовых полей (неважно каких типов) объявлены друг за другом, то компилятор их ужимает так, чтобы они имели наименьший размер. При этом неиспользуемые в битовых полях биты становятся недоступными и превращаются в паддинг (a.k.a. ‘struct offset’).

В приведенном ниже примере (нумерация с нуля) биты 5, 6, 7 игнорируются и программа выведет 31 и 255. Битовое поле `c` позволяет получить доступ к первым пяти битам числа:

```
union {
    struct {
        unsigned char c: 5;
    } bitfield;
    unsigned char num;
};

// Все биты заполнены единицами
num = 255;
std::cout << (unsigned int) bitfield.c
           << ' ' << (unsigned int) num << '\n';
```

49 Локальные и глобальные переменные

Время хранения (storage duration)

Описанное в этом подразделе важно для понимания этого и последующих двух вопросов. **Время хранения** — это свойство объ-

екта, которое определяет минимальное возможное время жизни хранилища, содержащего объект¹¹.

Время хранения зависит от способа объявления объекта и может быть одним из следующих:

- **Статическое.** Все глобальные переменные и переменные, впервые объявленные с использованием спецификатора `static` или `extern`, которые не имеют потоковое время хранения. Хранилище живет на протяжении всего исполнения программы.
- **Потоковое (C C++11).** Все переменные, объявленные `thread_local`. Хранилище живет на протяжении жизни потока, в котором переменная создана. У каждого потока имеется своя уникальная копия объекта.
- **Автоматическое.** Смотреть ниже.
- **Динамическое.** Все объекты, созданные во время исполнения программы: объекты, созданные с помощью оператора `new` или динамически выделенные на куче, а также исключения (“allocated and deallocated in an unspecified way”).

Локальные переменные

Локальные переменные объявляются внутри тела функции или блока и доступны только изнутри функции или блока. Локальные переменные могут иметь *любое* время хранения.

Глобальные переменные

Глобальные переменные объявляются вне тела функции и доступны из любых функций текущей единицы трансляции или всей программы (см. вопрос про статические и внешние переменные). Глобальные переменные имеют статическое или потоковое время хранения.

¹¹https://en.cppreference.com/w/cpp/language/storage_duration

50 Автоматические переменные

Переменная имеет **автоматическое** время хранения, если выполнено одно из двух условий:

1. Переменная принадлежит области видимости блока (`{}`) и явно не объявлена `static`, `extern` или `thread_local` (см. следующий вопрос). Хранение этих переменных длится до тех пор, пока существует блок, в котором они объявлены.
2. Переменная является параметром функции. Хранение параметров функции длится до их уничтожения

Других автоматических переменных нет.

Автоматические переменные обычно хранятся на стеке, однако компиляторы с целью оптимизации могут помещать их в регистры процессора.

До C++11 можно было явно указать автоматическое время хранения с помощью ключевого слова `auto`. Начиная со стандарта C++11 ключевое слово `auto` приобрело новое значение: теперь оно позволяет явно не указывать тип переменной. В таком случае тип переменной выводится статически во время компиляции и не может быть изменен во время исполнения.

51 Внешние и статические переменные, особенности их реализации

Прежде, чем начать, надо отметить, что существует путаница в терминологии между понятиями ‘статическая переменная’ и ‘переменная, объявленная `static`’. Понятие **статическая переменная** более общее: под ним понимают переменную, имеющую статическое время хранения (`static storage duration`). Все такие переменные хранятся в статической области памяти на протяжении всего времени исполнения программы.

Переменной, объявленной `static` называют переменную, объявленную с использованием спецификатора `static`. Переменные, объявленные `static`, доступны только из той единицы трансляции, где они объявлены (внутреннее связывание, `internal linkage`). Все эти переменные являются статическими. По умолчанию статические переменные инициализируются нулями.

Переменные, объявленные `static` могут быть как глобальными, так и локальными. Про глобальные переменные разговор будет ниже, а пока остановимся на локальных. Поскольку локальные статические переменные хранятся не на стеке, они не очищаются при выходе из функции. Это позволяет сохранять состояние между вызовами функции. Статические локальные переменные инициализируются только один раз: при первой попытке сделать это.

```
#include <iostream>

void f(int val0) {
    static int saved = val0;
    std::cout << saved << ' ';
    ++saved;
}

int main() {
    // 4
    f(4);
    // 5
    f(0);
    // 6
    f(-3);
    std::cout << '\n';
}
```

Внешняя переменная — это переменная, которая имеет внешнее связывание (`external linkage`), то есть доступна из других единиц трансляции. Такими переменными являются глобальные переменные, определенные (`defined`) без спецификатора `static` или с спе-

цификатором **extern**. Внешние переменные хранятся в статической области памяти.

Хотя нормальные компиляторы (GCC, Clang, MSVC) поддерживают определение переменной с ключевым словом **extern** (см. переменную с ниже), оно предназначено для объявления переменной. Если определить значение этой переменной в нескольких единицах трансляции, то возникнет ошибка компоновки.

```
// obj.cc
// Внешние переменные, доступные из любых единиц трансляции
int a = 1;
int b = 2;
// Предупреждение GCC и Clang
extern int c = 3;
static float pi = 3.1416;

float GetPi() {
    return pi;
}

// main.cc
#include <iostream>

extern int a;
extern int b;

// Использование этих переменных приведет к ошибке компоновки
extern int d;
extern float pi;

extern float GetPi();

void Swap() {
    int buf = a;
```

```

    a = b;
    b = buf;
}

int main() {
    // Да, так тоже можно. Переменная 'с' (если не объявлена
    // в другом месте) будет доступна в теле функции 'main'.
    // Но в этой строке попытаться присвоить
    // переменной значение, программа не скомпилируется
    extern int c;

    // 1 2
    std::cout << a << ' ' << b << '\n';
    a += 23;
    // 24
    std::cout << a << '\n';
    Swap();
    // 2 24
    std::cout << a << ' ' << b << '\n';

    // 3.1416 3
    std::cout << GetPi() << ' ' << c << '\n';

    // Ошибка компоновки
    // std::cout << ' ' << pi << ' ' << d << '\n';
}

```

52 Символические константы: `#define`. Включение файла: `#include`

`#include` подставляет вместо себя содержимое указанного файла.
Синтаксис:

```
#include <файл>
```

или

```
#include "файл"
```

Подключаемый файл может находиться либо в той же директории, в которой лежит и исходный файл, либо в одном из системных путей (на Linux обычно `/usr/include/` и `/usr/include/c++/<версия GCC>/`).

При использовании синтаксиса с кавычками препроцессор сначала ищет файлы в той же директории, где находится сам файл, и только потом — в системных путях; а при использовании треугольных скобок — наоборот.

Можно добавить системные пути с помощью флага `-I` (GCC, Clang) или `/I` (MSVC) компилятора.

Хотя директива `#include` может использоваться для подключения произвольных файлов в произвольное место любого файла, делать это не рекомендуется. Директиву надо применять для подключения заголовочных файлов, содержащих объявления функций, структур, классов и т. д. Например, в заголовочном файле `cmath` стандартной библиотеки содержатся объявления математических функций `std::sqrt`, `std::sin`, `std::round` и других. В файле `iostream` содержатся функции и структуры для ввода-вывода информации в консоль.

Руководство Google по стилю кода рекомендует использовать `<>` для подключения системных заголовков и стандартной библиотеки; и `" "` для подключения всех остальных заголовков (за редкими исключениями, напр. `<Python.h>`).

#define позволяет определять символьные константы¹², вместо которых на этапе препроцессинга будет подставляться указанное выражение. Синтаксис таков:

```
#define идентификатор выражение
```

```
#define идентификатор(параметры, через, запятую) выражение
```

¹²для краткости я буду их называть макросами, но гипотетически Вадим может к этому придраться

где **идентификатор** — это имя макроса (любой валидный идентификатор), а **выражение** — то, что будет подставляться вместо **идентификатора**.

В первом случае директива создает символическую константу (object-like macro), вместо которой просто в лоб подставляется выражение.

Во втором случае директива создает функциональный макрос (function-like macro), в которые можно передать несколько аргументов. Они будут подставлены в выражение вместо параметров (см. пример). Поскольку в качестве аргумента макроса может выступать любое выражение, которое подставляется в макрос прямым текстом, «как есть», параметры при использовании следует оборачивать в скобки, чтобы избежать неожиданных результатов (ср. `MUL` и `CORRECT_MUL` в примере).

В выражении функционального макроса можно использовать два специальных оператора: **#параметр**, который оборачивает значение параметра в кавычки, превращая его в строковый литерал и **##**, который позволяет сконкатенировать параметр с чем угодно.

В **выражении** можно использовать другие макросы (и они будут корректно разворачиваться), а в процедурные макросы можно передавать другие макросы (в том числе и процедурные).

Отметим, что **выражение** может быть пустым (в таком случае вместо макроса подставится ничто). Также любой макрос можно впоследствии переопределить с помощью директивы **#define** либо разопределить с помощью директивы **#undef**. В коде после разопределения макроса компилятор будет вести себя так, как будто этого макроса никогда и не было; но в коде между **#define** и **#undef** этот макрос будет доступен.

В C++11 появилась возможность создавать макросы с переменным числом параметров. Это ужасно страшное колдунство. Подробнее смотри по ссылке: <https://en.cppreference.com/w/cpp/preprocessor/replace>

Пример:

```
#define QUESTION 52
#define ANSWER 42
#define SUM QUESTION + ANSWER
```

```

#define MERGE(x) v##x
#define MKSTRING(x) #x

#define MUL(x, y) x*y
#define CORRECT_MUL(x, y) (x) * (y)

int v42 = 24;
int vANSWER = -24;

// 10 94
std::cout << QUESTION - ANSWER << ' ' << SUM << '\n';
// 24 0
std::cout << MERGE(42) << ' ' << MERGE(42) + MERGE(ANSWER) << '\n';
// Hello OAiP
std::cout << MKSTRING>Hello OAiP) << '\n';
// 5 9
std::cout << MUL(3, 1 + 2) << ' ' << CORRECT_MUL(3, 1 + 2) << '\n';

#undef ANSWER
// Ошибка компиляции
std::cout << ANSWER << '\n';

```

53 Директивы препроцессора: #if, #ifdef, #ifndef, #else, #endif

Эти директивы препроцессора предназначены для условной компиляции, то есть они позволяют включить или выключить компиляцию определенных участков кода. Директивы создают ветвления на этапе препроцессора.

#if имеет следующий синтаксис:

```

#if <условие>
// скомпилировать код

```

`#endif`

Подобно оператору(?) ветвления в C++, включает компиляцию нижеледующего участка кода, если выполнено заданное условие. В условии можно использовать:

- Числовые и символьные константы (42, 'Y')
- Арифметические, побитовые и логические операции
- Макросы
- Оператор `defined(<макрос>)`. Если `<макрос>` был ранее по тексту программы определен с помощью директивы препроцессора `#define`, то оператор возвращает 1, иначе – 0
- Идентификаторы, которые не являются ранее определенными макросами. Вместо них подставляется число 0.

Если при вычислении записанного в условии выражения получится 0, то оно считается ложным; если же выйдет любое ненулевое число — истинным.

Следует отметить, что в директивах нельзя использовать оператор `sizeof`, поскольку препроцессор ничего не знает о типах.

Также существует директива препроцессора `#elif`, которая является полным аналогом конструкции `else if`.

`#ifdef`, `#ifndef` имеют одинаковый синтаксис:

```
#ifdef <макрос>
// скомпилировать код
#endif
```

```
#ifndef <макрос>
// скомпилировать код
#endif
```

Директива `#ifdef` включает компиляцию участка кода, если `<макрос>` был ранее определен с помощью директивы `#defined`.

Директива `#ifndef`, наоборот, включает компиляцию участка кода, если `<макрос>` **не** был ранее определен с помощью директивы `#defined`.

Примечание. Макросы можно разопределить с помощью директивы `#undef`

`#else` может использоваться только в связке с вышеназванными директивами:

```
#ifndef <условие>
    // скомпилировать, если <условие> выполнено
#else
    // скомпилировать, если <условие> не выполнено
#endif
```

Если оказывается, что условие директив `#if`, `#ifdef`, `#ifndef` ложно, то все то, что находится между директивами `#if` и `#else` игнорируется, а компилируется то, что находится между `#else` и `#endif`.

`#endif` обозначает конец ветвления.

Эти директивы можно использовать для определения операционной системы (проверка макросов `__linux__`, `__ANDROID__`, `_WIN32`, `macintosh`), различения C и C++ (макрос `__cplusplus`).

Также Руководство по стилю кода Google рекомендует использовать эти директивы для предотвращения повторного включения одного и того же файла (include guards):

```
#ifdef MY_FANCY_HEADER_H_
#define MY_FANCY_HEADER_H_ 1

int Sum(int a, int b);
int Odd(int a, int b);
```



```
typedef int(*FunctionPtr)(int, int);
```

```
#endif // MY_FANCY_HEADER_H_
```

Более сложный и бесполезный пример:

```
#ifndef __cplusplus
#include <stdio.h>
void SayHi() {
    printf("Thou usest C!\n");
}
#elif __cplusplus >= 202300L
#include <print>
void SayHi() {
    std::print("Your C++ version is {}, supergood!\n",
               __cplusplus);
}
#else
#include <iostream>
void SayHi() {
    std::cout << "Your C++ version is "
               << __cplusplus << ", kinda old :(\n";
}
#endif

int main() {
    SayHi();
    return 0;
}
```

<https://gcc.gnu.org/onlinedocs/cpp/If.html>

<https://sourceforge.net/p/predef/wiki/OperatingSystems/>

54 Понятие алгоритма. Введение в алгоритмизацию

Алгоритм — точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату.

Алгоритмизация — процесс построения алгоритма решения задачи, результатом которого является выделение этапов процесса обработки данных, формальная запись содержания этих этапов и определение порядка их выполнения.

Свойства алгоритмов

1. **Дискретность.** Процесс решения задачи должен быть разбит на последовательность отдельных шагов — простых действий, которые выполняются одно за другим в определенном порядке. Каждый шаг называется командой (инструкцией). Только после завершения одной команды можно перейти к выполнению следующей.
2. **Детерминированность** (определенность). Каждая команда алгоритма в отдельности и последовательность команд в целом должна быть точно и однозначно определена. Результат выполнения команды не должен зависеть ни от какой дополнительной информации. У исполнителя не должно быть возможности принять самостоятельное решение (т. е. он исполняет алгоритм формально, не вникая в его смысл). Благодаря этому любой исполнитель, имеющий необходимую систему команд, получит один и тот же результат на основании одних и тех же исходных данных, выполняя одну и ту же цепочку команд.
3. **Конечность и результативность.** Исполнение алгоритма должно завершиться за конечное число шагов; при этом должен быть получен результат.

4. **Массовость.** Алгоритм предназначен для решения не одной конкретной задачи, а целого класса задач, который определяется диапазоном возможных входных данных.
5. **Понятность.** Каждая команда алгоритма должна быть понятна исполнителю. Алгоритм должен содержать только те команды, которые входят в систему команд его исполнителя.

Способы описания алгоритмов

1. словесный;
2. формульно-словесный;
3. блок-схемный;
4. псевдокод;
5. структурные диаграммы;
6. языки программирования.

Пример словесного способа. *(деление обыкновенных дробей)*

В качестве входных данных даны две обыкновенные дроби. Для того чтобы разделить первую дробь на вторую, необходимо:

1. Числитель первой дроби умножить на знаменатель второй дроби.
2. Знаменатель первой дроби умножить на числитель второй дроби.
3. Записать дробь, числителем которой является результат выполнения шага 1, знаменателем — результат выполнения шага 2.

Описанный алгоритм применим к любым двум обыкновенным дробям. В результате его выполнения будут получены выходные данные — результат деления двух дробей (входных данных).

Алгоритмические языки

Алгоритмический язык — это искусственный язык (система обозначений), предназначенный для записи алгоритмов. Он позволяет представить алгоритм в виде текста, составленного по определенным правилам с использованием специальных служебных слов. Количество таких слов ограничено. Каждое служебное слово имеет точно определенный смысл, назначение и способ применения. При записи алгоритма служебные слова выделяют полужирным шрифтом или подчеркиванием.

В алгоритмическом языке используются формальные конструкции, но нет строгих синтаксических правил для записи команд. Различные алгоритмические языки различаются набором служебных слов и формой записи основных конструкций.

Алгоритмический язык, конструкции которого однозначно преобразуются в команды для компьютера, называется **языком программирования**. Текст алгоритма, записанный на языке программирования, называется **программой**.