Ответы по ОАиПу

Вопросы 37-54

C'est la fin.

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

и $\texttt{кр}u^{\text{в}}{}_{\text{о}}\text{е}$.

форматирование.

Содержание

37 Операции над указателями разного порядка

Для указателей разрешены следующие операции:

1. Присваивание `=`. Работает так же, как и с обычными переменными: операция `a = b` запишет в указатель `a` значение указателя `b` и вернет записанное значение. Присваивание разрешено только если оба операнда одноименные или один из них является указателем на `void`. С данными, на которые указывал `a`, ничего не будет. Если на эти данные не было других указателей, то доступ к ним будет утерян и освобождения памяти не последует.

2. Разыменование `*` возвращает тот объект, на который указывает указатель.

```
int *a = 4;
// 4
std::cout << a << '\n';
*a += 7;
// 11
std::cout << *a << '\n';
```

3. Взятие адреса самой перменной, которая является указателем `&` возвращает указатель на указатель (то есть указатель высшего порядка):

```
int a = 13;
int *b = &a;
int **c = &b;
int ***d = &c;
// ...
// Можно продолжать сколько угодно
```

```
// 13
std::cout << **c << '\n';
// Два одинаковых числа
std::cout << *d << ' ' <<c << '\n';
```

4. Сравнение значений указателей `<`, `<=`, `>`, `>=`, `==`, `!=`. Адреса, на которые указываают сравниваемые указатели сравниваются как обычные целые числа.

5. Арифметические операции.

Все эти операции работают для указателей любого порядка.

Указатели высших порядков обычно используются в для представления многомерных массивов[1]. Для получения конкретного элмента n-мерного массива достаточно n раз применить оператор `[]`[2]или сложения с разыменованиями, если вы желаете страдать|.

Перебор многомерных массивов осуществляется с помощбю вложенных циклов Ниже приведен пример сложения матриц.

```
/// Складывает матрицы `lhs` и `rhs` размера `m * n`.
/// Результат записывает в матрицу `res`.
void AddMatrices(int **lhs, int **rhs, int **res,
                 int m, int n)
{
  for (int i = 0; i < m; ++m) {
    for (int j = 0; j < n; ++j) {
      // Вместо того, что в левой части
      // для доступа к элементам массива `res`
      // можно использовать выражение `res[i][j]`
      *(*(res + i) + j) = lhs[i][j] + rhs[i][j];
    }
```

[1]адекватные люди (если они программируют не микроконтроллеры) для этого используют вложенные std::vector

[2]|

```
  }
}
```

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`[3] 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 указвает указатель, увеличивается на `n*sizeof(*a)`. Таким образом, указатель сдвигается на одну или несколько ячеек.

   Выражение `*(a + n)` также можно записывать как `a[n]`. С точки зрения языка обе записи эквивалентны. Пример (выведет `OK`):

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

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

2. **Инкремент и декремент** прибавляет и отнимает единицу к указателю (**не** к адресу!) по правилу, указанному выше, соответственно. При этом, как и с обычными чисами, префиксные операторы возвращают измененное значение, а постфиксные — неизмененное.

   Единственное, следует обратить внимание на приоритет оператора инекремента (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 `<cstddef>`, который является `typedef`'ом от какого-то[4] базового *знакового* целочисленного типа.

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

[4] implementation-defined

```
// -12
std::cout << b - c << '\n';
```

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

39 Массивы переменных размеров. Аллокаторы 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 `[]`

В языке C++ память можно выделять двумя основными способами:

1. Функциии `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 освобждая 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[]` Оператор `new type[x]` позволяет выделети динамический массив из `x` элементов типа `type`, вызывая для каждого элемента конструктор по умолчанию. Для освобождения выделенного массива используется операватор `delete[]`, который не только освободит память, но и вызовет деструкторы всех элементов массива.

   Язык C++ не предоставляет аналога функции `realloc` из C. Поэтому для изменения размера массива необходимо выделить память заново и вручную переместить 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 случаях позволяет облегчить перевыделение памяти, поскольку избегает копирования всех элементов массива, однако возлагает на программиста

ответственность за ручной вызов деструкторов и placement new, если приходится работать с объектами производных типов.

Аллокатор — высокоуровневая абстракция над выделением и освобождением памяти, которая позволяет задать конекретный способ того, как будет выделяться память. Аллокатор должен предоставлять функционал выделения и освобождения выделенной им памяти.

Для непосредственного выделения памяти используются системные вызовы `mmap` на Linux; `GlobalAlloc`, `HeapAlloc` и др. на Windows. Они требуют переключения контекста на процессоре и передают управление ядру ОС, что является относительно дорогостоящей операцией. Из-за этого программисты обычно стремятся уменьшить число системных вызовов.

В случае выделения памяти этого можно достичь, если, например, обращаться к системным вызовам только для выделения достаточно больших участков памяти, которые распределяются уже функциями, которые работают в пространстве пользователя.

Примерно такую стратегию используют функции `*alloc` и оператор `new`, поэтому в широком смысле их можно назвать аллокаторами.

Наконец, отметим, что аллокатор в общем случае не обязан выделять память на куче или вообще использовать системные вызовы. Ниже приведен пример примитивного аллокатора, который выделяет статическую память.

Он резервирует 100 000 байт статической памяти и хранит размер выделенной памяти, а вызове метода `Allocate` возвращает указатель на участок зарезрвированной при создании памяти и увеличивает значение переменной, хранящей размер выделенной памяти.

Для простоты реализации этот аллокатор не может переиспользовать освобожденную память, поэтому функция `Deallocate`, которая должна освобождать участок памяти, ничего не делает. Вся память, занятая аллокатором будет атоматически осовобождена при завершении исполнения прграммы.

```
#include <cmath>
#include <iostream>

struct StaticAllocator {
  static constexpr const size_t kPoolSize = 100'000;

  /// Выделяет память размером `size`.
  /// В случае неудачи возвращает `nullptr`,
  /// иначе - указатель на выделенную память.
  void *Allocate(size_t size) {
    if (allocated_ + size <= kPoolSize) {
      void *result = pool_ + allocated_;
      allocated_ += size;
      return result;
    }
    return nullptr;
  }
  /// Освобождает выделенный указатель `ptr`
  void Deallocate(void *ptr) {
    /// nop
  }

private:
  char pool_[kPoolSize];
  size_t allocated_ = 0;
};

static StaticAllocator allocator;

int main() {
  // Массив из 3 int'ов
  int *a = (int *)allocator.Allocate(3 * sizeof(int));
  std::cin >> a[0];
  std::cin >> a[1];
  a[2] = a[0] + a[1];
```

```
  std::cout << a[2] << '\n';

  // Один double
  double *b = (double *)allocator.Allocate(sizeof(double));
  std::cin >> *b;
  std::cout << std::sqrt(*b) << '\n';

  // Освобождение памяти
  allocator.Deallocate(a);
  allocator.Deallocate(b);
}
```

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 $f_n = f_{n-1} + f_{n-2} \quad (n \geq 3)$, причем $f_1 = f_2 = 1$:

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

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

```
long FibLoop(long n) {
  long f1 = 1;
  long f2 = 1;
  for (int i = 1; i <= n; ++i) {
```

```
    f1 += f2;
    std::swap(f1, f2);
  }
  return f2;
}
```

Рекурсия используется во многих алгоритмах, например в сортировках Quicksort и Mergesort, в обходе графов (поиск в глубину).

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

Из-за кривого дизайна (как в примере выше) выполнение рекурсивной функции может иметь экспоненциальную сложность. Например, для вычисления 10-го числа Фибоначчи эта функция два раза вычислит 8-ое число, для чего ей понадобится 4 раза вычислить 7-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 мемоизации позволяет снизить алгоритмическую сложность до $O(n)$ (в данном примере), пожертвовав дополнительной памятью.

```
#include <cstdint.h>

uint64_t FibMem(uint64_t n) {
  // Число Фибоначчи F(94) уже не
  // помещается в uint64_t
  if (n >= 94) {
    return 0;
  }
  static uint64_t memory[95] {0};
  if (n == 1 || n == 2) {
```

```
    memory[n] = 1;
    return 1;
  }
  if (memory[n] == 0) {
    memory[n] = FibMem(n - 1) + FibMem(n - 2);
  }
  return memory[n];
}
```

41 Алгоритмы сортировки. Асимптотическая сложность

Сортировка — процесс расположения элементов массива (последовательности) в определенном порядке, удобоном для работы. Если отсортировать массив чисел в порядке возрастания, то первый элемент всегда будет наименьшим, а последний — наибольшим. 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 [5].

Можно доказать, что асимптотическая сложность сортировки, основанной на сравнениях, не может быть лучше, чем $O(n \cdot \log n)$. При этом существуют алгоритмы сортировки (например, Radix Sort и Bucket Sort), 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 сор-

[5] Здесь и далее, если не указано иное, за n принимается размер массива

тировки. Для простоты будем считать, что мы сортируем массивы чисел по возрастанию.

Пузырьковая сортировка. Самый примитивный алгоритм сортировки. Выполняет проходы по массиву до тех пор, пока массив не будет 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. Затем проходит по массиву начиная с третьего элемента, находит минимальный и меняет его с третьим и т. д. Этот шаг повторяется до тех пор, пока число шагов не совпадет с длиной исходного массива. Time Complexity — $O(n^2)$, Space Complexity — $O(1)$.

Сортировка вставками. В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. Time Complexity — $O(n^2)$ в худшем случае и $O(n)$, если массив уже отсортирован. Space Complexity — $O(1)$.

Алгоритм можно ускорить, если для поиска позиции элемента в отсортированной части массива использовать бинарный поиск вместо линейного.

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. Time Complexity — $O(n \log n)$ в среднем, $O(n^2)$ в худшем (если входной массив уже отсортирован) случае. Space Complexity — $O(1)$. Быстрая сортировка функционирует по принципу «разделяй и властвуй»:

1. Массив $a[l \dots r]$ (l — индекс самого первого, r — самого последнего элемента) разбивается на два подмассива $a[l \dots q]$ и $a[q+1 \dots r]$, таких что каждый элемент $a[l \dots q]$ меньше или равен $a[q]$, который в свою очередь, не превышает любой элемент подмассива $a[q+1 \dots r]$. *Индекс опорного элементаq* вычисляется в ходе процедуры разбиения.

2. Подмассивы $a[l \dots q]$ и $a[q+1 \dots r]$ сортируются рекурсивно.

Распространенной является функция разбиения Хоара, которая выбирает средний элемент массива в качестве опорного. Однако так делать не обязательно. В качестве опорного можно выбирать абсолютно любой элемент массива.

Если кому-либо известен алгоритм функции разбиения, то он может злонамеренно соорудить такой массив, на котором функция быстрой сортировки уйдет в $O(n^2)$ и/или возникнет переполнение стека. Чтобы избежать этого, в качестве опорного можно выбирать случайный элемент массива.

Ниже приведен алгритм Quicksort с разбиением Хоара.

```
#include <iostream>
#include <utility>

/// a - массив, который сортируется
/// l - левая граница сортируемого отрезка
/// r - правая граница
int Partition(int *a, int l, int r) {
  int v = a[(l + r) / 2];
  int i = l;
  int j = r;
  while (i <= j) {
```

```
    while (a[i] < v) {
      ++i;
    }
    while (a[j] > v) {
      --j;
    }
    if (i >= j) {
      break;
    }
    std::swap(a[i++], a[j--]);
  }
  return j;
}

void Quicksort(int *a, int l, int r) {
  if (l < r) {
    int q = Partition(a, l, r);
    Quicksort(a, l, q);
    Quicksort(a, q + 1, r);
  }
}

int main() {
  int a[7] = {5, 10, -2, -3, 0, 1, 7};
  Quicksort(a, 0, 6);
  for (int i = 0; i < 7; ++i) {
    std::cout << a[i] << ' ';
  }
  std::cout << '\n';
}
```

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 оста-

нется 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 участко 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

stricmp(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 с является буквой или цифрой, и false в других случаях

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

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

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

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

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

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

Функции поиска

strchr(s,c) - поиск первого вхождения символа с в строке s. В случае удачного поиска возвращает указатель на место первого вхождения символа с. 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 с 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 ощибкам 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. 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). Коснтруктор 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 intializer,
  // 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';

  // Ипользование 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 струтруры 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 располагается в статической памяти или в куче[6] ; 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;
};
```

[6]при 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.)

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

Оба этих вида инициализации также поддерживаются языком C++ (вторая — начиная с C++20). Кроме того, начиная с C++11 поддерживается еще один вид инициализации:

```
Example e3{0, 3.14};
```

Выравнивание

Обычно процессоры эффективнее рабтают, 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 однобайтного типа `char` может быть любым. Выравнивание всей структуры равно выравниванию ее первого поля.

В заголовочном файле `<cstddef>` определен макрос `offsetof(type, member)`, значение которого равно отступу поля `member` структуры `type`.

Определить выравнивание типа (начиная с C++11), вообще говоря, можно с помощью оператора `alignof`, аналогичного оператору `sizeof`.

С целью выравнивания при создании структур компилятор может добавлять неиспользуемые байты — **паддинги**.

Основными компиляторами (MSVC, GCC, Clang) поддерживается нестандартная директива `#pragma pack(`N`)`, которая позволяет ограничить максимальное выравнивание полей структуры N байтами, где $N \in \{1, 2, 4, 8, 16\}$. В частности, `#pragma pack(1)` полностью отключаент выравнивание.

```
#include <cstddef>
```

```
#include <iostream>

struct S {
  char   m0;
  double m1;
  short  m2;
  char   m3;
};

#pragma pack(1)
struct SPacked {
  char   m0;
  double m1;
  short  m2;
  char   m3;
};

int main() {
  std::cout
    // 24
    << "S:          " << sizeof(S) << '\n'
    // 0
    << "char   m0 = " << offsetof(S, m0) << '\n'
    // 8
    << "double m1 = " << offsetof(S, m1) << '\n'
    // 16
    << "short  m2 = " << offsetof(S, m2) << '\n'
    // 18
    << "char   m3 = " << offsetof(S, m3) << "\n\n";

  // 8 4
  std::cout
    << alignof(double) << ' '
    << alignof(int) << "\n\n";
```

```
  std::cout
    // 12
    << "SPacked:    " << sizeof(SPacked) << '\n'
    // 0
    << "char   m0 = " << offsetof(SPacked, m0) << '\n'
    // 1
    << "double m1 = " << offsetof(SPacked, m1) << '\n'
    // 9
    << "short  m2 = " << offsetof(SPacked, m2) << '\n'
    // 11
    << "char   m3 = " << offsetof(SPacked, m3) << '\n';
}
```

46 Вложенные структуры и массивы структур

Бессмысленный вопрос.

Полем 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
};

struct List {
  PtrNode head;
  PtrNode tail;
};
```

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 освобождать 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

Совершенно 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 стурктура 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 запихнуть «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 IEE754.
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 побитовго преобразования (начиная с C++20) — `std::bit_cast`.

Битовые поля

Битовое поля позволяют 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 внимение, что только битовые поля могут иметь только целочисленные типы. Желательно, чтобы они были `unsigned`. Хотя использование обычных (знаковых) чисел не запрщается, 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 заполенены 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 объекта, которое определяет минимальное возможное время жизни хранилища, содержащего объект[7].

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 перменные, 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.

[7] `https://en.cppreference.com/w/cpp/language/storage_duration`

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 пременные имеют статическое или потоковое время хранения.

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 пора, пока существует блок, в котором они объявлены.

2. Переменная является параметром функции. Хранение параметров функции длится до их уничтожения при выходе из функции.

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 связывние, internal linkage). Все эти переменные являются статическими. По умолчанию статические переменные инициализируются улями.

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 спецификатором `extern`. Внешние переменные хранятся в статической области памяти.

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

```
// obj.cc
// Внешиние 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 ‘c‘ (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 константы[8], вместо которых на этапе препроцессинга будет подставляться указанное выражение. Синтаксис таков:

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

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

[8]для краткости я буду их называть макросами, но гипотетически Вадим может к этому придраться

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 нижеследующего 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 ничегно 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 алгноритмов

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 являет 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).

Алгоитмические 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**.