Ответы по ОАиПу

6 января 2025 г.

Содержание

37 Операции над указателями разного порядка

38 Арифметика указателей

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 `[]`

Существует три различных подхода к выделению динамической памяти:

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` Язык C++ не предоставляет аналога функции `realloc` из C. Поэтому для изменения размера массива необходимо 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 случаях[1] позволяет ускорить перевыделение памяти[2], 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 конекретный способ

[1] Почти всегда?

[2] Также функции выделения памяти обычно устроены так, чтобы сократить число системных вызовов

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.

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 раходов на вызов функции обычно показывают худшую производительность[3], чем циклические, несмотря на одинаковую асимптотику. Кроме того, для работы рекурсивных алгоритмов необходимо поддерживать стек вызовов. Но размер стека ограничен[4], что накладывает ограничение на максимальную глубину рекурсии.

Из-за кривого дизайна (как в примере выше) выполнение функции может иметь экспоненциальную сложность. Например, для вы-

[3] и Time, и Space Complexity

[4] как?

числения 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 мемоизации позволяет снизить алгоритмическую сложность до $O(n)$ (в данном примере), пожертвовав дополнительной памятью [5].

41 Алгоритмы сортировки. Асимптотическая сложность

Сортировка — процесс расположения элементов массива (последовательности) в определенном порядке, удобоном 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 [6].

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.

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

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

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 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, блягодаря 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 ихпо 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: $O(n)$.

Сортировка Шелла.

Сортировка кучей.

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 —

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:

- Linux(компилятор GCC):
  - `/usr/include/c++/14.2.1`
  - `/usr/include/c++/14.2.1/x86_64-pc-linux-gnu`
  - `/usr/include/c++/14.2.1/backward`
  - `/usr/lib/gcc/x86_64-pc-linux-gnu/14.2.1/include`
  - `/usr/local/include`
  - `/usr/lib/gcc/x86_64-pc-linux-gnu/14.2.1/include-fixed`
  - `/usr/include`

  Компилятор Clang использует те же самые пути (в том числе и GCC), но добавляет один путь: `/usr/lib/clang/18/include`

При использовании синтаксиса с кавычками препроцессор сначала ищет файлы в той же директории, где находится и файл, и только потом — в системных путях; а при использовании треугольных скобок — наоборот. Также отметим, что можно добавить системные пути с помощью флага `-I` (GCC, clang) или `/I` (MSVC) компилятора. Руководство по стилю кода Google рекомендует использование

<> для подключения заголовков сторонних и стандартной библиотек и "" для подключения заголовков текущего проекта.

Как правило, директива `#include` 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.

#define позволяет определять символьные константы[7] (англ. macro), вместо которых 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 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.

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

В `выражении` можно использовать другие макросы, 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`. Тогда 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. Эта 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';
// All human beings are born free and equal
std::cout << MKSTRING(All human beings are born free and equal) << '\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