Ответы по ОАиПу

Вопросы 37-54

C'est la fin.

15 января 2025 г.

Содержание

1. Основные характеристики языка C++

C++ - это компилируемый статически типизированный язык программирования общего назначения. Язык программирования — это набор формальных правил, по которым пишутся программы.

Язык C++ **является**:

1. Компилируемым - программы, написанные на C++, перед выполнением сперва преобразуются в целевой (машинный) код целевой платформы - компилируется; за это отвечает специальная программа - компилятор. В результате получается исполнимый модуль, который уже может быть запущен на исполнение как отдельная программа;

2. Статически типизированным - за каждой переменной закреплен определенный **тип** - класс данных, характеризуемый членами класса и операциями, которые могут быть к ним применены. Тип переменной задается единожды при ее объявлении и не может быть изменен;

3. Слабо типизированным - значения разных, порой несвязных, типов в C++ можно приводить друг к другу встроенными в язык методами;

4. Высокоуровневым - программы на C++ проще в понимании человеком, чем с программы в машинных кодах и на языке ассемблера;

5. Мультипарадигмальным - C++ поддерживает несколько различных парадигм программирования - совокупностей идей и понятий, определяющих стиль написания программ, иными словами, парадигмы - подходы к программированию. В частности, C++ поддерживает: - Процедурное программирование - парадигма, при которой последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка; - Обобщенное программирование - парадигма, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание; - **Объектно-ориентированное программирование** - парадигма, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение;

6. Языком общего назначения - на C++ пишутся программы для различный сфер, начиная встраиваемыми системами и заканчивая разработкой игр. В качестве примера можно привести *драйверы* периферийных устройств, *операционные системы* и их компоненты, *браузеры*, *игры* и *игровые движки*, *базы данных*, *системы программирования*, в том числе *другие языки программирования* и библиотеки для них и т. д.

Язык C++ также обладает богатой стандартной библиотекой, включающей в том числе общеупотребительные структуры данных и алгоритмы.

C++ строго регламентирован Международной организации по стандартизации (ISO). На сегодняшний день выпущен стандарт C++23 и разрабатывается стандарт C++26.

Комментарии автора

TODO: Возможно, стоит добавить определения альтернативных подходов/терминов?

2. Отличия языка C++ от языка C

Язык программирования **C++** многое, в том числе синтаксис, унаследовал от **C**. Обратная совместимость с C также является одной из целей создателей языка. Однако между этими языками есть и существенные различия.

1. Различен подход к управлению динамической памятью. В C используются `malloc()` и `free()`, в C++ - `new` и `delete` (и их вариации).

2. Различны способы представления и работы со строками. В C под строкой понимается последовательность (точнее, массив - все элементы расположены в смежных ячейках памяти) символов `char`, оканчивающихся т. н. *null-терминатором* - символом с кодом *0*. В C++ для работы со строками стандартной библиотекой предоставлен тип `std::string`.

3. Различны возможности по организации кода. В C++ существует понятие **пространства имен** - это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т.д.). Пространства имен позволяют предотвращать конфликт имен (коллизии) - типы с одинаковым названием, но в разных пространствах имен считаются различными и доступ к ним однозначен. Помимо этого в C++ можно обращаться к типам структур без использования ключевого слова `struct`, что является **обязательным** в **C** и создает необходимость использования `typedef`.

4. Для косвенного обращения к данным помимо указателей (переменных, хранящих в качестве значения адрес ячейки памяти) в C++ существуют ссылки - их использование удобнее, поскольку не требует постоянного повторения оператора обращения через указатель (`*` или `->`).

5. В C++ реализована поддержка **объектно-ориентированного программирования** (ООП), при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. Можно определять поля и функции (точнее, такие функции называются методы), связанные с конкретным объектом (чаще всего это помещается в определении класса этого объекта). В тоже время встроенной поддержки ООП в **C** нет.

6. C++ позволяет писать более гибкий код с помощью перегрузки функций (методов) и операторов.

7. В C++ существует механизм обработки ошибок - исключения.

8. В современном стандарте C есть ключевые слова, которых нет в C++, например `restrict`, сигнализирующее о единственности указателя на заданную область памяти.

Идентификатор — это последовательность символов, используемая для обозначения переменной, функции или любого другого объекта. Ключевые слова - это предварительно определенные зарезервированные идентификаторы, имеющие специальные значения. Их нельзя использовать в качестве идентификаторов в программе.

3. Область применения и системы программирования языка C++

Язык **C++** получил широкое распространение в сферах с требованиями к быстродействию ПО, а также в областях, требующих работу с низкоуровневыми интерфейсами. На **C++** разрабатывают, в том числе (примеры тут)

1. Драйверы устройств;
2. Операционные системы и их компоненты;
3. Базы данных;
4. Другие языки программирования: компиляторы, интерпретаторы, программные библиотеки;
5. В целом системы программирования: редакторы исходного кода, в т.ч. IDE (интегрированная среда разработки), отладчики;
6. Прикладное ПО: браузеры, 3D-редакторы, программы для редактирования текста и видео;
7. Игры и игровые движки;
8. ...

Такое разнообразие в первую очередь обусловлено высокому *быстродействию* и *гибкости* программ, написанных на **C++**.

Система программирования - совокупность языка программирования и программных средств, обеспечивающих подготовку исходного кода программы, его перевод на машинный код, и последующую отладку. Иными словами системы программирования создаются для удобства работы пользователя с выбранным языком программирования.

Как правило, системы программирования включают в свой состав: - интегрированную среду разработки или программирования (Integrated Development Environment - IDE); - компилятор; - редактор связей или компоновщик; - библиотеки заголовочных файлов; - библиотеки классов и функций; - программы-утилиты.

Наиболее распространены следующие системы программирования (не включая IDE):

Набор инструментов (toolchain)	Компилятор	Компоновщик	Стандартная библиотека	Отладчик
GCC	g++	ld	libstc++	gdb
LLVM	clang++	lld	libc++	lldb
MSVC	cl.exe	link.exe	MSVC STL	Visual Studio Windows Debugger

Интегрированную Среду Разработки можно трактовать как среду в которой есть все необходимое для проектирования, запуска и тестирования приложений и где все нацелено на облегчение процесса создания программ.

Что требуется от IDE: - Способность IDE корректно «понимать» код. IDE должна уметь индексировать все файлы проекта, а также все сторонние и системные заголовочные файлы и определения (defines, macro). - IDE должна предоставлять возможность кастомизации команд для построения проекта, а так же где искать заголовочные файлы и определения. - Должна эффективно помогать в наборе кода, т.е. предлагать наиболее подходящие варианты завершения, предупреждать об ошибках синтаксиса и т.д. - Навигация по большому проекту должна

быть удобной, а нахождение использования быстрым и простым. - Предоставлять широкие возможности для рефакторинга: переименование и т.д. - Также необходима способность к генерации шаблонного кода — создание каркаса нового класса, заголовочного файла и файла с реализацией. Генерация геттеров/сеттеров, определения методов, перегрузка виртуальных методов, шаблоны реализации чисто виртуальных классов (интерфейсов) и т.д.

В качестве примеров можно привести

1. Microsoft Visual Studio;
2. JetBrains CLion;
3. Qt Creator.

3.1. Примеры

1. Драйверы в Windows;
2. Ядра ОС обычно не пишут на C++, а вот API и драйверы - могут. Например, Windows;
3. MySQL, свободная реляционная БД;
4. LLVM - библиотека и программная платформа для написания языков программирования - сама написана на C и C++;
5. VisualStudio разрабатывается на C# и C++;
6. Mozilla Firefox и Google Chrome, Blender, LibreOffice, Premiere Pro
7. Unreal Engine, Unity, Godot - все это на C++
8. ...

4. Исходные и объектные модули, процессы компиляции и связывания (линковка)

Компиляция — сборка программы, включающая: - трансляцию всех модулей программы, написанных на одном или нескольких исходных языках программирования высокого уровня и/или языке ассемблера, в эквивалентные программные модули на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера) или непосредственно на машинном языке или ином двоичнокодовом низкоуровневом командном языке; - последующую сборку исполняемой машинной программы, в том числе вставка в программу кода всех функций, импортируемых из статических библиотек и/или генерация кода запроса к ОС на загрузку динамических библиотек, из которых программой функции будут вызываться.

Соответственно, программа, осуществляющая компиляцию, называется **компилятором**.

Примеры компиляторов языка **C++**:

1. **g++**, компилятор из набора инструментов (англ. toolchain) GCC;
2. **clang++**, компилятор из набора инструментов LLVM;
3. **cl.exe** - программа-драйвер MSVC (Microsoft Visual C++).

Компоновщик (редактор связей, линкер, сборщик) — это программа, которая производит компоновку («линковку», «сборку»): принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль.

Примеры компоновщиков:

1. **ld** - из набора инструментов GCC;
2. **lld** - из набора инструментов LLVM;
3. **link.exe** - из набора инструментов MSVC.

Исходный модуль - программный модуль на исходном языке, обрабатываемый транслятором.

Объектный модуль - двоичный файл, который содержит в себе особым образом подготовленный исполняемый код, который может быть объединён с другими объектными файлами при помощи редактора связей (компоновщика) для получения готового исполняемого модуля, либо библиотеки.

Исполняемый модуль (исполняемый файл) — файл, который может быть запущен на исполнение процессором под управлением операционной системы.

Препроцессор — программа для обработки текста. Может существовать как отдельная программа, так и быть интегрированной в компилятор. В любом случае, входные и выходные данные для препроцессора имеют текстовый формат. Препроцессор преобразует текст в соответствии с директивами препроцессора. Если текст не содержит директив препроцессора, то текст остаётся без изменений.

В общем виде, сборка программы (C++) производится следующим образом:

1) Исходный модуль обрабатывается **препроцессором**.

   В этой фазе происходит текстовая обработка директив препроцессора (например, `#include ''foo/bar.h''` заменит строчку `#include ''foo/bar.h''` на содержимое файла по пути `./foo/bar.h`);

2) **Фаза трансляции**. **Компилятор** на основе исходного модуля (файл с расширением **.cpp**) с внесенными изменениями создает *объектный модуль*.

3) **Фаза компоновки**. Компоновщик собирает один или несколько *объектных модулей*, файлы *статических библиотек* и объединяет их в один исполняемый модуль.

4.1. Процесс компиляции

Компиляция состоит из следующих этапов: - Лексический анализ - объединение символов в лексемы; - Синтаксический анализ - построение лексем в дерево разбора; - Семантический анализ - проверка и построение семантической модели кода; - Оптимизация - перестроение программы для увеличения ее быстродействия без видимых побочных эффектов; - Генерация кода - создание итогового объектного модуля.

4.2. Процесс компоновки (линковки)

В объектном модуле сохраняется информация обо всех определенных функциях и глобальных переменных. Эта информация сведена в таблицу символов (англ. symbol table). При этом производятся необходимые искажения (англ. mangling) имен функций для предотвращения коллизий имен (возникающих, например, при перегрузке функций - в имя кодируются типы аргументов). По таблицам символов компоновщик разрешает межмодульные зависимости, в частности, подставляет реальные адреса функция в места их вызова. Аналогично линкуются и глобальные переменные.

5. Алфавит языка C++. Лексемы

5.1. Алфавит

Алфавит языка C++ для файлов исходного кода обязательно (по стандарту **C++23**) включает в себя:

1. Строчную базовую латиницу: `a-z`;
2. Прописную базовую латиницу: `A-Z`;
3. Арабские цифры: `0-9`;
4. Специальные знаки: `,.;:?!'"|/\~_^(){}[]<>#%&-=+*`;
5. Пробельные и управляющие символы (приведены также их *escape-последовательности - символы которые выталкиваются в поток вывода, с целью форматирования вывода или печати некоторых управляющих знаков C++*):
   - `\t` - табуляция;
   - `\v` - вертикальная табуляция;
   - `\f` - смена страницы (англ. *form feed*);
   - `\n` - перевод строки;
   - `\0` - null-символ;
   - `\b` - возврат на шаг;
   - `\a` - звуковой сигнал (англ. *bell*);
   - `\r` - перевод каретки;
   - ` ` - пробел.

Впрочем, компиляторы могут поддерживать и более расширенный алфавит, например GCC поддерживает также символы UTF-8, например кириллицу. Строки и комментарии могут состоять вообще говоря из любых символов, поддерживаемых платформой.

5.2. Лексемы (a.k.a. Tokens)

Лексема (иначе *токен*, от англ. token) - минимальный лексический элемент языка C++ на этапе компиляции.

Категории лексем: 1) Идентификаторы; 2) Ключевые слова; 3) Литерал; 4) Операторы; 5) Знаки пунктуации (`;` `,` `{}` `()` и т.д.).

Идентификатор - это произвольно длинная последовательность цифр, знаков нижнего подчеркивания букв латиницы верхнего и нижнего регистров (и большинства символов Unicode, если присутствует поддержка платформы), обозначающая имя какой-либо программной сущности (напр. переменной, типа, метки и т. д.).

Ключевое слово - это предварительно определенный зарезервированный идентификатор, имеющий специальные значение. Его нельзя использовать в качестве идентификатора в программе.

Литерал - это непосредственное значение (целочисленное, вещественное, символьное, логическое, литерал-указатель `nullptr`, строковое).

Оператор - элемент программы, который контролирует способ и порядок обработки объектов.

Знаки пунктуации сами по себе смысла не несут, однако они являются составными частями операторов, и иных синтаксических конструкций.

5.2.1. Идентификаторы

Идентификатор - это произвольно длинная последовательность цифр, знаков нижнего подчеркивания букв латиницы верхнего и нижнего регистров (и большинства символов Unicode, если присутствует поддержка платформы), обозначающая имя какой-либо программной сущности (напр. переменной, типа, метки и т. д.).

Пользовательские идентификаторы не могут начинаться с цифры и содержать внутри себя пробельные символы. Также пользовательский идентификатор не может совпадать с каким-либо ключевым словом языка C++. Помимо этого не рекомендуется создавать идентификаторы, начинающиеся с символа подчеркивания, поскольку они могут являться внутренней деталью реализации стандартной библиотеки C++ или определяемым компилятором макросом.

5.2.2. Ключевые слова

Ключевое слово - это предварительно определенный зарезервированный идентификатор, имеющий специальные значение. Его нельзя использовать в качестве идентификатора в программе.

6. Ключевые слова языка C++, их область применения

Примечание автора

НЕ НУЖНО ПИСАТЬ ВСЮ ТАБЛИЦУ. Она для справки, по большому счету. Просто выбрать пару ключевых слов и расписать. Даже лучше по-подробнее, чем здесь.

Ключевое слово - это предварительно определенный зарезервированный идентификатор, имеющий специальные значение. Его нельзя использовать в качестве идентификатора в программе.

Помимо ключевых слов, стандарт C++ начиная с **C++11** также из общей массы идентификаторов выделяет *идентификаторы со специальным значением* (англ. *identifiers with special meaning*). На сегодняшний день (стандарт **C++23**) их 4: - `final` (запечатывает иерархию наследования); - `import` (подключает модуль, начиная с C++20); - `module` (объявляет модуль, начиная с C++20); - `override` (переопределяет член родительского класса).

Эти особые идентификаторы являются ключевыми словами лишь в определенном контексте, в частности, вот пример их использования в обычном коде:

```
#include <iostream>

// пример использования идентификаторов со
// специальным значением в качестве обычных названий переменной
int main() {
  int final = 42;
  long import = 13;
  int module = final + import;
  bool override = true;

  std::cout << (override ? module : -1); // 55

  return 0;
}
```

Еще следует обозначить ключевые слова - заменители некоторых операторов. Они предназначены для платформ, поддерживающих только 6-битную ASCII, и потому не имеющую символов `~ & |` и т. д.

Ключевое слово	Альтернативное представление
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

6.1. Таблица ключевых слов в языке C++

Ключевое слово	Использование	Примечание/пример
`alignas`	Указывает выравнивание типа	Ошибкой было бы указать выравнивание меньше естественного.
`alignof`	Возвращает выравнивание типа	
`asm`	Вставка на языке ассемблера	Зависит от платформы.
`auto`	Спецификатор вывода типа автоматически	`auto i = 42; // i имеет тип int`
`bool`	Логический тип	
`break`	Принудительный выход из цикла/ветки `switch`	
`case`	Ветка оператора `switch`	
`catch`	Открывает блок захвата исключений	
`char`	Символьный тип	
`char8_t`	Символьный тип, строго 8 бит на символ	С **C++20**, для представления текста в кодировке UTF-8.
`char16_t`	Символьный тип, строго 16 бит на символ	Для представления текста в UTF-16.
`char32_t`	Символьный тип, строго 32 бита на символ	Для представления текста в UTF-32.
`class`	Объявляет тип класса	
`concept`	Объявляет концепт	С **C++20**. Концепты используются для ограничения параметров шаблонов.
`const`	Спецификатор константности (неизменяемости)	Может применятся к параметрам, указателям, локальным и глобальным переменным
`consteval`	Объявляет *немедленную функцию*	С **C++20**. Вызов такой функции обязан быть константой времени компиляции
`constexpr`	Функцию можно вычислить при компиляции	Ошибочно объявить никогда ни вычислимую при компиляции функцию `constexpr`
`constinit`	Утверждает статическую инициализацию	С **C++20**. Переменная `constinit` всегда должна инициализироваться статически. Только для глобальных переменных и `thread_local`
`const_cast`	Приведение типов с разными cv-квалификаторами	Позволяет добавить и/или убрать квалификаторы `const` и `volatile`
`continue`	Досрочно завершает текущую итерацию цикла	
`co_await`	Приостанавливает корутину до получения значения	С **C++20**.
`co_return`	Завершает корутину	С **C++20**.
`co_yield`	Приостанавливает корутину с возвратом значения	С **C++20**.
`decltype`	Возвращает тип заданного выражения на этапе компиляции	
`default`	Определяет ветку по умолчанию в операторе `switch`. Также указывает использовать реализацию по умолчанию	
`delete`	Явным образом удаляет сгенерированную компилятором реализацию по умолчанию	
`do`	Часть объявления цикла с постусловием	
`double`	Вещественный тип двойной точности	
`dynamic_cast`	Приведение типов в иерархии наследования	
`else`	Ветка иначе в условном операторе	
`enum`	Объявляет перечисление	
`explicit`	Отличает конструктор с одним параметром от перегруженного оператора приведения типов	
`export`	До **C++11** использовалось в шаблонах; после **C++20** используется для экспорта кода в модулях	
`extern`	Определяет *внешнюю компоновку*	Переменные и функции с внешней компоновкой доступны из этого объектного модуля в других.
`false`	Литерал *ложь*	
`float`	Вещественный тип одинарной точности	
`for`	Объявляет цикл с параметром и цикл по коллекции (*range-based for*)	
`friend`	Спецификатор видимости; позволяет дать другому типу (функции) доступ к приватным полям класса	
`goto`	Оператор безусловного перехода	
`if`	Объявляет условный оператор	
`inline`	Функция, помеченная inline будет встроена в местах вызова	Компилятор имеет право (по соображению сохранения двоичного интерфейса приложения, ABI) проигнорировать этот спецификатор.
`int`	Целый тип, обычно 4 байта в размере	
`long`	Целый тип, не меньше чем int	
`mutable`	Спецификатор, позволяющий изменять поле, даже если объект константен	
`namespace`	Определяет или подключает пространство имен	
`new`	Оператор выделения памяти	
`noexcept`	Спецификатор отсутствия исключений	Возвращает `true`, если выражение не бросает исключений. Также используется как часть объявления функции, чтобы обозначить, что она не бросает исключений.
`nullptr`	Литерал *0-ого указателя*	
`operator`	Используется в переопределении операторов	
`private`	Спецификатор видимости; член (наследование) недоступен(-о) за пределами класса	
`protected`	Спецификатор видимости; член (наследование) доступен(-о) только дочерним классам	
`public`	Спецификатор видимости; член (наследование) доступен(-о)	
`register`	Спецификатор локальной переменной/параметра	Показывает компилятору, что указанная переменная/параметр часто используется, и потому ее следует поместить в регистр процессора.
`reinterpret_cast`	Приведение несвязных типов	Не рекомендуется.
`requires`	Используется в концептах	
`return`	Возврат значения из функции и лямбда-выражения	
`short`	Целый тип, не больше, чем `int`	
`signed`	Показывает, что следующий целый тип знаковый	
`sizeof`	Оператор получения размера типа	
`static`	Определяет *внутреннюю компоновку*. Также объявляет статический член класса (для его вызова не нужен объект)	при внутренней компоновке глобальные переменные и функции недоступны извне объектного модуля.
`static_assert`	Определяет условие на этапе компиляции	В случае ложности условия, завершает компиляцию с ошибкой. Вторым аргументом можно передать строку - пользовательское сообщение об ошибке.
`static_cast`	Приведение типов	
`struct`	Объявляет структуру	
`switch`	Объявляет оператор выбора `switch`	
`template`	Объявляет шаблон	
`this`	Указатель на текущий объект (в методе)	
`thread_local`	Спецификатор, делающий переменную локальной для каждого потока	
`throw`	Бросает исключение	
`true`	Литерал *истина*	
`try`	Объявляет защищаемый блок оператора `try-catch`	
`typedef`	Дает существующему типу новое имя (т.е. псевдоним)	
`typeid`	Получает информацию о типе	
`typename`	Используется в шаблонах	
`union`	Объявляет объединение	
`unsigned`	Показывает, что следующий целый тип беззнаковый	
`using`	Подключает пространство имен, член пространства имен. Также может давать типам псевдонимы	
`virtual`	Создает виртуальный метод	Виртуальные методы можно переопределять в дочерних классах; то, какой конкретно метод вызовется, зависит от типа конкретного объекта, даже при приведении к родительскому типу.
`void`	Определяет тип *ничто*.	Нет и не может быть объектов этого типа. Также используется для функций, не возвращающих значения.
`volatile`	Спецификатор, показывающий, что переменная может изменится под влиянием внешних причин	`volatile` переменные не могут быть оптимизированы, поскольку компилятор не может строить о них предположения. Значения таких переменных для компилятора могут изменяться непредсказуемо.
`wchar_t`	Тип символа, хранит не меньше, чем `char`	Понимается как тип символов платформно-зависимой расширенной кодовой таблицы.

Ключевое слово	Использование	Примечание/пример
`while`	Участвует в объявлении циклов с пред- и постусловием	

Не считая альтернативные формы некоторых операторов и идентификаторы с особым значением, всего в **C++ 81 ключевое слово**.

19. Операции сдвига (побитовый, циклический)

Все числа в памяти представлены в двоичной форме. Операция побитового сдвига применяется для целочисленных значений и позволяет сдвинуть биты числа влево («) или вправо (»). Синтаксис:

6 » 1 — сдвиг вправо на один бит. 0110 → 0011. Результат — 3.
6 « 1 — сдвиг влево на один бит. 0110 → 1100. Результат — 12.

При сдвиге в какую-либо сторону, с противоположной стороны на месте сдвинутых появляются нули. Если число имеет в знаковом разряде еденицу (отрицательное число), то при сдвиге влево на месте сдвинутых появляются не нули, а единицы — происходит расширение знака. Для отрицательных чисел выполнение побитового сдвига не рекомендуется — лучше поиграться с преобразованием в беззнаковый.

Сдвиг влево эквивалентен умножению на два. Сдвиг вправо — делению на два.

При сдвиге информация о битах, ушедших за границу разрядности переменной (размер переменной в памяти) теряется полностью. Если это был значимый бит(1) — происходит потеря данных.

Циклический сдвиг — при котором биты, уходящие за границу разрядности не исчезают, а заменяют собой сдвигаемые с противоположной стороны. В C++ можно использовать данную реализацию:

Пусть дано число ***x*** надо совершить циклический сдвиг его битов на величину **d**.
Желаемый результат можно получить, если объединить числа, полученные при выполнении обычного битового сдвига в желаемую сторону на **d** и в противоположном направлении на разность между разрядностью числа и величиной сдвига. Таким образом, мы сможем поменять местами начальную и конечную части числа.

```
int32 rotateLeft(x, d: int32):
  return (x << d) | (x >>> (32 - d))

int32 rotateRight(x, d: int32):
  return (x >>> d) | (x << (32 – d))
```

20. Операции отношения, логические операции

В C++ существуют операции отношиения и логические операции. В зависимости от операндов, к которым они применяются, операция возвращает значение true или false.

Операции отношения:

- == эквиваленция. Если операнды равны, возвращает true(1), иначе false(0)
- != неравенство. Если операнды не равны, возвращает true(1), иначе false(0)
- < меньше
- > больше
- <= меньше или равно
- >= больше или равно

Логические операции:

- && логическое И. Возвращает true(1), еслы все операнды истинны
- || логическое ИЛИ. Возвращает true(1), если хотя бы один из операндов истиннен
- ! унарное НЕ. Возвращает true(1), если исходная операция возвращала false(0) и наоборот, если исходная false(0), то возвращает true(1).

21. Операция присваивания

Операция присваивания выполняется над двумя операндами и сохраняет значение второго операнда в объект, указанный первым операндом.

Синтаксис: a = 10; b = a.

Если оба объекта имеют арифметические типы, правый операнд преобразуется в тип операнда слева перед выполнением операции.

Существуют составные операторы присваивания, сочитающие операцию присваивания с другими операциями:

- += сложение с присваиванием
- -= вычитание с присваиванием
- *= умножение с присваиванием
- /= деление с присваиванием
- <<= сдвиг влево с присваиванием
- >>= сдвиг вправо с присваиванием
- &= поразрядная конъюнкция с присваиванием
- |= поразрядная дизюнкция с присваиванием
- ^= поразрядное исколючающее ИЛИ с присваиванием

22. Условная трехместная (тернарная) операция

Тернарная операция — операция с тремя операндами. В C++ существует тернарный оператор. Он имеет следующий синтаксис:

выражение ? выражение : выражение

Тернарный оператор работает следующим образом:

- Первый операнд преобразуется в bool
- Если первый операнд является true (1), то выполняется второй операнд
- Если первый операнд является false (0), то выполняется третий операнд

Результатом тернарного оператора является второй или третий операнд.

Если типы второго и третьего операндов не одинаковы, то компилятором вызывается преобразование типов в соответствии со стандартом C++, что может привести к неверной работе программы. Лучше следить за типами вручную.

Следует помнить, что любое значение кроме единицы в переменной типа bool воспринимается как false.

Тернарные операторы можно вкладывать друг в друга, получая многоуровневые условные конструкции.

23. Безопасность преобразования типов

Тип данных — это характеристика значений, которые может принимать переменная или функция.

C++ - язык программирования со статической типизацией. Это значит, что переменная имеет только один изначально назначенный ей тип данных, и поменять мы его не можем. Но часто возникает необходимость присвоить переменной значение другого типа. Тогда используется преобразование типов. Это перевод значения из одного типа данных в другой.

Если присвоить переменной значение, выходящее за диапозон её типа данных, то компилятор произведёт неявное преобразование значения к соответствующему типу. В арифметических операциях все операнды должны быть одного типа, и компилятор приводит все значения к типу данных с наибольшим диапозоном, участвующим в операции. При этом велика вероятность потери информации, так что этого лучше избегать.

Существует два способа преобразованияя типов: круглые скобки и static_cast. Круглые скобки — унаследованный от C способ, static_cast появился уже в C++ и является более приоритетным, как более безопастный.

Пример:

```
int a = 6;
double b = (double)a;
double c = static_cast<double>(a);
```

Безопастное преобразование типов — при котором не происходит потеря информации. Опастное — при котором происходит потеря информации. Общее правило безопастного преобразования — от меньшего к большему, от целого к вещетсвенному.

Цепочки безопастного преобразования:

bool → char → short → int → double → long double
bool → char → short → int → long → long long
unsigned char → unsigned short → unsigned int → unsigned long
float → double → long double

24. Приоритет операций и порядок вычисления выражений.

В C++ существует приоритет выполнения операций. Если у всех операций в выражении одинаковый приоритет, то почти всегда они выполняются слева направо. Если в выражении операции с разным приоритетом — операции выполняются по убыванию приоритета.

Ссылка на таблицу: https://en.cppreference.com/w/cpp/language/operator_precedence
На русском: https://pvs-studio.ru/ru/blog/terms/0064/
Приоритет операций в C++:

Приоритет	Оператор	Описание
1	`::`	Разрешение области видимости
2	`a++ a--`	Суффиксный/постфиксный инкремент и декремент
	тип`()` *тип*`{}`	Функциональный оператор приведения типов
	`a()`	Вызов функции
	`a[]`	Индексация
	`. ->`	Доступ к элементу
3	`++a --a`	Префиксный инкремент и декремент
	`+a -a`	Унарные плюс и минус
	`! ~`	Логическое НЕ и побитовое НЕ
	(*тип*)	Приведение типов в стиле C
	`*a`	Косвенное обращение (разыменование)
	`&a`	Взятие адреса
	`sizeof`	Размер в байтах
	`co_await`	Выражение await (C++20)
	`new new[]`	Динамическое распределение памяти
	`delete delete[]`	Динамическое освобождение памяти
4	`.* ->*`	Указатель на элемент
5	`a*b a/b a%b`	Умножение, деление и остаток от деления
6	`a+b a-b`	Сложение и вычитание
7	`<< >>`	Побитовый сдвиг влево и сдвиг вправо
8	`<=>`	Оператор трёхстороннего сравнения (C++20)
9	`< <= > >=`	Для операторов отношения < и ≤ и > и ≥ соответственно
10	`== !=`	Операторы равенства = и ≠ соответственно
11	`&`	Побитовое И
12	`^`	Побитовый XOR (исключающее или)
13	`|`	Побитовое ИЛИ (включающее или)
14	`&&`	Логическое И
15	`||`	Логическое ИЛИ
16	`a?b:c`	Тернарный условный оператор
	`throw`	Оператор throw
	`co_yield`	Выражение yield (C++20)
	`=`	Прямое присваивание
	`+= -=`	Присваивание с сложением и вычитанием
	`*= /= %=`	Присваивание с умножением, делением и остатком
	`<<= >>=`	Присваивание с побитовым сдвигом влево и вправо
	`&= ^= |=`	Присваивание с побитовым И, XOR и ИЛИ
17	`,`	Запятая

25. Функции форматированного вывода printf и ввода информации scanf

Поток — непрерывный процесс, который обрпбатывает и выполняет команды. Поток ввода-вывода нужен для получения данных (из файла, от пользователя) и для вывода данных(например, в консоль). При обмене с потоком используется вспомогательный участок памяти — буфер потока. Для работы с потоками ввода и вывода в С необходимо подключить заголовочный файл stdio.h.

Для вывода информации используется функция printf(). Общая форма записи:

printf("СтрокаФорматов", объект1, объект2, …, объектn);

Строка формата состоит из следующих элементов:

- управляющие символы
- текст, предназначенный для непосредственного вывода
- форматы, предназначенные для вывода значений переменных

Управляющие символы не выводятся на экран, а управляют расположением выводимых символов. Отличительной чертой управляющего символа является наличие обратного слэша '\' перед ним. Основные - /n — перевод строки и /t — горизонтальная табуляция.

Форматы нужны для того, чтобы указывать вид, в котором информация будет выведена на экран. Отличительной чертой формата является наличие символа процент ‘%’ перед ним:

- %d — int
- %u — unsigned int
- %x — int в шестнадцатиричной
- %f — float
- %lf — double
- %c — char
- %s — строка
- %p — указатель

Пример:

```
int x = 5;
printf("У меня %d яблок", x);
```

Для форматированного ввода информации используется функция scanf(). Общая форма записи:

scanf ("СтрокаФорматов", адрес1, адрес2,...);

В данной функции работаем не переменную, а её адрес!

Строка форматов аналогична функции printf().

Пример:

```
scanf("%f", &y);
```

Функция scanf() является незащищённой, так что для её работы в современных компиляторах необходимо разрешить её использование, добавив в программу строчку
#define _CRT_SECURE_NO_WARNINGS

Существует защищённый вариант — scanf_s().

26. Методы форматированного вывода cout и ввода информации cin

Поток — непрерывный процесс, который обрпбатывает и выполняет команды. Поток ввода-вывода нужен для получения данных (из файла, от пользователя) и для вывода данных(например, в консоль). При обмене с потоком используется вспомогательный участок памяти — буфер потока. Для работы с потоками ввода и вывода в C++ необходимо подключить библиотеку iostream.

Основными функциями для работы с потоками ввода-вывода являются cin и cout. Они находятся в пространстве имён std, поэтому их вызов записывается следующим образом:

```
std::cin » a;
std::cout « a « b;
```

Для операций ввода-вывода переопределены операции поразрядного сдвига:

- >>получить из входного потока
- <<поместить в выходной поток

Для работы с широкими символами (wchar_t) существуют аналоги: wcout и wcin.

Существуют функции-манипуляторы. Функцию — манипулятор потока можно включать в операции помещения в поток и извлечения из потока (<<, >>).
В C++ имеется ряд манипуляторов. Рассмотрим основные:

- endl — очищает буфер потока и переходит на новую строку
- width — устанавливает ширину поля вывода
- precision — устанавливает количество цифр после запятой

У потока ввода также есть функции:
- cin.peek — возвращает следующий символ без извлечения
- cin.ignore(число, символ) — удаляет из потока ввода *число* символов, пока не встретит *символ*.

27. Понятие оператора. Оператор простой и составной, блок оператора

Оператор — это команда, обозначающая определенное математическое или логическое действие, выполняемое с данными (операндами).

В языке С существует следующие группы операторов:

- Условные операторы (if, switch)
- Операторы цикла (while, for, do-while)
- Операторы безусловного перехода (break,continue,goto,return)
- Метки (case,default)
- Операторы-выражения – операторы, состоящие из допустимых выражений.

Блоки – фрагмент текста программы, оформленный фигурными скобками {}. Блок иногда называют составным оператором.

Простым оператором является такой оператор, который не содержит в себе других операторов.

Составной оператор – представляет собой два или более операторов, объединенных с помощью фигурных скобок. Составной оператор иногда называют блоком. Составной оператор рассматривается компилятором как один оператор.

На языке Си любой оператор заканчивается точкой с запятой. Операторы можно условно подразделить на две категории: исполняемые - с их помощью реализуется алгоритм орешаемой задачи, и описательные, необходимые для определения типов пользователя и объявления объектов программы, например, переменных.

Исполняемые операторы также можно разбить на две группы: простые и структурированные. В структурированных операторах можно выделить части, которые сами могут выступать в качестве отдельных операторов, а простые операторы на более элементарные разложить не удастся.

К простым операторам относятся: оператор присваивания, оператор-выражение, пустой оператор, операторы перехода (goto, continue, break, return), вызов функции как отдельного оператора.

Структурированные операторы – это операторы ветвления (if), выбора (switch), цикла (for, while, do).

28. Виды управляющих конструкций программы

Программы не ограничиваются линейной последовательностью выполнения команд. Во время выполнения программа может повторять сегменты кода или разветвляться в зависимости от некоторого условия. Для реализации этого существуют управляющие конструкции.

В C++ несколько видов управляющих конструкций:

- Ветвление
- Циклы
- break
- continue
- goto

Добавить общее описание каждой конструкции, привести примеры.

29. Операторы ветвления, условный оператор

Для реализации ветвления в C++ существует условный оператор if else. Он имеет следующий синтаксис:

```
if(условие){
        код
} if else(условие) {
        код
} else {
        код
}
```

Оператор if проверяет условие в скобках на истинность. Если оно истинно — код дальше выполняется, если ложно — проверяет следующий if else. Если if и все if else ложны, выполняется блок кода else. Блоки if else и блок else являются необязательными.

30. Метки и переходы. Оператор выбора (switch-case)

В дополнение к стандартным методам работы с ветвлением в C++ существует оператор **goto**. Он позволяет перейти в конкретное место в коде, заданное **меткой —** идентификатором, состоящим из названия и двоеточия после него. В функции не может быть двух меток с одинаковыми названиями.
Пример:

```
int x = 9;
Loop1:
if(x < 20){
        ++x;
        goto Loop1;
};
```

Switch-case — конструкция для ветвления, позволяющая выбирать между несколькими разделами кода в зависимости от значения **целочисленного** выражения. Она также использует метод переходов по меткам.
Switch-case имеет следующий синтаксис:

```
switch(инициализация, выражение){
        case значение 1:
                //код
                break;
        case значение 2:
                //код
                break;
        default:
                //код
}
```

case и **default** — **метки**. Если выражение в объявлении совпадает со значением **case**, то выполнение кода переходит на эту метку и продолжается в обычном режиме. Для того, чтобы другие случаи case не выполнились, необходимо добавить оператор прерывания **break**. Случай **default** выполняется, если выражение в объявлении не соответствует ни одному case.

Инициализация переменной в объявлении необязательна, при её отсутствии нет необходимости оставлять пустое место и ставить запятую.
Хорошим тоном считается каждый блок case заключать в свою область видимости {}(как в циклах).

В целом использование **goto** и **swicth-case** не рекомендуется. Переход в произвольное место кода нарушает ход следования чтения программы сверху-вниз. С помощью метки возможен переход прямо в центр другого блока кода, цикла, ветвления, что также плохо для понимания. Возникают трудноотслеживаемые ошибки. Компилятору труднее (а иногда — невозможно) оптимизировать такой код.

Однако в некоторых случаях **goto** является хорошим инструментом. Один из примеров — выход из глубоко вложенного цикла.

31. Понятие цикла. Операторы цикла: цикл с заданным числом повторений

Цикл — управляющая конструкция, которая заставляет блок кода повторятся несколько раз. В программировании используются повсеместно для повторения каких-либо инструкций, например — проход по массиву и изменение каждого элемента. В C++ существуют циклы с заданным числом повторений (for), циклы с предусловием (while), и циклы с постусловием (do while).

Цикл с заданным числом повторений в C++ это цикл **for**. Синтаксис цикла **for** при определении имеет следующую форму:

```
for(инициализатор; условие; итерация ){
        //очень важный код;
}
```

Все параметры являются **обязательными** при определении. Поле параметра при надобности можно оставить пустым, но точка с запятой необходимы.
Инициализатор — обычно используется для инициализации итерируемой переменной
Пример:

```
for(int i = 0; ; ){}
```

Условие — условие выхода из цикла. Зачастую в условии используется итерируемая переменная, но это не обязательно. Если возвращает true — цикл продолжается, если false — цикл прерывается.
Пример:

```
for(int i = 0; i < 10; ){}
```

Итерация — действие, которое производится после выполнения одного повторения цикла. Зачастую это изменение итерируемой переменной на некоторый шаг.
Пример:

```
for(int i = 0; i < 10; ++i){
        //очень важный код;
}
```

Последний цикл повторит очень важный код 10 раз. В первой итерации i = 0, в последней итерации i = 9. При проверке на следующее повторение цикла i станет равна 10, и при проверке условия i < 10 будет неверным, что не позволит циклу повториться и программа пойдёт обрабатывать команды дальше.

32. Понятие цикла. Операторы цикла: цикл с предусловием и с постусловием

Цикл — управляющая конструкция, которая заставляет блок кода повторятся несколько раз. В программировании используются повсеместно для повторения каких-либо инструкций, например — проход по массиву и изменение каждого элемента. В C++ существуют циклы с заданным числом повторений (for), циклы с предусловием (while), и циклы с постусловием (do while).

Цикл с предусловием перед выполнением блока кода проверяет на истинность условие. Если оно ложно, то блок не выполняется. Если истинно — то блок кода выполняется и условие проверяется заново с последующим повторением (или нет) выполнением кода. Таким образом образуется цикл с предусловием. В C++ цикл с предусловием имеет следующий синтаксис:

```
while(условие){
	// код
}
```

Для выхода из такого рода циклов часто кроме условия используется оператор прерывания цикла **break**, который позволяет прямо в цикле прервать его выполнение. Обычно используется в связке с условным оператором **if**.

Цикл с постусловием отличается от цикла с предусловием тем, что блок кода **в любом случае** выполняется 1 раз и после этого при истинности условия выполняется ещё раз. В C++ имеет следующий синтаксис:

```
do {
	// код
} while (условие)
```

Данные виды циклов повсеместно используются в написании программ. Такие циклы используются, когда неизвестно необходимое количество повторений кода для получения нужного разельтата. Часто цикл с предусловием используется для подсчёта инераций. Для этого заводится переменная-счётчик.
Пример:

```
int number = 100;
int count = 0;
while(number > 10){
	number = number / 2;
	++count;
}
std::cout « count; // сколько раз нужно число number разделить на 2, чтобы оно стало меньше 10
```

Также стоит упомянуть схему написания всегда истинных циклов с условием.
Пример:

```
while (true){}
```

Для выхода из таких циклов используется оператор прерывания **break**.

33. Понятие цикла. Бесконечные циклы. Проблемы

Цикл — управляющая конструкция, которая заставляет блок кода повторятся несколько раз. В программировании используются повсеместно для повторения каких-либо инструкций, например — проход по массиву и изменение каждого элемента. В C++ существуют циклы с

заданным числом повторений (for), циклы с предусловием (while), и циклы с постусловием (do while).

Бесконечный цикл — цикл, условие выхода из которого никогда не выполняется. Если программа захоидт в бесконечный цикл, зачастую это заканчивается ошибкой, чаще всего — переполнением стэка (stack overflow).

При создании программы важно следить, чтобы используемые циклы никогда в условиях программы не могли превратиться в бесконечные.

Примеры бесконечных циклов:

```
int i = 10
while(i>5){
	std::cout « «Этот цикл бесконечен» « std::endl;
}

for(int i = 0; 1; ++i){
	std::cout « i « std::endl;
}
```

34. Понятие цикла. Операторы прерывания и продолжения цикла

Цикл — управляющая конструкция, которая заставляет блок кода повторятся несколько раз. В программировании используются повсеместно для повторения каких-либо инструкций, например — проход по массиву и изменение каждого элемента. В C++ существуют циклы с заданным числом повторений (for), циклы с предусловием (while), и циклы с постусловием (do while).

Операторы перехода:
Оператор **break** завершает выполнение ближайшего внешнего цикла или оператора switch.
Оператор **continue** начинает новую итерацию ближайшего внешнего цикла.

Оператор **break** завершает выполнение только одного цикла. Для выхода из нескольких вложенных циклов искользуются другие методы.

35. Одномерные и многомерные массивы статические массивы

Массив — это определённое число ячеек памяти, расположенных подряд. Они позволяют эффективно хранить однотипные данные. Имя массива является указателем на его первый элемент. Все элементы массива имеют одинаковый тип данных.

По способу хранения в памяти массивы различаются на два вида: статические и динамические.

Статические массивы хранятся в статической памяти. Объявляются в глобальной области видимости. Их размер определяется на этапе компиляции и не может быть изменён. Объявление статического массива:

тип элемента[размер массива]

```
int a[16]
int array[5] = { 0, -9, 23, 61, -15 };
```

Динамический массив хранится в динамической памяти(куче). Скорость обращения к нему немного меньше, но его размер мы можем определять произвольно и во время выполнения программы. Для этого нужно выделить место в куче и создать указатель на это место. После отработки массива хорошим тоном считается очистка занимаемой им памяти. Пример объявления динамического массива:

```
int *numbers = new int[4];
```

Массивам, объявленым в локально, выделяется память в стеке, нужно учитывать её ограниченность.

Массивы бывают одномерные и многомерные.

Одномерный массив — простая последовательность элементов.

Многомерные массивы — массивы указателей. Как частный пример многомерного массива можно привести матрицу — двухмерный массив.

Доступ к элементам массива осуществляется через индексатор:

```
int a[4];
a[0] = 42;
int t = a[3];
```

Выход за границы массива не контролируется, ошибка может привести к неопределённому поведению.

36. Указатели. Связь между указателями и динамическими массивами

Указатель – переменная, значением которой является адрес ячейки памяти. Может ссылаться на переменную или функцию. Для взятия адреса существует унарная операция &

Указатели используются для передачи по ссылке данных, что намного ускоряет процесс обработки этих данных (в том случае, если объём данных большой), так как их не надо копировать, как при передаче по значению, то есть, используя имя переменной.

Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки *

```
int a = 5;
int *p = &a;  // p — указатель на a
```

Так как указатель хранит адрес, то мы можем по этому адресу получить хранящееся там значение, то есть значение переменной x. Для этого применяется операция * или операция разыменования, то есть та операция, которая применяется при определении указателя. Результатом этой операции всегда является объект, на который указывает указатель.

Размер указателя равен разрядности системы

Указатели используются для организации динамических массивов. Имя массива — указатель на его первый элемент. Для создания динамического массива необходимо выделить память в куче. Чтобы выделенное место не потерять, адрес на её начало сохраняется в указатель. Для работы с динамическими массивами указатели испульзуются постоянно. Двумерный массив — массив указателей на массивы. Объявление двумерного динамического массива и очистка памяти:

```
int nstr = 5; // количество строк
int nstb = 5; // количество столбцов
// объявление двумерного динамического массива на 25 элементов
int** a = new int*[nstr]; // пять строк в массиве
for (int i = 0; i < nstr; ++i) {
   a[i] = new int[nstb]; // и пять стобцов
}

for (int i = 0; i < nstr; ++i) {
   delete[] a[i]; // удаляем строку
   a[i] = nullptr; // убираем висячий указатель
}
delete[] a; // удаляем сам двумерный массив
a = nullptr; // убираем висячий указатель
```

Если указатель указывает на массив, то его можно индексировать, как массив.

37. Операции над указателями разного порядка

Над указателями можно проводить 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 указыввают 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.

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

[2]|

```
/// 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];
    }
  }
}
```

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”;
   }
   ```

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])
   ```

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

```
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';
// -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);
```

[4] implementation-defined

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 сортировки. 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, блягодаря 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

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

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 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 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 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 объявленные внутри структуры 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.

Элементами структуры являются поля и методы структуры. Доступ к элементам структуры можно получать через объект структуры с помощью оператора `.` или через указатель на объект структуры через оператор `->`. Пример в предыдущем вопросе.

Инициализация

Инициализация струтруры в языках C и C++ частично 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;
```

[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.)

```
  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 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 загружают и выгружают данные, когда они **выравнены**, то есть их адрес кратен какой-либо степени числа 2. Это число называется **выравниванием** и зависит от типа данных.

Все простые типы (кроме `long double`, он выравнен по 16 байтам) 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 (расстоянию в байтах от начала структуры до начала самого поля) `member` структуры `type`.

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

С целью выравнивания при создании структур компилятор может добавлять неиспользуемые байты — **паддинги** между двумя полями, если у предыдущего выравнивание меньше, чем следующего. Также паддинги могут добавляться и в конце структуры, вероятно, чтобы обеспечить корректное выравнивание для последующих структур в массиве.

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

Использование директивы `#pragma pack` применяет упаковку ко *всем* структурам, которые объявлены после нее[7]. Это относится и к тем структурам, которые могут быть объявлены в других заголовочных файлах. Если при компиляции данного файла компилятор будет думать, что структура упакована, но другой код, с которым линкуется этот файл, компилировался без учета упаковки, то бинарное представление структур в памяти (ABI) окажется несовместимым и программа, вероятно, упадет.

Чтобы избежать этого, упаковки структур следует оборачивать в директивы `pack(push)` и `pack(pop)`, как в примере ниже.

```
#include <cstddef>
#include <iostream>

struct S {
  char   m0; // 1 байт
  // <паддинг 7 байт>
  double m1;
  short  m2;
  char   m3;
  // <паддинг 5 байт>
};

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

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
```

[7]по крайней мере, в GCC и Clang

```
    << ''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

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;
};

// Валидно
struct RefNode {
  int value;
  Node &next;
};

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

// Двусвязный список
struct List {
  PtrNode head;
  PtrNode tail;
};
```

Выравнивание структуры в принципе и вложенной структуры в частности равно самому большому выравниванию среди ее (вложенной структуры) полей.

```
struct A {
  int x;
  char c;
  double y;
};
```

```
struct B {
  char c;
  A a;
};

std::cout
  // 16 8
  << sizeof(A) << ' ' << alignof(A) << '\n'
  // 24 8
  << sizeof(B) << ' ' << alignof(B) << '\n'
  // 8
  << offsetof(B, a) << '\n';
```

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 можно использовать оператор стрелочка (`->`): (`ptr->field`); или садомазохистскую запись с разыменованием структуры и доступом к полю через объект структуры с помощью оператора точка (`.`): `(*ptr).field`.

```
struct Person {
  int age;
  std::string name;
};

Person *jack = new Person {27, ''Jack''};

Person *peter = new Person;
peter->name = ''Peter'';
(*peter).age = 17;

// Jack is 27
std::cout << jack->name << ''is '' << jack->age << '\n';
// Peter is 17
std::cout << peter->name << ''is '' << peter->age << '\n';
```

```
delete peter;
delete jack;
```

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 объединения;

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 одного из типов, которые есть в объединении. При попытке перезаписать данные другим типом новые данные записываются поверх старых, а для старых данных деструктор не вызывается. Поэтому в `union` без дополнительных плясок с бубном нельзя поместить «умный» производный 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
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 {
  <bool | 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 ($31 = 2^5 - 1$) 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';

// ============================================================
// Битовые поля позволяют компактно хранить булевых значений
struct BitSet {
  bool b1 : 1;
  bool b2 : 1;
  bool b3 : 1;
  bool b4 : 1;
  bool b5 : 1;
  bool b6 : 1;
  bool b7 : 1;
  bool b8 : 1;
};

BitSet Compress(bool b[8]) {
  BitSet res;
  res.b1 = b[0];
  res.b2 = b[1];
```

```
  res.b3 = b[2];
  res.b4 = b[3];
  res.b5 = b[4];
  res.b6 = b[5];
  res.b7 = b[6];
  res.b8 = b[7];
  return res;
}

int main() {
  bool bool_array[8] = {true, true, true, false,
                        false, true, false, true};
  BitSet bitset = Compress(bool_array);
  std::cout
    << bool_array[0] << ' ' << bitset.b1 << '\n'  // 1 1
    << bool_array[1] << ' ' << bitset.b2 << '\n'  // 1 1
    << bool_array[2] << ' ' << bitset.b3 << '\n'  // 1 1
    << bool_array[3] << ' ' << bitset.b4 << '\n'  // 0 0
    << bool_array[4] << ' ' << bitset.b5 << '\n'  // 0 0
    << bool_array[5] << ' ' << bitset.b6 << '\n'  // 1 1
    << bool_array[6] << ' ' << bitset.b7 << '\n'  // 0 0
    << bool_array[7] << ' ' << bitset.b8 << '\n'; // 1 1
}
```

Время хранения (storage duration). Связывание

Эта информация в равной мере относится к последующим трем вопросам. Поэтому я решил ее вынести в отдельный раздел.

Время хранения — это свойство объекта, которое определяет минимальное возможное время жизни хранилища, содержащего объект[8]. 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’’).

Будем говорить, что переменная (символ) имеет **внутреннее** связывание, если он доступен только из той единицы трансляции, в которой объявлен.

Будем говорить, что переменная (символ) имеет **внешнее** связывание, если доступ к нему можно получить из любой единицы трансляции.

Стандарт также выделяет переменные **без связывания** — все переменные внутри функций (блоков), которые явно не объявлены `static` или `extern`.

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

49. Локальные и глобальные переменные

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

```
// Два файла компилировать вместе
// В файле lib.cc
int a = 42;

// В файле main.cc
#include <iostream>

void Func() {
  // статическое время хранения (внешнее связывание)
  // сейчас эта переменная локальная
  // но если эту же декларацию вынести за пределы
  // функции, то эта переменная станет глобальной
  extern int a;
  std::cout << ”a = ” << a << ’\n’;
}

void Counter() {
  // статическое время хранения (внутреннее связывание)
  // чисто локальная переменная
  static int count = 0;
  std::cout << ”count = ” << ++count << ’\n’;
}

void PrintHi() {
  // автоматическое время хранения
  std::string name;
  std::cin >> name;
  std::cout << ”Hello, ” << name << ”!\n”;
}

int main() {
  // 42
  Func();
  // 42
  Func();

  // Ошибка компиляции
  // a = 24;

  // 1
  Counter();
  // 2
  Counter();

  // Ошибка компиляции
  // std::cin >> count;
  PrintHi();
```

```
  // Ошибка компиляции
  // name = ''Doomguy'';

  //3
  Counter();

  {
    int x = -3;
    std::cout << x << '\n';
  }
  // Ошибка компиляции
  // x = 3;
}
```

Глобальные переменные объявляются вне тела функции и доступны из любых функций. Глобальные пременные имеют статическое или потоковое время хранения и могут иметь как внешнее, так и внутренне связывание. Все глобальные переменные хранятся в статической области памяти.

Поскольку глобальные переменные доступны из любой функции, их значение может поменять кто угодно. Это может нарушить внутренние взаимосвязи в программе, из-за чего их использование (особенно с внешним связыванием) не рекомендуется.

```
// Глобальная переменная
std::string name;

void ReadName() {
  std::cout << ''Введите ваше имя: '';
  std::cin >> name;
}

void Welcome() {
  std::cout << ''Добро пожаловать, '' << name << ''!\n'';
}

int main() {
  ReadName();
  Welcome();
}
```

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.

```
std::string ReadName() {
  // автоматическая переменная
  std::string name;
  std::cout << ”Введите ваше имя: ”;
  std::cin >> name;
  return name;
}

void Welcome(/* автоматическая переменная */
             std::string name1)
{
  std::cout << ”Добро пожаловать, ” << name1 << ”!\n”;
} // <- Здесь переменная name1 уничтожена

int main() {
  // автоматичекая локальная переменная
  std::string name = ReadName();
  Welcome(name);
} // <- Здесь переменная name уничтожена
```

51. Внешние и статические переменные, особенности их реализации

Термин ‘статическая переменная’ неоднозначен: он может обозначать как переменную, которая находится в статической области памяти (любая не-`thread_local` глобальная переменная или локальная переменная с внутренним или внешним связыванием), так и переменную с внутренним связыванием (глобальная или локальная переменная, объявленная `static`). **Статической переменной** будем называть переменную, которая располагается в статической облати памяти.

Переменные как с внешним, так и с внутренним связыванием хранятся в статической области памяти, то есть являются статическими в смысле данного выше определения.

В отличие от кучи и стека, размер статической памяти постоянен и не может меняться во время исполнения.

Переменные с внутренним связыванием могут быть как глобальными, так и локальными. Про глобальные переменные разговор будет ниже, а пока остановимся на локальных. Однако локальные переменные с внутренним связыванием хранятся не на стеке, а в статической памяти, потому они не очищаются при выходе из функции. Это позволяет сохранять состояние между вызовами функции. Статические локальные переменные инициализируются только один раз: тогда, когда строка с такой переменной впервые выполняется. Когда статическая локальная переменная при выполнении программы встречаеся в следующий раз, она не инициализируется повторно.

```
#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, то есть доступна из других единиц трансляции. Такими переменными являются глобальные переменные, определенные (defined) без спецификатора `static` или с спецификатором `extern`.

Хотя нормальные компиляторы (GCC, Clang, MSVC?) поддерживают определение переменной с ключевым словом `extern` (см. переменную `c` ниже), оно предназначено только 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;
// Переменная с внутренним связыванием
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++/<ве`

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[9], 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 `идентификатор`а.

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';
```

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

```
#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**.