

Приемы программирования на языке C

Содержание

1	Ресурсы по языку Си	2
2	Вводные замечания	2
3	Установка MSYS2 и MinGW-W64 для ОС Windows	2
4	Приемы работы в редакторе Eclipse	3
4.1	Настройка редактора Eclipse	3
4.2	Сборка и запуск проекта	3
4.3	Компиляция и запуск программы в редакторе Eclipse	5
5	Visual Studio Code как среда разработки для языка Си	6
6	Алфавит, идентификаторы, служебные слова	6
6.1	Константы и строки	6
6.2	Переменные и именованные константы	7
7	Структура программы	8
8	Ввод / вывод	10
8.1	Вывод данных	10
8.2	Ввод данных	12
8.2.1	Получение данных из командной строки	13
8.2.2	Преждевременное завершение программы	14
9	Переменные и типы данных	14
9.1	Спецификаторы хранения	16
9.2	Массивы	17
9.2.1	Строка	18
9.2.2	Указатели	18
9.2.3	Динамическое выделение памяти	21
9.2.4	Структуры	24
9.2.5	Битовые поля	25
9.2.6	Объединения	26
9.2.7	Перечисления	26
	Список литературы	27
	Список листингов	27

1. Ресурсы по языку Си

<https://learn.c.info/c/>

2. Вводные замечания

Язык Си – это компилируемый язык программирования высокого уровня, кроссплатформенный, позволяющий создавать программы которые будут работать во всех операционных системах, но для каждой операционной системы компиляцию нужно выполнять отдельно.

Существует несколько стандартов языка Си: C90 (ANSI C/ISO C), C99 и C11. Для того чтобы использовать правила конкретного стандарта, нужно в составе команды компиляции указать следующие флаги: `-std=c90`, `-std=c99` или `-std=c11`. Современный язык Си включает возможности стандарта C11.

Узнать используемый стандарт языка Си внутри программы можно с помощью *макроса* `__STDC_VERSION__`

```
printf("%ld\n", __STDC_VERSION__); // 201112
```

Получить информацию о версии компилятора позволяет макрос `__VERSION__`

```
printf("%s\n", __VERSION__); // Apple LLVM 12.0.0 (clang-1200.0.32.2)
```

Когда мы в командной строке вводим название программы без предварительного указания пути к ней, то

- вначале поиск программы выполняется в текущем рабочем каталоге (обычно это каталог, из которого запускается программа),
- а затем в путях, указанных в системной переменной `PATH`.

Системные каталоги имеют более высокий приоритет, чем каталоги, указанные в переменной `PATH`.

3. Установка MSYS2 и MinGW-W64 для ОС Windows

Установить компилятор `gcc` на ОС Windows можно следующим образом. Детали процедуры установки можно найти в книге [2, стр. 24]. Предварительно нам нужно установить библиотеку MSYS2. Переходим на сайт <https://www.msys2.org> и скачиваем файл `msys2-x86_64-20230718.exe`, а затем запускаем его.

Библиотека MSYS2 будет установлена в каталог `C:\msys64`. В этом каталоге расположены скрипты для запуска: `msys2.exe`, `mingw32.exe`, `mingw64.exe`.

Файл `msys2.exe` запускает командную строку, в которой мы можем установить различные библиотеки. Сначала обновим программу, выполнив команду



Окно `msys2.exe`

```
$ pacman -Syu
```

Теперь можно установить библиотеку MinGW-W64

```
$ pacman -S mingw-w64-x86_64-toolchain
```

Для установки всех компонентов нажимаем клавишу <Enter>, а затем на запрос подтверждения установки вводим букву Y и нажимаем клавишу <Enter>.

Библиотека MinGW-W64 будет установлена в каталог  C:\msys64\mingw64. Добавив путь до  C:\msys64\mingw64\bin в системную переменную Path, можно будет вызывать компилятор gcc из любой точки

```
$ gcc --version
g++.exe (Rev2, Built by MSYS2 project) 13.2.0
...
```

Все установленные библиотеки скомпилированы под 64-битные операционные системы. Для установки 32-битных версий библиотек нужно в команде заменить фрагмент x86_64 фрагментом i686. Пример

```
$ pacman -S mingw-w64-i686-toolchain
```


4. Приемы работы в редакторе Eclipse

4.1. Настройка редактора Eclipse


Чтобы сделать иконки панели покрупнее, следует добавить в файл eclipse.ini следующие строки

```
$HOME/eclipse/cpp-2023-06/eclipse/eclipse.ini
```

```
-Dswt.enable.autoScale=true
-Dswt.autoScale=150
-Dswt.autoScale.method=nearest
```

Чтобы редактор поддерживал Vim, следует в меню  Eclipse Markertplace в строке Find вбить «Vgarper» и затем следовать инструкциям по установке.

4.2. Сборка и запуск проекта

Для того чтобы преобразовать текстовый файл Test64c.c с программой в исполняемый exe-файл, делаем текущей вкладку с содержимым файла Test64c.c и в меню Project выбираем пункт Build Project. В результате компиляции в рабочем каталоге будет создан каталог  Debug. Внутри этого каталога находится файл Test64c.exe, который можно запустить на выполнение с помощью двойного щелчка мыши на значке файла.

Для запуска делаем текущей вкладку с содержимым файла Test64c.c и в меню Run выбираем пункт Run. В открывшемся окне выбираем пункт Local C/C++ Application и нажимаем кнопку ОК. Результат выполнения программы отобразится в окне Console.

Простейший пример программы

```
#include <stdio.h>

int main(void) {
    printf("Hello, world");

    // в main() ключевое слово return можно не указывать
    return 0;
}
```

Здесь `#include` – это *директива препроцессора*, с помощью которой включается файл `stdio.h`, в котором есть функция `printf()`, предназначенная для форматированного вывода данных в окно консоли. Так как название файла указано внутри угловых скобок, его поиск будет выполнен в *путях поиска заголовочных файлов*.

Содержимое файла `stdio.h` на одной из стадий компиляции целиком вставляется вместо инструкции с директивой `#include`.

Функция `printf()` содержится внутри файла `stdio.h`, поэтому в первой строке программы мы включаем этот файл с помощью директивы `#include`. Если заголовочный файл не включить, то функция будет недоступна.

После всех *инструкций* указывается точка с запятой. Исключением являются [2, стр. 46]:

- *составные инструкции* (в нашем примере после закрывающей фигурной скобки блока функции `main()` точка с запятой не указывается)
- и *директивы препроцессоров* (в нашем примере нет точки с запятой после инструкции с директивой `#include`).

Согласно стандарту, внутри функции `main()` ключевое слово `return` можно не указывать. В этом случае компилятор должен самостоятельно вставить инструкцию, возвращающую значение 0 [2, стр. 46].

Программу можно скомпилировать и без редактора кода. Пример компиляции на ОС Windows

```
gcc -Wall -Wconversion -O3 -finput-charset=cp1251 -fexec-charset=cp1251 -o helloworld.exe helloworld.c
```

Первое слово (`gcc`) вызывает компилятор `gcc.exe`. Флаг `-Wall` указывает выводить все предупреждающие сообщения, возникающие во время компиляции программы, флаг `-Wconversion` задает вывод предупреждений при возможной потере данных, а флаг `-O3` определяет уровень оптимизации. С помощью флага `-finput-charset` указывается кодировка файла с программой, а с помощью флага `-fexec-charset` – кодировка C-строк. Название создаваемого в результате компиляции файла (`helloworld.exe`) задается после флага `-o`. Далее указывается название исходного текстового файла с программой на языке Си (`helloworld.c`).

Помимо файлов с исходным кодом (имеют расширение `*.c`) в проекте могут быть *заголовочные файлы* (имеют расширение `*.h`).

В заголовочных файлах указываются *прототипы функций* и *различные объявления*. Инструкции, начинающиеся с символа `#`, – это *директивы препроцессора* [2, стр. 47].

Например для заголовочного файла с именем `HelloWorld.h`

```
#ifndef HELLOWORLD_H_
#define HELLOWORLD_H_
#endif /* HELLOWORLD_H_ */
```

Здесь директива препроцессора `#ifndef` проверяет отсутствие константы с именем `HELLOWORLD_H_`, `#define` – создает константу с именем `HELLOWORLD_H_`, а `#endif` – обозначает конец блока проверки отсутствия константы.

Заголовочный файл мы подключаем к файлу с исходным кодом (`*.c`) с помощью директивы `#include`: `#include "HelloWorld.h"` (кавычки!!! а не угловые скобки). Встретив в исходном коде директиву `#include`, *компилятор* вставляет все содержимое заголовочного файла на место директивы. Если мы вставим две одинаковые директивы `#include`, то содержимое заголовочного файла будет вставлено дважды. Чтобы этого избежать прототипы функций и прочие объявления вкладываются в блок, ограниченный директивами `#ifndef` и `#endif`. В директиве `#ifndef`

указывается константа, совпадающая с именем заголовочного файла. Все буквы в имени константы заглавные, а точка заменена символом подчеркивания. Если константа не существует (при первом включении так и будет), то с помощью директивы `#define` эта константа создается и содержимое блока вставляется в исходный код. При повторном включении заголовочного файла константа уже существует, поэтому содержимое блока будет проигнорировано. Таким образом, заголовочный файл вставлен не будет, а значит, и ошибки не возникает.

Вместо этих директив в самом начале заголовочного файла можно указать директиву препроцессора `#pragma` со значением `once`, которая также препятствует повторному включению файла (в старых компиляторах директива может не поддерживаться)

```
#pragma once
// Объявление функций и пр.
```

Название заголовочного файла в директиве `#include` может быть указано [2, стр. 51]:

- внутри угловых скобок `#include <stdio.h>`,
- внутри кавычек `#include "HelloWorld.h"`.

В первом случае заголовочный файл ищется в путях поиска заголовочных файлов. При этом *текущий рабочий каталог* не просматривается. Добавить каталог в пути поиска заголовочных файлов позволяет флаг `-I` в команде компиляции. Обычно с помощью *угловых скобок* включаются заголовочные файлы *стандартной библиотеки* или библиотеки *стороннего разработчика*.

Во втором случае мы имеем дело с заголовочным файлом, который *вначале* ищется в *текущем рабочем каталоге* (или относительно него), а *затем в путях поиска заголовочных файлов*, как будто название указано внутри угловых скобок. Таким способом (`#include "HelloWorld.h"`) обычно включаются *заголовочные файлы проекта*.

Можно указать:

- просто название заголовочного файла

```
#include "HelloWorld.h"
```

- абсолютный путь к нему

```
#include "C:\\cpp\\projects\\HelloWorld\\src\\HelloWorld.h"
```

- или относительный путь к нему

```
#include "../HelloWorld.h"
```

4.3. Компиляция и запуск программы в редакторе Eclipse

Компиляция в редакторе Eclipse выполняется в два прохода. При первом проходе создается объектный файл `HelloWorld.o`, а на втором проходе на его основе создается исполняемый файл.

По умолчанию для проекта задается *режим компиляции Debug*. В этом режиме дополнительно сохраняется информация для отладчика, и EXE-файл будет создан *без оптимизаций*. Когда программа уже написана и отлажена, нужно выбрать режим `Release`. Для этого в меню `Project` выбираем пункт `Build Configuration >> Set Active >> Release`.

В результате компиляции в разных режимах были созданы два EXE-файла – в подкаталоге `Debug` и в подкаталоге `Release`. Первый файл содержит отладочную информацию, а второй – нет. При компиляции второго была дополнительно выполнена оптимизация, поэтому именно этот файл нужно отдавать заказчику.

5. Visual Studio Code как среда разработки для языка Си

Скачать Visual Studio Code можно здесь <https://code.visualstudio.com/>. Для ОС Windows нужно еще установить GCC. На ОС Linux компилятор gcc доступен «из коробки». На ОС MacOS компилятор gcc можно установить с помощью утилиты **brew**.

После установки IDE останется только создать директорию с проектом под язык Си. Когда Visual Studio Code увидит файл с расширением *.c, она предложит установить специальное расширение «C/C++ Extension Pack v1.X.X».

6. Алфавит, идентификаторы, служебные слова

Идентификаторы, начинающиеся с одного символа подчеркивания «_» или с двух символов подчеркивания «__», зарезервированы для использования в библиотеках и компиляторах. Поэтому такие идентификаторы не рекомендуется выбирать в качестве имен в прикладной программе на языке Си. Рекомендуется при программировании имена констант записывать целиком заглавными буквами [1, стр. 15].

6.1. Константы и строки

По определению, константа представляет значение, которое не может быть изменено. Синтаксис языка определяет 5 типов *констант*:

1. символы,
2. константы перечисляемого типа,
3. вещественные числа,
4. целые числа,
5. нулевой указатель («null»-указатель).

Управляющие последовательности ('\\n', '\\r', etc.) являются частным случаем экскейп-последовательностей (ESC-последовательностей), к которым также относятся лексемы вида '\\ddd', либо '\\xhh'.

Символьная константа (символ) имеет *целый тип*, то есть символы можно использовать в качестве целочисленных операндов в выражениях.

Целочисленные именованные константы можно вводить с помощью перечисления **enum**. Пример

```
enum DAY {SUNDAY, MONDAY, ...};  
enum BOOLEAN {NO, YES};
```

В первой строке DAY, а во второй строке BOOLEAN это необязательный произвольный идентификатор – название перечисления.

Если в списке нет ни одного элемента со знаком '=', то значения констант начинаются с 0 и увеличиваются на 1 слева направо. Таким образом, NO равно 0, а YES – 1. Именованная константа со знаком '=' получает соответствующее значение, а следующая за ней именованные константы без явных значений увеличиваются на 1 каждая.

То есть если

```
enum BOOLEAN {NO=10, YES};  
printf("NO=%d, YES=%d", NO, YES); // NO=10, YES=11
```

В Python можно сделать так

```
from enum import Enum, auto

class Boolean(Enum):
    NO = 0
    YES = auto()

Boolean.NO.value # 0
Boolean.YES.value # 1
```

Формально строки не относятся к константам языка Си, а представляют собой отдельный тип его лексем. Строковая константа определяется как последовательность символов, заключенных в двойные кавычки (не в апострофы).

Представление *строковых констант* в памяти ЭВМ подчиняются следующим правилам. Все символы строки размещаются подряд, и каждый символ (в том числе представленный эскейп-последовательностью) занимает ровно 1 байт. В конце записи строковой константы компилятор помещает символ `'\0'`.

Таким образом, количество байтов, выделяемое в памяти ЭВМ для представления значения строки, ровно на 1 больше, чем число символов в записи этой строковой константы.

При работе с символьной информацией нужно помнить, что длина символьной константы `'F'` равна 1 байту, а длина строки `"F"` равна 2 байтам.

6.2. Переменные и именованные константы

Одним из основных понятий языка Си является *объект* – именованная область памяти. Частный случай объекта – переменная.

Каждый из целочисленных типов (`char`, `short`, `int`, `long`) может быть определен либо как *знаковый* **signed** либо как *беззнаковый* **unsigned** (по умолчанию **signed**).

Различие между этими двумя типами – в правилах интерпретации *старшего бита внутреннего представления*. Спецификатор **signed** означает, что старший бит внутреннего представления воспринимался как знаковый; **unsigned** означает, что старший бит внутреннего представления входит в код представляемого числового значения, которое считается в этом случае беззнаковым. Выбор знакового или беззнакового представления определяет предельные значения, которые можно представить с помощью описанной переменной. Например на IBM PC переменная типа **unsigned int** позволяет представить числа от 0 до 65 535, а переменная типа **signed int** (или просто **int**) соответствуют значения в диапазоне от -32768 до +32767.

Именованные константы можно вводить с помощью *директивы препроцессора* **#define**, например

```
// препроцессорная константа
#define EULER 2.718282 // точка с запятой не нужна!!!
```

Что эквивалентно

```
const double EULER = 2.718282;
```

До начала компиляции текст программы на языке Си обрабатывается специальным компонентом транслятора – *препроцессором*. Далее текст от препроцессора поступает к компилятору. Итак, основное отличие констант, определяемых *препроцессорными директивами* **#define**, состо-

ит в том, что эти *константы вводятся* в текст программы *до этапа компиляции* – препроцессор обрабатывает исходный код программы и делает в этом тексте замены и подстановки [1, стр. 29].

7. Структура программы

Программ состоит из инструкций, расположенных в текстовом файле

```
<Подключение заголовочных файлов>
<Объявление глобальных переменных>
<Объявление функций и пр.>
int main(void) {
    <Инструкции>
    return 0;
}
<Определения функций и пр.>
```

В самом начале программы подключаются *заголовочные файлы*, в которых содержатся *объявления идентификаторов без их реализации*.

После подключения файлов производится *объявление глобальных переменных*. Глобальные переменные видны во всей программе, включая функции. Если объявить переменную внутри функции, то *область видимости переменной* будет ограничена рамками функции и в других частях программы использовать переменную нельзя. Такие переменные называются *локальными*.

При объявлении переменной можно сразу присвоить начальное значение. Присваивание значения переменной при объявлении называется *инициализацией переменной*.

```
int x = 10;
int x = 21; int y = 85; int z = 56;
```

Если глобальной переменной не присвоено значение при объявлении, то она будет иметь значение 0. Если *локальной* переменной не присвоено значение, то переменная будет содержать *произвольное значение*. Как говорят в таком случае: переменная содержит «мусор» [2, стр. 59].

После директив препроцессора точка с запятой не указывается. В этом случае концом инструкции является конец строки. Директиву препроцессора можно узнать по символу # перед названием директивы.

После объявления глобальных переменных могут располагаться *объявления функций*. Такие объявления называются *прототипами*. Схема прототипа функции выглядит следующим образом

```
<Тип возвращаемого значения> <Название функции>(
    [<Тип> [<Параметр 1>]
    [, ..., <Тип> [<Параметр N>]]]);
```

Например, прототип функции, которая складывает два целых числа и возвращает их сумму, выглядит так

```
int sum(int x, int y);
```

После объявления функции необходимо описать ее реализацию, которая называется *определением функции*. Определение функции обычно располагается после определения функции `main()`. Обратите внимание на то, что объявлять прототип функции `main()` не нужно.

Пример определения функции `sum()`

```
int sum(int x, int y) {
    return x + y;
}
```


Первая строка в определении функции `sum()` совпадает с объявлением функции. Следует заметить, что в объявлении функции можно не указывать названия параметров. Достаточно будет указать информацию о типе данных. Таким образом, объявление функции можно записать так

```
int sum(int, int);
```

После объявления функции ставится точка с запятой. Если функция не возвращает никакого значения, то перед названием функции вместо типа данных указывается ключевое слово `void`. Пример объявления функции, которая не возвращает значения

```
void print(int); // объявление функции; прототип

// определение функции, которая не возвращает значение
void print(int x) {
    printf("%d", x);
}
```

Самой главной функцией в программе является функция `main()`. Именно функция с названием `main()` будет автоматически вызываться при запуске программы. Функция имеет три прототипа

```
int main(void);
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char **penv);
```

Значение `void` внутри круглых скобок означает, что функция не принимает параметры. Второй прототип применяется для получения значений, указанных при запуске программы из командной строки. Количество значений доступно через параметр `argc`, а сами значения через параметр `argv`. Параметр `penv` в третьем прототипе позволяет получить значения переменных окружения.

Ключевое слово `int` означает, что функция возвращает целое число. Число 0 означает нормальное завершение программы. Если указано другое число, то это свидетельствует о некорректном завершении программы. Согласно стандарту, внутри функции `main()` ключевое слово `return` можно не указывать. В этом случае компилятор должен самостоятельно вставить инструкцию, возвращающую значение 0. Возвращаемое значение передается операционной системе и может использоваться для определения корректности завершения программы.

Вместо безликого значения 0 можно воспользоваться макроопределением `EXIT_SUCCESS`, а для индикации некорректного завершения программы – макроопределением `EXIT_FAILURE`. Предварительно необходимо включить заголовочный файл `stdlib.h`.

Пример программы

```
// Включение заголовочных файлов
#include <stdio.h>
#include <stdlib.h>

// Объявление глобальных переменных
int x = 21;
int y = 85;

// Объявление функций и пр.
int sum(int, int);
void print(int);
```

```
// Главная функция (точка входа в программу)
int main(void) {
    int z;
    z = sum(x, y);
    print(z);
    return EXIT_SUCCESS;
}

// Определение функций
int sum(int x, int y) {
    return x + y;
}

void print(int x) {
    printf("%d", x);
}
```

Объявление функций можно вынести в отдельный заголовочный файл и включить его с помощью директивы `#include`

MyPrototypes.h

```
#ifndef MYPROTOTYPES_H_
#define MYPROTOTYPES_H_

// Объявление функций и пр.
int sum(int x, int y);

#endif /* MYPROTOTYPES_H_ */
```

Директивы препроцессора `#ifndef`, `#define` и `#endif` препятствуют повторному включению заголовочного файла. Вместо этих директив можно указать в самом начале файла директиву препроцессора `#pragma` со значением `once`, то есть

MyPrototypes.h

```
#pragma once

// Объявление функций и пр.
int sum(int x, int y);
```

8. Ввод / вывод

8.1. Вывод данных

Для вывода одиночного символа в языке Си применяется функция `putchar()`: `putchar('w');`. Вывести строку позволяет функция `puts()`, которая выводит строку и вставляет символ перевода строки: `puts("String");`.

Для форматированного вывода используется функция `printf()`. Можно также воспользоваться функцией `_printf_l()`, которая позволяет дополнительно задать локаль. Функции `printf()` можно передавать обычные символы и спецификаторы формата, начинающиеся с символа `%`

```
printf("String\n");
printf("Count %d\n", 10); // спецификатор формата %d
printf("%s %d\n", "Count", 10); // спецификаторы формат %s и %d
```

NB: Тип данных переданных значений должен совпадать с типом спецификатора. Если, например, в качестве значения для спецификатора `%s` указать число, то это приведет к ошибке времени выполнения.

Спецификаторы имеют следующий синтаксис [2, стр. 66]

```
%[<Флаги>][<Ширина>[.<Точность>]][<Размер>]<Тип>
```

В параметре `<Тип>` могут быть указаны следующие символы:

- `c` – символ: `printf("%c", 'w');`
- `s` – строка: `printf("%s", "String");`
- `d` или `i` – десятичное целое со знаком: `printf("%d %i", 10, 30);`
- `u` – десятичное целое без знака: `printf("%u", 10);`
- `o` – восьмеричное число без знака: `printf("%#o %#o", 10, 77);`
- `x` – шестнадцатичное число без знака в нижнем регистре: `printf("%#x %#x", 10, 0xff);`
- `f` – вещественное число в десятичном представлении: `printf("%#.0f %.0f", 100.0, 100.0);`
- `e` – вещественное число в экспоненциальной форме: `printf("%e", 18657.81452);`
- `g` – эквивалентно `f` или `e` (выбирается более короткая запись числа):
`printf("%g %g %g", 0.086578, 0.000086578, 1.865E-05);`
- `p` – вывод адреса переменной: `printf("%p", &x);`
- `%` – символ процента: `printf("10%%");`

Параметр `<Ширина>` задает минимальную ширину поля. Если строка меньше ширины поля, то она дополняется пробелами. Если строка не помещается в указанную ширину, то значение игнорируется и строка выводится полностью:

```
printf("'%3s'", "string"); // 'string'
printf("%10s", "string"); // '      string'
```

Параметр `<Точность>` задает количество знаков после точки для вещественных чисел. Перед этим параметром обязательно должна стоять точка. Пример

```
printf("'%10.5f'", 3.1445454545); // ' 3.14454'
printf("'%3f'", 3.1434534545); // '3.143'
```

Вместо минимальной ширины и точности можно указывать символ `*`. В этом случае значения передаются через параметры функции `printf()` в порядке указания символов в строке формата.

В параметре `<Флаги>` могут быть указаны следующие символы:

- `#` – для восьмеричных значений добавляет в начало символ `0`, для шестнадцатичных значений добавляет комбинацию символов `0x` (если используется тип `x`) или `0X` (если используется тип `X`), для вещественных чисел указывает всегда выводить дробную точку, даже если задано значение `0` в параметре `<Точность>`.
- `0` – задает наличие ведущих нулей для числового значения,
- `-` – задает выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе: `printf("'%-5d'", 3);`
- пробел – вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус.
- `+` – задает обязательный вывод знака как для отрицательных, так и для положительных чисел.

8.2. Ввод данных

Для ввода одного символа предназначена функция `getchar()`. В качестве значения функция возвращает код введенного символа.

Для получения и *автоматического преобразования данных в конкретный тип* (например, в целое число) предназначена функция `scanf()`. При вводе строки функция не производит никакой проверки длины строки, что может привести к переполнению буфера. Функция возвращает количество произведенных присваиваний.

NB: Чтобы избежать переполнения буфера, обязательно указывайте *ширину* при использовании спецификатора `%s` (например, `%255s`).

```
int x = 0;

printf("Enter number: ");
fflush(stdout);
fflush(stdin);

int status = scanf("%d", &x); // &x -- адрес переменной 'x', а не ее значение!
if (status == 1) {
    printf("You entered: %d\n", x);
} else {
    puts("Error!");
}

printf("status = %d\n", status);
```

То есть конструкция в языке Си

```
float x = 0.0;
printf("Enter number: ");
scanf("%f", &x); // адрес переменной x
```

это то же самое, что в Python конструкция

```
x = float(input("Enter number: "))
```

Если ожидается строка, то символ `&` указывать не следует, так как имя переменной в случае строки это ссылка на первый элемент массива символов

```
char word[255] = "";
printf("Enter word: ");
fflush(stdout); // сброс буфера вывода в консоль
fflush(stdin); // очистка буфера ввода
// пользовательский ввод будет обрезан до 255 символов
scanf("%255s", word); // %255s чтобы избежать переполнения буфера

printf("You entered: %s", word);
```

Функция `fflush(stdout)` сбрасывает данные из буфера потока вывода `stdout` в консоль. Если *буфер вывода* принудительно не сбросить, пользователь может не увидеть подсказку вообще [2, стр. 74].

Функция `fflush(stdin)` очищает буфер потока ввода `stdin`. Если не очистить *буфер ввода* перед повторным получением числа, то это число может быть получено из предыдущего ввода из буфера. Например, при запросе первого числа было введено значение "47_3". Функция `scanf()` получит число 47, а второе число оставит в буфере, и оно будет доступно для следующей операции ввода.

Для ввода строки предназначена функция `gets()`, НО применять ее в программе не следует, так как функция не производит никакой проверки длины строки, что может привести к переполнению буфера.

Лучше получать строку посимвольно с помощью функции `getchar()` или воспользоваться функцией `fgets()`.

Строки в языке Си представляют собой последовательность (массив) символов, последним элементом которого является нулевой символ (`'\0'`).

Пример указателя

```
char *p = NULL;
```

То, что переменная `p` является указателем, говорит символ `*` перед ее именем при объявлении. Значением указателя является адрес данных в памяти компьютера. Указатель, которому присвоено значение `NULL`, называется нулевым указателем. Такой указатель ни на что не указывает, пока ему не будет присвоен адрес.

Размер массива символов и длина строки – это разные вещи. Размер массива – это общее количество символов, которое может хранить массив. Длина строки – это количество символов внутри символьного массива до первого нулевого символа.

```
char buf[256] = "abc"; // массив символов
printf("%s\n", buf); // abc
printf("%d\n", (int)sizeof(buf)); // 256
printf("%d\n", (int)strlen(buf)); // 3
```

Здесь `(int)sizeof(buf)` приводит результат вычисления `sizeof(buf)` к целочисленному типу.

Функция `getchar()` позволяет получить символ только после нажатия клавиши `<Enter>`. Если необходимо получить символ сразу после нажатия клавиши на клавиатуре, то можно воспользоваться функциями `_getche()` и `_getch()`.

Функция `_getche()` возвращает код символа и выводит его на экран. При нажатии клавиши функция `_getch()` возвращает код символа, но сам символ на экран не выводится. Это обстоятельство позволяет использовать функцию `_getch()` для получения конфиденциальных данных.

8.2.1. Получение данных из командной строки

Передать данные можно в командной строке после названия файла. Для получения данных в программе используется следующий формат функции `main()`

```
int main(int argc, char *argv[]) {
    // Инструкции
    return 0;
}
```

Параметр `argc` – количество аргументов, переданных в командной строке. Следует учитывать, что первым аргументом является название исполняемого файла, поэтому значение параметра `argc` не может быть меньше единицы. Через второй параметр `argv` доступны все аргументы в виде строки (тип `char *`). Квадратные скобки после названия второго параметра означают, что доступен массив строк.

Чтобы окно программы сразу не закрывалось следует использовать функцию `getchar()`

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void) {
    printf("Process ...\n");
    printf("Press key\n");
    fflush(stdout); // сбросить буфер вывода в консоль
    fflush(stdin); // очистить буфер ввода
    getchar(); // приостанавливает закрытие окна программы
    return 0;
}
```

8.2.2. Преждевременное завершение программы

Для того чтобы прервать выполнение программы дострочно можно использовать функцию `exit()`. В качестве параметра функция принимает число, которое является *статусом завершения*. Число 0 означает нормальное завершение программы, а любое другое число – некорректное завершение. Эти числа передаются операционной системе.

Вместо чисел можно использовать макроопределения `EXIT_SUCCESS` (нормальное завершение) и `EXIT_FAILURE` (аварийное завершение). Пример

```
exit(EXIT_FAILURE); // То же что exit(1);
```

9. Переменные и типы данных

Переменные – это участки памяти, используемые программой для хранения данных. Прежде чем использовать переменную, ее необходимо предварительно *объявить* глобально (вне функции) или локально (внутри функции). В большинстве случаев *объявление переменной* является сразу и ее *определением*.

При использовании старых компиляторов все локальные переменные должны быть объявлены в самом начале функции.

Каждая переменная должна иметь уникальное имя. Регистр бука имеет значение.

В языке Си доступны следующие элементарные типы данных:

- `_Bool` – логический тип данных. Может содержать значения «Истина» (соответствует числу 1) или «Ложь» (соответствует числу 0)

```
_Bool is_int = 1;
printf("%d\n", is_int); // 1
printf("%d\n", !is_int); // 0
```

В языке Си нет ключевых слов `true` и `false`. Вместо них используются числа 1 и 0 соответственно. Любое число, отличное от нуля, является истиной, а число, равно нулю, – ложью. Если требуется объявлять логические переменные так же как в языке C++, можно подключить заголовочный файл `stdbool.h`, в котором определены макросы `bool`, `true` и `false`.

- `char` – код символа. Занимает 1 байт. Пример:

```
char ch = 'w';
printf("%c\n", ch); // выводим символ w
printf("%d\n", ch); // выводим код символа
printf("%d\n", (int)sizeof(char)); // размер в байтах
```

- `int` – целое число со знаком.

- **float** – вещественное число. Занимает 4 байта.
- **double** – вещественное число двойной точности. Занимает 8 байт.
- **void** – означает отсутствие типа. Используется в основном для того, чтобы указать, что функция *не возвращает никакого значения* или *не принимает параметров*, а также для *передачи* в функцию данных произвольного типа.

Перед элементарным типом данных могут быть указаны следующие *модификаторы* или их комбинация:

- **signed** – указывает, что *символьный* или *целочисленный типы* могут содержать отрицательные значения. Тип **signed_int** (или просто **signed**) соответствует типу **int**.
- **unsigned** – указывает, что *символьный* или *целочисленный типы* не могут содержать отрицательные значения.
- **short** – может быть указан перед целочисленным типом. Занимает 2 байта.
- **long** – может быть указан перед целочисленным типом и типом **double**.

При использовании модификаторов тип **int** подразумевается по умолчанию, поэтому тип **int** можно не указывать.

Если переменная может изменять свое значение извне, то перед модификатором указывается ключевое слово **volatile**. Это ключевое слово предотвращает проведение оптимизации программы, при котором предполагается, что значение переменной может быть изменено только в программе.

Вместо указания конкретного целочисленного типа можно воспользоваться макроопределениями **__int8**, **__int16**, **__int32** и **__int64**. В заголовочном файле **stdint.h** объявлены знаковые типы фиксированной длины **int8_t**, **int16_t**, **int32_t** и **int64_t**, а также беззнаковые **uint8_t**, **uint16_t**, **uint32_t** и **uint64_t**.

После *объявления переменной* под нее *выделяется* определенная *память*, размер которой зависит от используемого типа данных и разрядности операционной системы [2, стр. 99].

Оператор **sizeof**, например в **(int)sizeof(x)** возвращает значение типа **size_t**. При объявлении переменной ей можно сразу присвоить начальное значение, указав его после оператора **=**. Переменная становится видимой сразу после объявления, поэтому на одной строке с объявлением (после запятой) эту переменную уже можно использовать для инициализации других переменных.

Локальные (объявленные внутри функции) переменные инициализируются *при каждом вызове функции*, а *статические* (сохраняющие свое значение между вызовами) *локальные переменные* – один раз при первом вызове [2, стр. 100].

Оператор **typedef** позволяет создать псевдоним для существующего типа данных. В дальнейшем псевдоним можно указывать при объявлении переменной.

Оператор имеет следующий формат

```
typedef long int lint;
lint x = 5L, y = 10L;
```

После создания псевдонима его имя можно использовать при создании другого псевдонима. Псевдонимы предназначены для создания машинно-независимых программ. Например тип данных **size_t** является псевдонимом, а не новым типом.

Константы – это участки памяти, значения в которых не должны изменяться во время работы программы.

9.1. Спецификаторы хранения

Перед модификатором и типом могут быть указаны следующие *спецификаторы хранения* [2, стр. 104]:

- **auto** – локальная переменная создается при входе в блок и удаляется при выходе из блока. Так как локальные переменные по умолчанию *автоматические*, ключевое слово **auto** в языке Си практически не используется.
- **register** – является подсказкой компилятору, что переменная будет использоваться *интенсивно*. Для ускорения доступа значения такой переменной сохраняется в *регистрах процессора*. Компилятор *может проигнорировать* это объявление и сохранить значение *в памяти*. Ключевое слово может использоваться при объявлении переменной внутри блока или в параметрах функции. К глобальным переменным не применяется.

```
register int x = 10;
```

- **extern** – сообщает компилятору, что переменная определена в другом месте, например, в другом файле. Ключевое слово лишь *объявляет* переменную, а *не определяет* ее. Таким образом, *память* под переменную *повторно не выделяется*. Если при объявлении переменной производится инициализация переменной, то *объявление становится определением переменной*

```
#include <stdio.h>

int main(void) {
    extern int x;      // Определена в другом месте
    printf("%d\n", x); // 10
}

// Другое место
int x = 10; // Определение переменной x
```

- **static** – если ключевое слово указано перед локальной переменной, то *значение будет сохраняться между вызовами функции*. Инициализация статических локальных переменных производится только при первом вызове функции. При последующих вызовах используется сохраненное ранее значение.

```
#include <stdio.h>

int func(void); // объявление функции

int main(void) {
    printf("%d\n", func()); // 1
    printf("%d\n", func()); // 2
    printf("%d\n", func()); // 3

    return 0;
}

int func(void) {
    // статические локальные переменные инициализируются только один раз
    // при первом вызове функции
    static int x = 0;
    ++x;
    return x;
}
```


Пори каждом вызове функции `func()` значение статической переменной `x` будет увеличиваться на единицу. Если убрать ключевое слово `static`, то при каждом вызове будет выводиться число 1, так как *автоматические локальные переменные* инициализируются при входе в функцию и уничтожаются при выходе из нее.

Если ключевое слово `static` указано перед *глобальной* переменной, то ее значение будет *видимо только в пределах файла*.

В Python поведение статической локальной переменной можно симитировать, например, с помощью ключевого слова `global`

```
x = 0

def func() -> int:
    global x
    x += 1
    return x

def main():
    print(func()) # 1
    print(func()) # 2
    print(func()) # 3

if __name__ == "__main__":
    main()
```

Очень важно учитывать, что переменная, объявленная *внутри блока*, *видна только в пределах блока* (внутри фигурных скобок) [2, стр. 106].

9.2. Массивы

Массив – это нумерованный набор переменных одного типа. Объем памяти массива (в байтах), занимаемый массивом, определяется так

```
<Объем памяти> = sizeof(<Тип>) * <Количество элементов>
```

Объявление массива выглядит следующим образом

```
<Тип> <Переменная>[<Количество элементов>];
// Пример
long arr[3] = {10, 20, 30};
```

После закрывающей фигурной скобки обязательно указывается точка с запятой. Количество значений внутри фигурных скобок может быть меньше количества элементов массива.

В рассмотренном примере первому элементу массива присваивается значение 10, второму – значение 20, а третьему элементу будет присвоено значение 0.

Если при объявлении массива указывается начальные значения, то количество элементов внутри квадратных скобок можно не указывать

```
long arr[] = {10, 20, 30};

for (int i = 0; i < 3; ++i) {
    printf("arr[i=%d] = %ld", i, arr[i]);
}
```

Если при объявлении массива начальные значения не указаны, то:

- элементам глобальных массивов автоматически присваивается значение 0;

- элементы локальных массивов будут содержать произвольные значения, так называемый «мусор».

NB: следует учитывать, что *проверка выхода* указанного *индекса* за пределы диапазона на этапе компиляции *не производится*. Таким образом, можно перезаписать значение в смежной ячейке памяти и тем самым нарушить работоспособность программы или *даже повредить операционную систему* [2, стр. 108].

Все элементы массива располагаются в *смежных ячейках памяти*. После определения массива выделяется необходимый размер пмяти, а в *переменной* сохраняется *адрес первого элемента* массива.

9.2.1. Строка

Строка является массивом символов, последний элемент которого содержит нулевой символ ('\\0'). Такие строки часто называют *C-строками*.

Присваивать строку в двойных кавычках можно только при инициализации. Объявить массив строк можно следующим образом

```
char valid_solver_names[][10] = {"gurobi", "cplex", "scip"};

// обойти массив строк
for (int i = 0; i < 3; ++i) {
    printf("%s", valid_solver_names[i]);
}
```

9.2.2. Указатели

Указатель – это *переменная*, которая предназначена для *хранения адреса*. В языке Си указатели часто используются в следующих случаях:

- для управления динамической памятью,
- чтобы иметь возможность изменить значение переменной внутри функции,
- для эффективной работы с массивами и др.

Объявление указателя имеет следующий формат:

```
<Тип> *<Переменная>;
// Пример
int *p = NULL;
```

Для того чтобы *указателю присвоить* *адрес переменной*, необходимо при присваивании значения перед названием переменной добавить оператор *&*. Типы данных переменной и указателя должны совпадать. Это нужно, чтобы при адресной арифметике был известен размер данных

```
int *p = NULL, x = 10;
p = &x; // присваивание указателю адреса переменной
printf("%d\\n", *p); // чтение значения по адресу; 10

*p = *p + 20; // изменение значения
printf("%d\\n", *p); // ; 30
```

Указатель, которому присвоено значение NULL (это макрос), называется *нулевым указателем*.

Глобальные и *статические локальные указатели* автоматические получают значение 0. Однако *указатели*, которые объявлены в *локальной области видимости*, будут иметь *произвольные*

значения. Если попытаться записать какое-либо значение через такой указатель, то можно повредить операционную систему. Поэтому, согласно соглашению, указатели, которые ни на что не указывают, должны иметь значение NULL [2, стр. 113].

При инициализации указателя ему можно присвоить не только числовое значение, но и строку.

Пример

```
const char *str = "String";
printf("%s\n", str);
```

Указатели можно сохранять в массиве. При объявлении *массива указателей* используется следующий синтаксис

```
<Тип> *(<Переменная>[<Количество элементов>]);
```

Пример использования массива указателей

```
int *p[3]; // Массив указателей из 3 элементов
int x = 10, y = 20, z = 30;

p[0] = &x; // указателю присваивается адрес переменной x
p[1] = &y;
p[2] = &z;

for (int i = 0; i < 3; i++) {
    printf("p[%d] = %d\n", i, *p[i]);
}
```

Объявление массива указателей на строки выглядит так

```
const char *str[] = {"String1", "String2", "String3"};
printf("%s\n", str[0]); // String1
```

Указатели очень часто используются для обращения к элементам массива, так как *адресная арифметика* выполняется *эффективнее*, чем доступ по индексу [2, стр. 114]

```
#include <stdio.h>
#define ARR_SIZE 3 // препроцессорная константа

int main(void) {
    int *p = NULL, arr[ARR_SIZE] = {10, 20, 30};
    p = arr; // присвоить указателю адрес первого элемента массива arr

    for (int i = 0; i < ARR_SIZE; ++i) {
        printf("arr[%d] = %d\n", i, *p);
        ++p; // переместить указатель на следующий элемент
    }
    p = arr; // восстановить положение указателя
}
```

В строке `p = arr;` указателю присваивается *адрес* первого элемента массива. Перед названием массива отсутствует оператор `&`, так как *название переменной* содержит *адрес первого элемента* массива

```
p = &arr[0]; // то же что p = arr;
```

В строке `++p;` увеличивается *значение указателя* на единицу. Здесь изменяется *адрес*, а не значение элемента массива. При увеличении значения указателя используются правила *адресной арифметики*, а не правила обычной.

Увеличение значения указателя на единицу означает, что значение будет *увеличено на размер типа*. Например, если тип `int` занимает 4 байта, то при увеличении значения на единицу указатель вместо адреса `0x0012FF30` будет содержать адрес `0x0012FF34`. Значение увеличилось на 4, а не 1.

Вместо двух инструкций внутри цикла можно использовать одну

```
printf("%d\n", *p++); // то же что *(p++)
```

Выражение `p++` возвращает *текущей адрес*, а затем увеличивает его на единицу. Символ `*` позволяет получить *доступ к значению элемента по указанному адресу*.

Получить доступ к элементу массива можно несколькими способами

```
int arr[3] = {10, 20, 30};

printf("%d\n", arr[1]); // 20
printf("%d\n", *(arr + 1)); // 20
printf("%d\n", *(1 + arr)); // 20
printf("%d\n", 1[arr]); // 20
```

С указателем можно выполнять следующие арифметические и логические операции:

- прибавлять целое число; число умножается на размер базового типа указателя, а затем прибавляется к адресу,
- вычитать целое число,
- вычитать один указатель из другого. Это позволяет получить количество элементов базового типа между двумя указателями,
- сравнивать указатели между собой.

При использовании ключевого слова `const` применительно к указателям важно учитывать местоположение ключевого слова `const` [2, стр. 116].

Например

```
// константный указатель
// const действует на значение
const char *str = "String";
char const *str = "String";
```

это одно и то же и означает, что значение, на которое ссылается указатель изменять нельзя, но указателю можно присвоить другой адрес.

В этом случае

```
char * const p = "String"; // const действует на адрес
```

значение, на которое ссылается указатель изменить можно, но указателю нельзя присвоить другой адрес.

А в этом случае

```
const char * const p = "String";
```

запрещается изменение значения, на которое ссылается указатель и присвоение другого адреса.

Указатели часто используются при передаче параметров в функцию. По умолчанию в функцию передается *копия значения переменной*. Если мы в этом случае изменим значение внутри функции, то это действие не затронет значения внешней переменной.

Для того чтобы иметь возможность *изменять* значение внешней переменной, параметр функции объявляют как *указатель*, а при вызове передают адрес переменной.

Передать параметры в функцию можно *по значению* (применяется по умолчанию). При этом создается *копия* значения, и все операции выполняются с копией. Так как локальные переменные видны только внутри тела функции, после завершения выполнения функции копия удаляется.

А можно передать *по ссылке*. Внутри функции адрес переменной присваивается указателю. Используя операцию *разыменования указателя*, можно *изменить* значение *самой переменной*, а не значение копии.

9.2.3. Динамическое выделение памяти

При объявлении переменной необходимо указать тип данных, а для массива дополнительно задать точное количество элементов. На основе этой информации при запуске программы автоматически выделяется необходимый объем памяти. После завершения программы память автоматически освобождается. Иными словами, объем памяти необходимо знать до выполнения программы. Во время выполнения программы создать новую переменную или увеличить размер существующего массива нельзя.

Для того чтобы произвести увеличение массива во время выполнения программы, необходимо выделить достаточный объем *динамической памяти*, перенести существующие элементы, а лишь затем добавить новые элементы. После завершения работы с памятью необходимо самим возвратить память операционной системе. Если этого не сделать, то участок памяти станет недоступным для дальнейшего использования. Подобные ситуации приводят к утечке памяти.

Для выделения динамической памяти в языке Си предназначена функция `malloc()`. Функция `malloc()` принимает в качестве параметра размер памяти в байтах и возвращает *указатель*, имеющий тип `void *`. В языке Си указатель типа `void *` неявно приводится к другому типу, поэтому использовать явное приведение не нужно (в языке C++ нужно обязательно выполнять явное приведение).

```
const unsigned ARR_SIZE = 3;
int *p = malloc(ARR_SIZE * sizeof(int));
```

Если память выделить не удалось, то функция возвращает *нулевой указатель*. Все элементы будут иметь произвольное значение, «мусор».

Освободить ранее выделенную динамическую память позволяет функция `free()`. Функция `free()` принимает в качестве параметра указатель на ранее выделенную память и освобождает ее.

Пример использования функций `malloc()` и `free()`

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main(void) {
    const unsigned ARR_SIZE = 10;
    int *p = malloc(ARR_SIZE * sizeof(int));

    if (!p) {
        puts("Oops ...");
        exit(1);
    }

    for (int i = 0; i < ARR_SIZE; ++i) {
        p[i] = i + 1;
    }
}
```

```

    }

    for (int i = 0; i < ARR_SIZE; ++i) {
        printf("p[%d] = %d", i, p[i]);
    }

    free(p); // NB
    p = NULL; // NB
}

```

Вместо функции `malloc()` можно воспользоваться функцией `alloc()`. Если память выделить не удалось, то функция возвращает нулевой указатель. Все элементы будут иметь значение 0.

Пример динамического выделения памяти под двумерный массив

```

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const unsigned int ROWS = 2;
    const unsigned int COLUMNS = 4;
    int i = 0, j = 0;

    int **p = calloc(ROWS, sizeof(int*));
    if (!p) exist(1);

    for (i = 0; i < ROWS; ++i) {
        p[i] = calloc(COLUMNS, sizeof(int));
        if (!p[i]) exit(1);
    }

    int n = 1;
    for (i = 0; i < ROWS; ++i) {
        for (j = 0; j < COLUMNS; ++j) {
            p[i][j] = n++;
            // (*(p + i) + j) = n++;
        }
    }

    for (i = 0; i < ROWS; ++i) {
        for (j = 0; j < COLUMNS; ++j) {
            printf("%3d", p[i][j]);
            // printf("%3d", (*(p + i) + j));
        }
        printf("\n");
    }

    for (int i = 0; i < ROWS; ++i) {
        free(p[i]);
        free(p);
        p = NULL;
    }
}

```

При возвращении памяти вначале освобождается память, выделенная ранее под строки, а лишь затем освобождается память, выделенная ранее под массив указателей.

Строки в памяти могут быть расположены в разных местах, что не позволяет эффективно получать доступ к элементам двумерного массива. Для того чтобы доступ к элементам сделать максимально быстрым, можно представить двумерный массив в виде одномерного массива.

```

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const unsigned ROWS = 2;
    const unsigned COLUMNS = 4;
    unsigned i = 0, j = 0;
    int *p = calloc(ROWS * COLUMNS, sizeof(int));
    if (!p) exit(1);

    int n = 1;
    for (i = 0; i < ROWS; ++i) {
        for (j = 0; j < COLUMNS; ++j) {
            *(p + i * COLUMNS + j) = n++;
        }
    }

    for (i = 0; i < ROWS; ++i) {
        for (j = 0; j < COLUMNS; ++j) {
            printf("%3d", *(p + i * COLUMNS + j));
        }
        printf("\n");
    }
    free(p); // освободить память
    p = NULL; // обнулить указатель
}

```

В данном случае все элементы расположены в смежных ячейках, и можно получить доступ к элементам с помощью указателя и адресной арифметики.

Функция `realloc()` выполняет перераспределение памяти. Функция выделит динамическую память длиной `newSize`, скопирует в нее элементы из старой области памяти, освободит старую память и вернет указатель на новую область памяти. Новые элементы будут иметь произвольные значения, так называемый «мусор». Если новая длина меньше старой, то лишние элементы будут удалены. Если память не может быть выделена, то функцию вернет *нулевой указатель*, при этом старая область памяти не изменяется (в этом случае возможны утечки памяти, если значение присваивается прежнему указателю).

Пример

```

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    unsigned arr_size = 5;
    int *p = malloc(arr_size * sizeof(int));
    if (!p) exit(1);

    for (int i = 0; i < arr_size; i++) {
        *(p + i) = i + 1; // p[i] = i + 1;
    }

    arr_size += 2;
    p = realloc(p, arr_size * sizeof(int));
    // Здесь возможна утечка памяти, если realloc() вернет NULL
    if (!p) exit(1);

    *(p + 3) = 55; // p[3] = 55;
    *(p + 4) = 66; // p[4] = 66;
}

```

```

    for (int i = 0; i < arr_size; i++) {
        printf("%d", *(p + i));
    }

    free(p);
    p = NULL;
}

```

9.2.4. Структуры

Структура – это совокупность переменных (называемых элементами, или полями), объединенных под одним именем. Объявление структуры выглядит следующим образом:

```

struct [<Название структуры>] {
    <Тип данных> <Название поля 1>;
    <Тип данных> <Название поля 2>;
    ...
} [<Объявление переменных через запятую>];

```

Допустимо не задавать название структуры, если после закрывающей фигурной скобки указано объявление переменной. *Точка с запятой* в конце объявления структуры является *обязательной*.

Объявление структуры только описывает *новый тип данных*, а не определяет переменную, поэтому память под нее не выделяется. Для того чтобы объявить переменную, ее название указывается после закрывающей фигурной скобки при объявлении структуры или отдельно с помощью названия структуры в качестве типа данных:

```

struct <Название структуры> <Названия переменных через запятую>;

```

Одновременно с объявлением переменной можно выполнить инициализацию полей структуры, указав значения внутри фигурных скобок:

```

struct Point point1 = {10, 20};

```

Можно указать имя поля

```

struct Point point2 = {.x = 100, .y = 200};

```

После объявления переменной выделяется необходимый размер памяти. Для получения размера структуры внутри программы следует использовать оператор `sizeof`:

```

<Размер> = sizeof(<Переменная>);
<Размер> = sizeof(<struct <>>);

```

Одну структуру можно присвоить другой с помощью оператора `=`. В этом случае копируются значения всех полей структуры.

Структуры можно вкладывать. При обращении к полю вложенной структуры дополнительно указывается название структуры родителя

```

#include <stdio.h>

struct Point {
    int x;
    int y;
};

```



```

struct Rectangle {
    struct Point top_left;
    struct Point bottom_right;
};

int main(void) {
    struct Rectangle rec = {
        .top_left = {
            .x = 100,
            .y = 200,
        },
        .bottom_right = {
            .x = -1,
            .y = -2,
        }
    };

    printf("top left x: %d", rec.top_left.x);
    printf("bottom right y: %d", rec.bottom_right.y);
}

```

Адрес структуры можно сохранить в указателе. Для получения адреса структуры используется оператор `&`, а для доступа к полю структуры вместо точки применяется оператор `->`

```

#include <stdio.h>

struct Point {
    int x;
    int y;
} point1;

int main(void) {
    struct Point *p = &point1;

    p->x = 10;
    p->y = 20;

    printf("%d\n", p->x);
    printf("%d\n", (*p).y);
}

```

9.2.5. Битовые поля

Битовые поля предоставляют доступ к отдельным битам, позволяя тем самым хранить в одной переменной несколько значений, занимающих указанное количество битов. Один бит может содержать только числа 0 или 1.

Битовые поля объявляются только с типом `int`. В одной структуре можно использовать одновременно битовые поля и обычные поля. Название битового поля можно не указывать. Кроме того, если длина поля составляет 1 бит, то дополнительно следует указать ключевое слово `unsigned`

```

struct Status {
    unsigned int flag1:1;
    unsigned int flag2:1;
    unsigned int flag3:1;
} status = {0, 1, 1};

```

```
printf("%d\n", status.flag1); // 0
status.flag2 = 1;
printf("%d\n", (int)sizeof(struct Status));
```

9.2.6. Объединения

Объединение – это область памяти, используемая для хранения данных *разных* типов. В один момент времени в этой области могут храниться данные только одного типа. Размер будет соответствовать размеру более сложного типа данных. Например, если внутри объединения определены переменные, имеющие типы `int`, `float` и `double`, то размер объединения будет соответствовать размеру типа `double`. Точка с запятой в конце объявления обязательна. Объединение только описывает новый тип данных, а не определяет переменную, поэтому память под нее не выделяется.

9.2.7. Перечисления

Перечисление – это совокупность целочисленных констант, описывающих все допустимые значения переменной. Точка с запятой в конце объявления является обязательной.

Пример

```
enum Color {
    RED, // 0
    BLUE, // 1
    GREEN, // 2
    BLACK // 3
};
```

Константам `RED`, `BLUE` ... автоматически присваиваются целочисленные значения, начиная с нуля.

При объявлении перечисления константе можно присвоить другое значение. В этом случае последующая константа будет иметь значение на единицу больше того, другого значения. Пример

```
enum Color {
    RED = 3,
    BLUE, // 4
    GREEN = 7,
    BLACK // 8
} color1;
```

Операторы инкремента и декремента могут использоваться в постфиксной или префиксной форме. При постфиксной форме (`x++`) сначала возвращается значение, а потом выполняется операция. В префиксной форме (`++x`) сначала выполняется операция, а потом возвращается значение.

При вложении операторов `if` можно воспользоваться следующей схемой

```
if (<Cond-1>) {
    // Block-1
}
else if (<Cond-2>) { // получается как elif в Python
    // Block-2
}
else {
    // Block-3
}
```

Тернарный оператор

```
int x = 10;
printf("%d %s\n", x, (x % 2 == 0) ? "- четное число" : "- нечетное число");
```

В качестве операндов указываются именно выражения, а не инструкции. Кроме того, выражения обязательно должны возвращать какое-либо значение, причем одинакового типа.

В качестве операнда можно указать функцию, которая возвращает значение

```
int func1(int x) {
    printf("%d %s\n", x, "...");
}

int func2(int x) {
    printf("%d %s", x, "...");
}

int x = 10;
// значение, возвращаемое оператором можно проигнорировать
(x % 2 == 0) ? func1(x) : func2(x);
```

В языке Си практически все элементарные типы данных (`_Bool`, `char`, `int`, `float`, `double`) являются числовыми. Тип `_Bool` может содержать только значения 1 и 0, которые соответствуют значениям Истина и Ложь. Тип `char` содержит *код символа*, а не сам символ. Поэтому значения этих типов можно использовать в одном выражении вместо со значениями, имеющими типы `int`, `float` и `double`. Пример

```
_Bool a = 1;
char ch = 'w';

printf("%d\n", a + ch + 10); // 130
```

Знак числа хранится в *старшем бите*: 0 соответствует положительному числу, 1 – отрицательному.

При преобразовании значения из типа `signed` в тип `unsigned` следует учитывать, что знаковый бит (если число отрицательное, то бит содержит значение 1) может стать причиной очень больших чисел, так как старший бит у типа `unsigned` не содержит признака знака

```
int x = -1;
printf("%u\n", (unsigned int)x); // 4294967295
```

Список литературы

1. Подбельский В.В., Фомин С.С. Программирование на языке Си, 2005. – 600 с.
2. Прохоренок Н.А. Язык С. Самое необходимое. – СПб.: БХВ-Петербург, 2020. – 480 с.

Листинги