

Приемы программирования на языке C

Содержание

1	Ресурсы по языку Си	4
2	Вводные замечания	4
3	Установка MSYS2 и MinGW-W64 для ОС Windows	5
4	Приемы работы в редакторе Eclipse	5
4.1	Настройка редактора Eclipse	5
4.2	Сборка и запуск проекта	6
4.3	Компиляция и запуск программы в редакторе Eclipse	8
5	Visual Studio Code как среда разработки для языка Си	8
6	Алфавит, идентификаторы, служебные слова	8
6.1	Константы и строки	8
6.2	Переменные и именованные константы	9
7	Директивы препроцессора	10
7.1	Макросы	10
7.1.1	Вариативные макросы	12
7.2	Условная компиляция	13
8	Указатели на переменные	14
8.1	Обобщенные указатели	15
8.2	Размер указателей	15
8.3	Висячие указатели	16
8.4	Указатели на функции	18
9	Сборка проекта на языке Си	21
9.1	Этап 1. Предобработка	22
9.2	Этап 2. Компиляция в ассемблерный код	22
9.3	Этап 3. Компиляция в машинные инструкции	23
9.4	Этап 4. Компоновка	24
9.5	Принцип работы компоновщика	26
10	Объектные файлы	27
10.1	Двоичный интерфейс приложений	28
10.2	Форматы объектных файлов	28
10.3	Ручная загрузка разделяемых библиотек	33

11 Структура памяти процесса	34
11.1 Внутреннее устройство памяти процесса	34
11.1.1 Сегмент BSS	35
11.1.2 Сегмент Data	36
11.1.3 Сегмент Text (или сегмент Code)	36
11.2 Исследование динамической схемы размещения в памяти	36
11.3 Отражение памяти	37
12 Стек и куча	38
12.1 Стек	38
12.1.1 Рекомендации по использованию стековой памяти	39
12.2 Куча	41
12.2.1 Выделение и освобождение памяти в куче	41
12.3 ООП	46
13 Композиция и агрегация	52
13.1 Композиция	53
13.2 Агрегация	56
14 Наследование и полиморфизм	60
14.1 Наследование	60
14.2 Полиморфизм	63
14.3 Полиморфное поведение в языке Си	64
14.4 Абстракция данных	64
14.5 Интерфейса командной оболочки для пользовательских приложений	65
15 Нововведения в Си	66
15.1 Удаление функции <code>gets</code>	66
15.2 Функции с проверкой диапазона	67
15.3 Невозвращаемые функции	67
15.4 Макросы для обобщенных типов	67
15.5 Unicode	68
15.6 Анонимные структуры и анонимные объединения	68
16 Конкурентность	68
16.1 Процессы и потоки	69
16.2 Синхронизация	69
16.3 Семафоры и мьютексы	70
16.4 Условные переменные	70
17 Многопоточное выполнение	72
17.1 Потоки	72
17.2 POSIX-потоки	72
17.3 Порождение POSIX-потоков	72

18 Синхронизация потоков	76
18.1 POSIX-потоки и память	78
18.2 Методы разделения ресурсов	79
19 Синхронизация процессов	79
19.1 Локальное управление конкурентностью	79
20 Локальные сокеты и IPC	80
20.1 Коммуникационные протоколы	80
20.2 Введение в программирование сокетов	81
20.2.1 Комьютерные сети	81
21 Программирование сокетов	83
22 Интеграция с другими языками	85
23 Модульное тестирование и отладка	85
23.1 Уровни тестирования	85
23.2 Модульное тестирование	86
24 Системы сборки	89
24.1 Make	90
24.2 CMake – не система сборки	93
25 Структура программы	94
26 Ввод / вывод	97
26.1 Вывод данных	97
26.2 Ввод данных	98
26.2.1 Получение данных из командной строки	100
26.2.2 Преждевременное завершение программы	100
27 Переменные и типы данных	100
27.1 Спецификаторы хранения	102
27.2 Массивы	104
27.2.1 Строка	104
27.2.2 Указатели	105
27.2.3 Динамическое выделение памяти	107
27.2.4 Структуры	110
27.2.5 Битовые поля	112
27.2.6 Объединения	112
27.2.7 Перечисления	112
28 Символы и С-строки	115

29 Пользовательские функции	117
29.1 Способы передачи параметров в функцию	119
29.2 Передача массивов и строк в функцию	119
29.3 Переменное количество параметров	120
29.4 Константные параметры	121
29.5 Статические переменные и функции	122
29.6 Указатели на функции	123
29.7 Передача в функцию и возврат данных произвольного типа	124
30 Чтение и запись файлов	124
31 Потоки и процессы	126
32 Создание библиотек	127
32.1 Статические библиотеки	127
Список литературы	129
Список листингов	129

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

- о вначале поиск программы выполняется в текущем рабочем каталоге (обычно это каталог, из которого запускается программа),
- о 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 `Help»Eclipse Markertplace` в строке Find вбить «Wrapper» 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 **OK**. 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, равно 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`, состоит 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. Директивы препроцессора

Задача препроцессора – заменить специальные директивы подходящим кодом на Си и подготовить итоговые исходники к компиляции. Управлять препроцессором в Си и влиять на его поведение можно с помощью набора *директив*. Они представляют собой строчки кода, начинающиеся символом `#` как в заголовочных, так и в исходных файлах. Эти строчки имеют смысл *только для препроцессора, но не для компилятора*.

Макросы можно применять различными способами. Ниже перечислено несколько примеров [3, стр. 28]

- определение константы,
- использование вместо обычной функции на Си,
- разворачивание цикла,
- предотвращение дублирования,
- условная компиляция.

7.1. Макросы

Для *определения макросов* используется *директива* `#define`. Каждый макрос имеет имя и (иногда) список параметров. У него также есть значение, которое подставляется вместо его имени на этапе работы *препроцессора* под названием «развертывание макроса». С помощью директивы `#undef` макрос можно сделать *неопределенным*.

```
#define ABC 5 // макрос

int main(int argc, char *argv[]) {
```

```

int x = 2;
int y = ABC;

int z = x + y;
return 0;
}

```

В приведенном выше листенге ABC – не переменная с целочисленным значением и не целочисленная константа. На самом деле это *макрос* с именем ABC, значение которого равно 5.

Препроцессор развернет макрос, подставив его значение туда, где было указано его имя, и вдобавок уберет комментарий в начальной точке.

Определение *функционального макроса*

```

#define ADD(a, b) (a + b) // функциональный макрос

int main(int argc, char **argv) {
    int x = 2;
    int y = 3;
    int z = ADD(x, y); // после обработки эта строка будет выглядеть так: int z = x + y;
    return 0;
}

```

ADD не является функцией. Это всего лишь *функциональный макрос*, который принимает аргументы.

Поскольку функциональные макросы могут принимать аргументы, с их помощью можно имитировать функции Си. Иными словами, можно вынести часто используемую логику в функциональный макрос [3, стр. 30].

Макросы существуют *только перед этапом компиляции*. То есть компилятор теоретически нечего о них не знает. О функциях компилятор знает все, поскольку они являются частью грамматики языка Си и хранятся в синтаксическом дереве. А макрос – просто директива, которую понимает только препроцессор.

Современные компиляторы Си знают о директивах и анализируют исходный код еще до его обработки препроцессором [3, стр. 30].

Большинство современных компиляторов позволяют просматривать результаты работы препроцессора непосредственно перед компиляцией. Например, можно указать флаг -E, чтобы вывести обработанный препроцессором код

```
$ gcc -E main.c > gcc.out
```

Единица трансляции (или *единица компиляции*) – код на языке Си, который прошел через препроцессор и готов к компиляции. В *единице трансляции* все директивы заменены *подключенными файлами* или *развернутыми макросами*, благодаря чему получается один длинный блок кода на Си.

Пример

```

#define PRINT(value) printf("%d\n", value);
#define LOOP(var, init_value, upper_bound) for (int var = init_value; var < upper_bound; var++)
{
#define ENDLLOOP }

int main(int argc, char **argv) {
    LOOP(counter, 0, 5)
        PRINT(counter)
    ENDLLOOP
}

```

```
}
```

Пример использования операторов # и ##

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

#define CMD(NAME) \
    char NAME ## _cmd[256] = ""; \
    strcpy(NAME ## _cmd, #NAME);

int main(int argc, char **argv) { // char *argv[]
    CMD(copy) // char copy_cmd[256] = ""; strcpy(copy_cmd, "copy");
    CMD(paste)

    char cmd[256];
    printf("Enter command: ");
    scanf("%s", cmd);

    if (strcmp(cmd, copy_cmd) == 0) {
        // ...
    }
    if (strcmp(cmd, paste_cmd) == 0) {
        // ...
    }
}
```

При разворачивании макроса оператор # переводит параметр в строковую форму, заключенную в кавычки. Например, NAME в "copy". А оператор ## соединяет параметры с другими элементами в определении макроса – обычно в целях формирования имен переменных.

7.1.1. Вариативные макросы

Вариативные макросы могут принимать переменное число аргументов

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

// препроцессорная константа
#define VERSION "2.3.4"

// вариативный макрос
#define LOG_ERROR(format, ...) \
    fprintf(stderr, format, __VA_ARGS__)

int main(int argc, char **argv) {
    if (argc < 3) {
        LOG_ERROR("Invalid number of arguments for version %s\n", VERSION);
    }

    if (strcmp(argv[1], "-n") != 0) {
        LOG_ERROR("%s is a wrong param at index %d for version %s", argv[1], 1, VERSION);
    }
}
```

Согласно определению макроса входящие аргументы argv[1], 1 и VERSION HE присваиваются ни одному из параметров. Поэтому при разворачивании макроса они будут использованы вместо __VA_ARGS__

```
$ ./sample 10
# Invalid number of arguments for version 2.3.4
```

Имитировать цикл с помощью макроса можно только одним способом: разместить отдельные повторяющиеся инструкции одна за другой. Это значит, что простой цикл с 1000 итерациями превратится в 1000 инструкций в коде на Си и в результате у нас не будет никакого реального цикла.

Использование представленного выше метода увеличивает размер двоичного файла, что можно считать недостатком. Размещение инструкций друг за другом вместо того, чтобы выполнять их в цикле, называется *развертыванием цикла* (loop unrolling) и хорошо подходит в определенных ситуациях: например, когда нужен приемлемый уровень производительности независимо от того, сколько ресурсов доступно в той или иной среде. Развертывание цикла можно считать компромиссом между размером двоичного файла и производительностью [3, стр. 38].

Архитекторы и проектировщики ПО руководствуются следующим правилом: «Если макрос можно оформить в виде функции языка Си, то нужно делать выбор в пользу функции!»

Когда проект разрабатывается в соответствии с общепринятыми принципами проектирования ПО, он обычно состоит из *множества легковесных двоичных файлов* с минимально допустимыми размерами, а не из одного огромного исполняемого файла [3, стр. 40].

Если нужна высокая производительность, то иногда приходится жертвовать стройной архитектурой и размещать компоненты линейно. Например, можно развернуть свои циклы.

В ходе проектирования мы пытаемся организовать компоненты в виде иерархии и избавиться от линейности, но центральный процессор рассчитан на то, что все инструкции уже загружены в линейном порядке и готовы к обработке.

Обсудим *развертывание циклов*. Эта методика в основном применяется в разработке для встраиваемых систем и *особенно в средах с ограниченными вычислительными ресурсами*. Она представляет собой *замену циклов линейными инструкциями*, что улучшает производительность и избавляет от накладных расходов на циклическое выполнение [3, стр. 41].

7.2. Условная компиляция

Условная компиляция – еще одна уникальная особенность Си. Она позволяет препроцессору генерировать разный исходный код в зависимости от тех или иных обстоятельств. В условной компиляции могут участвовать разные директивы:

- `#ifdef`,
- `#ifndef`,
- `#else`,
- `#elif`,
- `#endif`

Пример

```
#define CONDITION

int main(int argc, char **argv) {
#ifdef CONDITION
    int i = 0;
    i++;
#endif
    int j = 0;
```

```
    return 0;
}
```

Макросы можно определять, передавая команде компиляции параметры `-D`

```
$ gcc -DCONDITION -E main.c
```

Она особенно полезна, когда один и тот же исходник необходимо скомпилировать для разных архитектур, таких как Linux или macOS, с разными библиотеками и определениями макросов по умолчанию

```
$ gcc -DCONDITION -o main main.c
$ ./main
```

Директива `#ifndef` часто используется для *предотвращения дублирования заголовков*. Она не позволяет препроцессору подключить один и тот же заголовок дважды

```
#ifndef EXAMPLE_1_8_H
#define EXAMPLE_1_8_H

void say_hello();
int read_age();

#endif /* EXAMPLE_1_8_H */
```

Определения всех переменных и функций находится между `#ifndef` и `#endif`, что защищает их от повторного включения.

Когда заголовок подключается в первый раз, *макрос* `EXAMPLE_1_8_H` все еще не определен, поэтому препроцессор заходит в блок `#ifndef-#endif`. Следующая инструкция определяет макрос `EXAMPLE_1_8_H`, и препроцессор копирует в преобразованный код все, что находится выше директивы `#endif`. Когда происходит второе включение, макрос `EXAMPLE_1_8_H` уже определен, поэтому препроцессор пропускает все содержимое блока и переходит к следующей за `#endif` инструкцией, если таковая имеется.

Между `#ifndef-#endif` обычно размещают все содержимое заголовочного файла, а снаружи остаются лишь комментарии.

Для защиты от *двойного включения* вместо пары `#ifndef-#endif` можно использовать одну директиву `#pragma once`. Ее единственной особенностью является то, что, несмотря на поддержку в почти всех препроцессорах, она не входит в стандарт Си. Поэтому если ваш код должен быть переносимым, то от нее лучше отказаться.

```
#pragma once // вместо #ifndef-#endif

void say_hello();
int read_age();
```

8. Указатели на переменные

Указатели в случае некорректного использования могут привести к катастрофическим последствиям. В основе любого *указателя* лежит простая идея; это всего лишь *обычная переменная*, которая хранит *адрес памяти* [3, стр. 45].

Все эти объявления будут корректными с точки зрения Си:

- `int* ptr = 0;`

- `int * ptr = 0;`
- `int *ptr = 0;`

Оператор разыменования `*` открывает непрямой доступ к ячейки памяти, на которую указывает указатель. Иными словами, он позволяет считывать и модифицировать переменную через указатель, который на нее ссылается.

Нулевой указатель не содержит действительного адреса памяти. Следует избегать его разыменования, поскольку оно приводит к *неопределенному поведению* (что обычно заканчивается сбоем программы).

Обычно нам доступен макрос `NULL` со значением 0, который можно использовать для обнуления указателей в момент их объявления. Этому макросу следует отдавать предпочтение, поскольку он помогает отличать обычные переменные от указателей.

Указатели обязательно нужно инициализировать во время объявления!

На каждой платформе все указатели, хранящиеся в памяти, должны занимать ячейки определенного размера, то есть содержать одинаковое количество байтов. Но это вовсе не значит, что все они имеют одну и ту же длину арифметического шага.

Например, указатели типа `int` и `char` имеют одинаковый размер, но длина их арифметического шага отличается: `int*` обычно имеет четырехбайтный шаг, а `char*` – однобайтный. Следовательно, при инкрементировании целочисленного указателя мы перемещаемся в памяти на четыре байта вперед (добавляем 4 байта к текущему адресу), а инкрементация указателя на тип `char` приводит к перемещению вперед на 1 байт.

8.1. Обобщенные указатели

Указатели типа `void *` называют *обобщенными* [3, стр. 50] (в книге [2] они называются нетипизированными указателями). Как и любые другие указатели, они могут ссылаться на любой адрес, но *их фактический тип данных неизвестен*, поэтому мы не знаем длину их арифметического шага. Обобщенные указатели обычно используются для хранения содержимого других указателей без запоминания их типов. В связи с этим *их нельзя разыменовывать* и с ними *невозможно проводить арифметические операции*, поскольку мы не знаем их тип данных.

Большинство компиляторов запрещают разыменование обобщенных указателей

Длина шагов `char *` и `unsigned char *` равна 1 байту. Это делает данный тип указателей наиболее подходящим для побайтового перебора адресов в заданном диапазоне и их обработки байт за байтом.

`size_t` – это *стандартный* беззнаковый тип данных, который в языке Си обычно используется для хранения размеров [3, стр. 52].

Функция `size_t` описана в подразделе 6.5.3.4 стандарта ISO/ICE:9899:TC3 (спецификация C99), пересмотренная в 2007 году. На сегодняшний день она является основой для всех реализаций Си. Текст стандарта доступен по ссылке www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

8.2. Размер указателей

Размер указателя не является понятием языка программирования как такового, а зависит от конкретной архитектуры. Язык Си мало знает о подобных подробностях, относящихся к аппаратному обеспечению; он пытается предоставить универсальный способ работы с указателями

и другими аспектами программирования. Именно поэтому язык Си является стандартом; в нем описываются только сами указатели и их арифметика [3, стр. 53].

Функцию `sizeof` всегда можно использовать для получения размера указателя. Достаточно проверить результат `sizeof(char *)` в вашей текущей архитектуре. В 32- и 64-битных архитектурах указатели, как правило, занимают соответственно 4 и 8 байт, однако в некоторых системах могут иметь другой размер.

8.3. Висячие указатели

Указатель обычно содержит *адрес участка памяти*, выделенного для переменной [3, стр. 53]. Чтение или изменение адреса, по которому не зарегистрировано никакой переменной, считается большой оплошностью и может привести к сбою программы или *ошибке сегментации*. Она обычно возникает при неправильном использовании указателей. Вы обращаетесь к участку памяти, к которому у вас нет доступа. Раньше там находилась ваша переменная, но к моменту обращения она уже не существует.

Пример висячего указателя

```
#include <stdio.h>

int *create_an_integer(int default_value) {
    int var = default_value;

    return &var;
}

int main() {
    int *ptr = NULL;
    ptr = create_an_integer(10); // получится висячий указатель
    printf("%d\n", *ptr);
}
```

На MacOS проблемы висячего указателя не возникает, но на Linux будут проблемы. Указатель `ptr` – висячий, поскольку участок памяти, на который он ссылается (принадлежит переменной `var`), уже освобожден. Но вместе с тем функция возвращает адрес переменной, который затем присваивается указателю `ptr` внутри `main`.

Исправить ошибку можно с помощью *выделения кучи*

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

int *create_an_integer(int default_value) {
    int *var_ptr = (int *)malloc(sizeof(int));
    *var_ptr = default_value;

    return var_ptr;
}

int main() {
    int *ptr = NULL;
    *ptr = create_an_integer(10);
    printf("%d\n", *ptr);
    free(ptr);
}
```


Теперь переменная, которая создается внутри функции `create_an_integer`, больше не является локальной. Она выделяется в куче, и ее время жизни не ограничено функцией, в которой была объявлена. Следовательно, к ней может обратиться вызывающая (внешняя) функция. Указатели, ссылающиеся на эту переменную, больше не висят, и их можно разыменовывать [3, стр. 56].

В конце своего жизненного цикла переменная освобождается путем вызова функции `free`. Обратите внимание: любое содержимое кучи, которое больше не нуждается, нужно обязательно освобождать.

В Си, как и во многих других языках программирования, на которые он повлиял, функции возвращают лишь одно значение. В Си все функции блокирующие. Это означает, что вызывающий код, ждет пока они не закончат работу, и только потом собирает возвращенные результаты [3, стр. 56].

Помимо блокирующих функций, существуют и *неблокирующие*. Вызывающий код может продолжать выполняться, не дожидаясь их завершения. В таком случае обычно используется механизм обратных вызовов, которые срабатывают, когда вызванная функция завершает работу. Непрокирующие функции также иногда называют *асинхронными*. Поскольку в Си они не существуют, для их реализации применяются многопоточные решения.

Стек – участок памяти, который по умолчанию выделяется для всех локальных переменных, массивов и структур. Поэтому, когда вы объявляете локальную переменную в функции, она всегда создается в стеке – а если точнее, на его вершине [3, стр. 57].

Стек используется еще и для вызова функций. Перед началом выполнения новой функции ее адрес возврата и все передаваемые ей аргументы собираются в *стековый фрейм* и кладутся на вершину стека. Когда эта функция завершается, фрейм извлекается из стека, после чего начинают выполняться инструкции, находящиеся по адресу возврата (это обычно приводит к продолжению работы вызывающей функции).

Все локальные переменные, объявленные в теле функции, помещаются на вершину стека. Как следствие, все они освобождаются по завершении функции. Именно поэтому мы называем их локальными переменными и можем быть уверены в том, что одна функция не имеет доступа к переменным другой.

В языке Си аргументы в функцию можно передать только по значению!!! В Си нет никаких ссылок, что делает передачу по ссылке невозможной. Все, что передается функции, копируется в ее локальные переменные и перестает быть доступным, когда она завершается. [3, стр. 60]

Указатель передается в функцию по значению (копируется). Разыменование данного указателя внутри функции позволяет обратиться к переменной, на которую он ссылается.

Обычно в качестве аргументов рекомендуется использовать указатели, а не крупные объекты, и несложно догадаться почему. Копирование 8-байтного указателя куда более эффективно, чем копирование большого объекта, занимающего сотни байтов.

В языке Си во время вызова все аргументы передаются по значению, а разыменование указателей позволяет менять переменные вызывающей функции [3, стр. 60].

Структуры и массивы копируются последовательно, байт за байтом, поэтому их лучше передавать по указателю.

8.4. Указатели на функции

Указатель на переменную содержит ее адрес. Точно так же указатель на функцию содержит ее адрес, позволяя вызывать ее опосредованным образом.

```
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int main() {
    int (*func_ptr) (int, int); // указатель на функцию
    func_ptr = NULL; // обязательно!

    func_ptr = &sum; // можно просто func_ptr = sum;
    int result = func_ptr(5, 4);
    printf("Sum: %d\n", result);

    func_ptr = &subtract; // можно просто func_ptr = subtract;
    result = func_ptr(5, 4);
    printf("Subtract: %d\n", result);
}
```

В данном примере `func_ptr` – указатель, который может ссылаться только на определенный вид функций, соответствующих его сигнатуре. Это означает, что функции, адрес которых он содержит, должны принимать два целочисленных аргумента и возвращать целочисленный результат.

Указатели на функции, как и любые другие, необходимо как следует инициализировать. Если указатель не инициализируется непосредственно во время объявления, то его обязательно необходимо обнулить. Для указателей на функции обычно рекомендуется определить новый псевдоним типа

```
#include <stdio.h>

typedef int bool_t; // псевдоним типа, добавляется "_t"
typedef bool_t (*less_than_func_t) (int, int); // псевдоним типа, добавляется "_t"

bool_t less_than(int a, int b) {
    return a < b ? 1 : 0;
}

bool_t less_than_modular(int a, int b) {
    return (a % 5) < (b % 5) ? 1 : 0;
}

int main(int argc, char** argv) { // char *argv[]
    less_than_func_t func_ptr = NULL; // обязательно!

    func_ptr = &less_than;
    bool_t result = func_ptr(3, 7);
    printf("%d\n", result);
}
```

Ключевое слово **typedef** позволяет определить псевдоним для уже существующего типа. В приведенном выше примере используются два таких псевдонима: **bool_t** для типа **int** и **less_than_func_t** для указателей на функции типа **bool_t(*)(int, int)**. Благодаря этому код более понятен и можно выбирать более короткое имя длинного и сложного типа.

В Си к именам *новых типов* принято добавлять **_t**; это соглашение об именовании соблюдается во многих других стандартных псевдонимах, таких как **sizt_t** и **time_t**.

Если требуется определить собственные, пользовательские типы данных, не предусмотренные в самом языке, то нужны структуры.

Обратите внимание: ключевое слово **typedef** не имеет никакого отношения к пользовательским типам. Оно лишь создает *псевдоним для типов*, которые уже существуют. Если программе требуются совершенно новые типы, то следует задействовать *структуры*.

При объявлении структурной переменной в Си необходимо указать ключевое слово **struct**, например, **struct sample_t var**, в котором **struct** находится перед структурным типом **sample_t**. Для доступа к полям *структурной переменной* используется символ **.** или **->**.

Если вы не хотите набирать **struct** при определении каждого нового структурного типа и объявления каждой структурной переменной, то можете создать *для своих структур псевдонимы*. Используйте для этого **typedef**

```
typedef struct {  
    char first;  
    char second;  
    char third;  
    short fourth;  
} sample_t;
```

Теперь можно *объявить структурную переменную*, не прибегая к использованию ключевого слова **struct**

```
sample_t var;
```

По своему размещению в памяти *структурная переменная* очень похожа на *массив*. В нем все элементы размещаются в памяти последовательно; то же самое относится к структурной переменной и ее полям. Разница лишь в том, что в *массиве все элементы имеют один и тот же тип* и, следовательно, одинаковый размер, чего нельзя сказать о структурных переменных. В структуре каждое поле может иметь свой тип и размер.

Выравнивание данных в памяти. Прежде чем совершать какие-либо вычисления, процессор должен сначала загрузить из памяти подходящие значения, а затем сохранить полученный результат обратно в память.

Вычисления сами по себе происходят невероятно быстро, но доступ к памяти выполняется сравнительно медленно.

Обычно при каждом обращении к памяти процессор считывает определенное количество байтов. Эту величину принято называть *машинным словом*. Таким образом, память поделена на машинные слова, каждое из которых является атомарной единицей чтения и записи. Количество байтов в машинном слове зависит от архитектуры. Например, в большинстве 64-битных компьютеров слово занимает 32 бита, или 4 байта. Касательно выравнивания принято считать, что переменная выровнена в памяти, если ее первый байт совпадает с началом машинного слова. Так процессор может оптимизировать обращения к памяти, необходимые для загрузки ее значений [3, стр. 68].

Получается, что процессор читает и пишет данные *машинными словами*.

Например в структуре

```
struct sample_t {  
    char first;    // 1 байт  
    char second;   // 1 байт  
    char third;    // 1 байт  
    short fourth;  // 2 байта  
};
```

первые три поля занимают по 1 байту и находятся в первом машинном слове структуры; *все они могут быть прочитаны за одно обращение к памяти*. А вот четвертое поле, занимает 2 байта. Если забыть о выравнивании данных в памяти, то его начальный байт должен стать последним байтом первого машинного слова [3, стр. 68].

В подобном случае для загрузки значения данного поля из памяти *процессору* пришлось бы выполнить *два обращения к памяти* и заодно сместить определенные биты. Можно сказать, что первое машинное слово дополняется одним нулевым байтом. Этот нулевой байт был добавлен с целью завершить текущее слово и начать новое с четвертого поля.

Компилятор использует *дополнительные байты* для выравнивания значений в памяти.

Сложные типы данных

```
typedef struct {  
    int x;  
    int y;  
} point_t;  
  
typedef struct {  
    point_t center;  
    int radius;  
} circle_t;  
  
typedef struct {  
    point_t start;  
    point_t end;  
} line_t;
```

Размер сложных и простых структур вычисляется одним и тем же способом: путем сложения размеров всех полей. Конечно, не стоит забывать о выравнивании, которое может повлиять на размер сложной структуры. Так, если `sizeof(int)` равно 4 байтам, то `sizeof(point_t)` будет равно 8 байтам. Точно так же `sizeof(circle_t)` и `sizeof(line_t)` равны 12 и 16 байтам соответственно.

Структурные переменные часто называют *объектами*. Они выступают прямыми аналогами объектов в объектно-ориентированном программировании и могут помимо значений инкапсулировать и функции [3, стр. 70].

Структурный указатель (то есть указатель на структуру) ссылается на адрес первого поля структурной переменной. С точки зрения размещения в памяти структурные переменные очень похожи на массивы, в том смысле, что поля структуры размещаются в памяти последовательно.

Указатель на `circle_t` хранил адрес своего первого поля `center`; но, поскольку это поле в действительности являлось объектом (структурой) `point_t`, данный указатель фактически ссылался на первое поле этого объекта, `x`. Таким образом, у нас может быть три указателя с адресом одной и той же ячейки памяти.

```

#include <stdio.h>

typedef struct {
    int x;
    int y;
} point_t;

typedef struct {
    point_t center;
    int radius;
} circle_t;

int main(int argc, char** argv) {
    circle_t c; // объявляем структурную переменную
    circle_t* p1 = &c;
    point_t* p2 = (point_t*)&c;
    int* p3 = (int*)&c;

    printf("p1: %p\n", (void*)p1);
    printf("p2: %p\n", (void*)p2);
    printf("p3: %p\n", (void*)p3);
}

```

9. Сброска проекта на языке Си

Заголовочный файл обычно содержит перечисления, макросы и определения типов, а также *объявления* функций, глобальных переменных и структур. В языке Си объявление и определение некоторых элементов программирования, таких как функции, переменные и структуры, могут находиться в разных файлах.

Объявления принято хранить в заголовочных файлах, а соответствующие *определения* – в исходных. *Объявления функций* настоятельно рекомендуется размещать в заголовочных файлах, а их *определения* – в соответствующих исходных файлах.

К заголовочным файлам можно подключать только другие заголовочные файлы, но не исходники. К исходным файлам можно подключать только заголовочные файлы. **Подключение одних заголовочных файлов к другим считается дурным тоном.** Если вы так делаете, то это обычно говорит о серьезной проблеме в архитектуре вашего проекта.

Возвращаемый тип входит в состав объявления, но редко считается частью сигнатуры функции. Параметры функции рекомендуется именовать даже при объявлении.

Функция `main` является *точкой входа* в программу. Без нее нельзя получить двоичный файл для запуска программы. Эта функция интерпретируется компилятором как место, с которого нужно начинать выполнение.

Пример заголовочного файла

ExtremeC_examples_chapter2_1.h

```

#ifndef EXTREMEC_EXAMPLES_CHAPTER_2_1_H
#define EXTREMEC_EXAMPLES_CHAPTER_2_1_H

typedef enum {
    NONE,
    NORMAL,
    SQUARED
} average_type_t;

```

```
// объявление функции
double avg(int*, int, average_type_t);

#endif
```

В языке Си у *перечислений* не может быть отдельных объявлений и определений: они должны *объявляться и определяться* в одном и том же месте.

В любом проекте на языке Си функция `main` служит точкой входа в программу. Сборка проекта на C/C++ требует компиляции его кодовой базы в переносимые объектные файлы (которые еще называют промежуточными) и затем объединения их в конечные продукты, такие как *статические библиотеки* или *исполняемые файлы*.

В других языках программирования сборка проходит аналогичным образом, только промежуточные и конечные продукты будут иметь другие имена и, скорее всего, другие форматы. Например, в Java промежуточными продуктами выступают class-файлы с байт-кодом, а конечными – JAR- или WAR-файлы.

Два важных правила [3, стр. 82]:

1. Компилируются *только исходные файлы*. Заголовочные файлы не должны содержать ничего, кроме объявлений.
2. *Каждый исходный файл* компилируется *по отдельности*. Таким образом, если проект состоит из 100 исходных файлов, то придется скомпилировать каждый из них отдельно; то есть компилятор нужно будет запустить 100 раз! Можно подумать, что это много, но именно так следует компилировать проекты на C/C++.

К исходному файлу не подключаются другие исходники, поэтому он всегда компилируется отдельно. Согласно общепринятым рекомендациям, исходные файлы в C/C++ подключать нельзя!

Рассмотрим подробнее этапы сборки проекта на Си.

9.1. Этап 1. Предобработка

Это первый этап компиляции. К исходному файлу подключается ряд заголовочных. Иными словами, после предобработки мы получаем один фрагмент кода на языке Си, сформированный путем копирования заголовочных файлов в исходные.

Код, прошедший такую обработку, называется *единицей трансляции* (или *единицей компиляции*). Единица трансляции – это отдельный логический блок кода на Си, сгенерированный препроцессором и готовый к компиляции. В единице трансляции не остается ни одной директивы препроцессора!

Все объявления из заголовочного файла были скопированы в единицу трансляции, а все комментарии – удалены. Итак, на вход поступает *исходный файл*, а на выходе получается соответствующая *единица трансляции*.

9.2. Этап 2. Компиляция в ассемблерный код

После получения единицы трансляции можно переходить ко второму этапу – *компиляции*. На вход подается единица компиляции, полученная на предыдущем этапе, а на выходе получается соответствующий *ассемблерный код*. Он все еще может быть прочитан человеком, но уже зависит от аппаратной архитектуры и приближен к оборудованию.

Сохранить полученный ассемблерный код можно, передав компилятору gcc параметр -S. Итоговый файл будет иметь то же имя, что и исходник, только с расширением *.s

```
$ gcc -S main.c
```

В рамках данного этапа компилятор анализирует единицу трансляции и превращает ее в ассемблерный код, рассчитанный на целевую архитектуру. Под таковой понимается аппаратное обеспечение или центральный процессор, на котором будет выполняться скомпилированная программа.

Для нескольких архитектур может быть сгенерирован разный ассемблерный код. И это не смотря на тот факт, что использовался один и тот же исходный код на Си.

Целевой называется архитектурой, для которой компилируется исходный файл и на которой будет выполняться программа. Архитектура сборки используется для компиляции исходника и может отличаться от целевой. Например, исходный код на Си можно скомпилировать для 64-битного процессора AMD, используя 32-битный компьютер ARM.

Ассемблерный код крайне близок к языку, который понимает центральный процессор.

9.3. Этап 3. Компиляция в машинные инструкции

Цель – сгенерировать инструкции машинного уровня (или *машинный код*) на основе *ассемблерного кода*, созданного компилятором на предыдущем этапе.

У каждой архитектуры есть свой *ассемблер*, который может преобразовать собственный ассемблерный код в машинные инструкции. Объектные файлы – файлы с машинными инструкциями.

Двоичный файл и *объектный файл* содержат машинные инструкции и являются синонимами.

В большинстве Unix-подобных операционных систем есть утилита под названием **as**, с помощью которой *ассемблерный код* можно превратить в *переносимый объектный файл*.

Однако эти *объектные файлы* не являются исполняемыми; они содержат только те *машинные инструкции*, которые были сгенерированы для единицы трансляции. Таким образом, в объектном файле находятся машинные инструкции лишь для соответствующих функций и предварительно выделенных глобальных переменных.

```
$ as main.s -o main.o
```

Переносимые объектные файлы обычно имеют расширение *.o (или *.obj в Microsoft Windows).

Содержимое объектного файла *нельзя* представить в *текстовом виде*, и потому вам не удастся его прочитать. В связи с этим принято говорить, что объектный файл имеет *двоичное содержимое*.

Не все объектные файлы *переносимые*. Бывают еще *разделяемые*.

Сгенерировать соответствующий *объектный файл* (*.o) на базе исходного файла (*.c) можно с помощью параметра -c

```
$ gcc -c main.c
```

Под компиляцией зачастую имеют в виду первые три этапа, а не только второй. Иногда этот термин используется в качестве синонима «сборки» и подразумевает все четыре этапа. Например, выражение «процесс сборки Си» можно заменить на «процесс компиляции Си» [3, стр. 90].

Компиляцию можно считать *завершенной*, когда получен соответствующий *переносимый объектный файл*. Далее можно переходить к компиляции других исходников. Можно использовать два флага -s и -o.

Переносимые объектные файлы не являются исполняемыми! Если конечным продуктом компиляции проекта должен быть исполняемый файл, то нужно взять все переносимые объектные файлы, уже сгенерированные, и скомпоновать их в единое целое.

9.4. Этап 4. Компоновка

Чтобы получить еще один объектный файл (на этот раз исполняемый), нужно объединить уже полученные переносимые объектные файлы. Это делается путем *компоновки*.

Наборы инструкций разрабатываются изготовителями процессоров, такими как компании Intel и ARM. Кроме того, эти компании создают для своей архитектуры *специальную разновидность ассемблера*.

Программу можно собрать для новой архитектуры, если выполнены два условия [3, стр. 91]:

1. Известная версия ассемблера.
2. Доступна утилита (или программа) от соответствующего производителя, позволяющая скомпилировать *ассемблерный код* в *машинные инструкции*.

Если оба условия выполняются, то мы можем сгенерировать машинные инструкции из исходного кода на Си. И только после того их можно будет сохранить в объектных файлах подходящего формата, например ELF или Mach-O.

Определившись с версией ассемблера, утилитой для компиляции и форматом объектных файлов, можно создать на их основе другие инструменты.

Двумя инструментами, без которых нельзя обойтись при работе с новой архитектурой, являются:

- компилятор языка Си,
- компоновщик.

Для работы платформам, аналогичным Unix-подобным операционным системам, нужны ранее упомянутые инструменты, такие как ассемблер и компоновщик (помните, что их можно использовать отдельно от компилятора).

Компоновщик по умолчанию в Unix-подобных системах – `ld`. Но применять компоновщик в ручную непросто. К счастью, в Unix-подобных системах большинство известных компиляторов сами умеют передавать улитите `ld` подходящие параметры и указывать дополнительные объектные файлы. Поэтому нам не нужно использовать ее.

Вот более простой способ создать исполняемый файл

```
# gcc запускается без флагов
$ gcc main.o # будет сгенерирован исполняемый файл a.out
$ ./a.out
# processing ...
```

Препроцессор позволяет модифицировать исходный код до его компиляции. С его помощью можно разделять исходный код, например вынести объявления в заголовочные файлы и затем подключать их к разным исходникам.

Необходимо помнить: *препроцессор* не имеет никакого представления о языке Си и потому не в состоянии найти какие-либо синтаксические ошибки. Вместо этого он выполняет относительно простые операции, которые в основном заключаются в *замене текста*.

Препроцессор выполняет простые задачи – *копирует* содержимое другого файла или *развертывает макрос*, заменяя текст [3, стр. 95].

Ввиду разницы в грамматике *препроцессор* и *компилятор* языка Си использует разные синтаксические анализаторы.

Об особенностях устройства препроцессора GNU C компилятора gcc можно здесь <http://www.chiark.greenend.org.uk/doc/cpp-4.3-doc/cppinternals.html> В этом документе описывается как препроцессор анализирует директивы и создает синтаксическое дерево, а также алгоритмы развертывания макросов.

Как уже говорилось, *компилятор* принимает на вход *единицу трансляции*, подготовленную *препроцессором*, и генерирует соответствующие *инструкции ассемблера*. После компиляции множества исходников на языке Си начинается преобразование сгенерированного *ассемблерного кода* в *переносимые объектные файлы*, которые затем объединяются (возможно, с помощью других объектных файлов) в библиотеку или исполняемую программу; при этом применяются такие инструменты, как ассемблер и компоновщик, входящие в состав платформы.

Одна из сложностей компиляции кода на Си состоит в получении *корректных ассемблерных инструкций*, совместимых с *целевой архитектурой*. Утилита gcc позволяет компилировать один и тот же исходник для разных архитектур, таких как ARM, Intel x86, AMD и др. У каждой архитектуры есть *свой набор инструкций процессора* и полная ответственность за генерацию корректного ассемблерного кода для конкретной архитектуры лежит на компиляторе gcc (или другом).

Чтобы преодолеть указанные трудности, gcc (или любой другой компилятор) разделяет данную задачу на два этапа. Вначале он анализирует *единицу трансляции* и переводит ее в переносимую структуру данных под названием «*дерево абстрактного синтаксиса*» (abstract syntax tree, AST); данная структура не имеет прямого отношения к языку Си. Первый этап не зависит от конкретной платформы и набора поддерживаемых инструкций. Все, что относится к определенной архитектуре, происходит на втором этапе. За первый этап отвечает подкомпонент *интерфейс компилятора* (compiler frontend), а за второй – *кодогенератор* (compiler backend).

AST можно сгенерировать для любого языка программирования, а не только для Си, поэтому данная структура должна быть достаточно *абстрактной* и независимой от синтаксиса Си.

Получив дерево абстрактного синтаксиса, *кодогенератор* может приступить к его оптимизации и последующему созданию *ассемблерного кода для целевой архитектуры*.

Объектные файлы, сгенерированные в Linux для 64-битной архитектуры AMD, могут отличаться от результатов компиляции той же программы в другой ОС, такой как FreeBSD или macOS, и на том же оборудовании. Это значит, что несмотря на одинаковые машинные инструкции, объектные файлы не могут совпадать ввиду различных форматов, используемых в разных системах.

Иными словами, каждая ОС поддерживает свой двоичный формат (*формат объектных файлов*) для хранения инструкций машинного уровня. Поэтому содержимое объектного файла определяется двумя факторами: архитектурой (аппаратным обеспечением) и операционной системой. Эту комбинацию обычно называют *платформой*.

Объектные файлы и, следовательно, *ассемблер*, который их сгенерировал, зависят от платформы [3, стр. 100]. В Linux используется формат ELF (Executable and Linking Format – формат исполняемых файлов). Проще говоря, в Linux ассемблер создает ELF-файлы.

Сборка проекта на Си начинается с компиляции всех его исходников в соответствующие переносимые объектные файлы.

Какие продукты (их еще называют *артефактами*) можно сгенерировать в проекте на языке Си:

- ряд *исполняемых файлов*, которые в большинстве Unix-подобных операционных систем имеют расширение `*.out`. В Microsoft Windows, как правило, используется расширение `*.exe`,
- ряд *статических библиотек* с расширением `*.a` (в большинстве Unix-подобных систем) или `*.lib` (в Microsoft Windows),
- ряд *динамических библиотек* или разделяемых объектных файлов. В Unix-подобных операционных системах они обычно имеют расширение `*.so`, в macOS – `*.dylib` и в Microsoft Windows – `*.dll`.

Переносимые объектные файлы не входят в число этих артефактов. Они представляют собой *промежуточные ресурсы*, которые используются на этапе компоновки для получения перечисленных выше продуктов; далее они не нужны.

Исполняемые файлы, статические и динамические библиотеки называют *объектными файлами*, поэтому объектные файлы сгенерированные ассемблером в качестве промежуточных ресурсов, лучше называть *переносимыми*.

Исполняемый объектный файл можно запустить как *процесс*. Он должен иметь точку входа, с которой начинается выполнение машинных инструкций.

Статическая библиотека – не что иное, как *архив* с несколькими *переносимыми объектными файлами внутри*. Статические библиотеки обычно подключаются к другим исполняемым файлам и становятся их частью. Это самый простой и удобный способ инкапсулировать программную логику и использовать ее в дальнейшем.

Для сравнения, статические библиотеки становятся частью итогового исполняемого файла на *этапе компоновки*.

9.5. Принцип работы компоновщика

Пусть требуется собрать проект на языке Си, состоящий из 5 исходных файлов, и получить итоговую исполняемую программу. В процессе сборки вы скомпилируете все исходники и получите 5 *переносимых объектных файлов*. Далее для создания исполняемого файла понадобится *компоновщик*.

Компоновщик объединяет все переносимые объектные файлы вдобавок к указанным статическим библиотекам, чтобы получить готовую программу.

Если не вдаваться в подробности, то объектный файл содержит инструкции машинного уровня, эквивалентные единице трансляции. Однако они хранятся не в произвольном порядке, а сгруппированы в так называемые *символы*.

Для просмотра символов, хранящихся в объектном файле, можно воспользоваться утилитой `nm`

```
# символы переносимого объектного файла
$ nm target.o
```

Если требуется вывести под каждым символом дизассемблированные машинные инструкции, то можно использовать утилиту `objdump`

```
$ objdump -d target.o
```

Компоновщик собирает все символы из разных переносимых объектных файлов в один большой объектный файл, формируя тем самым исполняемую программу.

Пример

simple_header.h

```
#ifndef SIMPLE_HEADER_H
#define SIMPLE_HEADER_H

// прототипы функций
int add(int, int);
int multiply(int, int);

#endif
```

add.c

```
// заголовочные файлы НЕ подключаются!
int add(int x, int y) {
    return x + y;
}
```

multiply.c

```
// заголовочные файлы НЕ подключаются!
int multiply(int x, int y) {
    return x * y;
}
```

main.c

```
#include "simple_header.h"

int main(int argc, char** argv) {
    int x = add(4, 5);
    int y = multiply(x, 10);
}
```

Функция `main` ничего не знает об определениях `add` и `multiply`. Вопрос: каким образом она их находит, если в ней не указаны никакие другие исходные файлы? Ответ: *компоновщик* собирает вместе все необходимые *определения* из разных объектных файлов; таким образом, код, написанный в функции `main`, может обращаться к коду других функций [3, стр. 107].

Выведем символы

```
$ nm main.o
                 U _add # компилятор не нашел определения
0000000000000000 T _main
                 U _multiply # компилятор не нашел определения
```

Скомпоновать полученные объектные файлы можно так

```
$ gcc main.o add.o multiply.o # будет получен исполняемый файл a.out
```

Регистры – участки центрального процессора, к которым можно *быстро* обращаться. Прежде чем выполнять вычисления, значения лучше переместить из *основной памяти* в *регистры* – это обеспечит высокую эффективность [3, стр. 114].

10. Объектные файлы

Следует отметить, что *переносимые объектные файлы* считаются *промежуточными* ресурсами; это ингредиенты для другого рода продуктов, которые являются конечными.

10.1. Двоичный интерфейс приложений

Если *программный интерфейс приложения* API (Application Programming Interface) обеспечивает совместимость двух программных компонентов с точки зрения их функционального взаимодействия, то *двоичный интерфейс приложений* ABI (Application Binary Interface) гарантирует, что *две программы* и их соответствующие *объектные файлы* совместимы на уровне *машинных инструкций*.

Например, программа не может использовать динамические или статические библиотеки с другими ABI. Что еще хуже, *исполняемый файл* (которые, в сущности, является *объектным*) нельзя запустить в операционной системе, поддерживающей другой ABI.

ABI обычно содержит следующую информацию:

- набор инструкций целевой архитектуры, включая инструкции процессора, структуры памяти, порядок следования байтов, регистры и т.д.
- существующие типы данных, их размеры и правила выравнивания,
- соглашение о вызове функций, включая такие подробности, как структура *стекового фрейма* и порядок размещения аргументов в стеке,
- механизм *системных вызовов* в Unix-подобных ОС,
- используемые форматы *объектных файлов*, включая *переносимые, исполняемые и разделяемые*.

10.2. Форматы объектных файлов

На каждой платформе для хранения машинных инструкций используется свой формат объектных файлов. Речь здесь идет о структуре самих файлов, а не о наборе инструкций, которые поддерживает архитектура. Два отдельных аспекта ABI: формат объектных файлов и набор инструкций архитектуры.

Форматы объектных файлов в разных операционных системах:

- ELF применяется в Linux и многих других Unix-подобных ОС,
- Mach-O используется в системах OS X (macOS и iOS),
- PE применяется в Microsoft Windows.

Название *a.out* расшифровывается как *assembler output*. Несмотря на то что этот формат уже устарел, его название до сих пор используется по умолчанию в качестве имени *исполняемых файлов*, которые генерирует большинство *компоновщиков*.

ELF – стандартный формат объектных файлов для Linux и большинства Unix-подобных операционных систем. На самом деле он входит в состав System V ABI и активно применяется в большинстве систем семейства Linux.

Это значит, что один и тот же объектный файл ELF, созданный для одной операционной системы, можно запускать в другой (при условии, что используется та же архитектура). У ELF, как и у любого другого *формата файлов*, есть структура.

В переносимом объектом файле, полученном из скомпилированной единицы трансляции, можно найти следующие элементы:

- инструкции машинного уровня, сгенерированные из функций, найденных в единице трансляции (код),
- значения инициализированных глобальных переменных, объявленных в единице трансляции (данные),

- *таблицу символов*, содержащую все символы, которые были объявлены и на которые ссылается единица трансляции.

Это ключевые элементы, которые можно встретить в любом *переносимом объектном файле*. Конечно, то, как эти элементы структурированы, зависит от конкретного *формата*.

На Linux переносимые объектные файлы можно прочитать с помощью, например, `readelf`. На MacOS X можно воспользоваться утилитой `otool`

```
$ otool -l main.o
```

Исполняемый объектный файл это один из *конечных продуктов* (артефактов) компиляции проекта на языке Си. Они содержат те же элементы, что и их переносимые аналоги: машинные инструкции, значения для инициализированных глобальных переменных и таблицу символов. Отличается только расположение этих элементов.

Скомпоновать исполняемый объектный файл можно так

```
$ gcc funcs.o main.o -o ex3_1.out
```

Статическая библиотека в Unix – обычный архив с *переносимыми объектными файлами*. Сами по себе статические библиотеки не являются объектными файлами. Они скорее служат их *контейнерами*. Иными словами, это не ELF-файлы в Linux и не Mach-O-файлы в macOS. Это просто *архивы*, созданные Unix-утилитой `ar`.

Перед компоновкой из *статической библиотеки* извлекаются *переносимые объектные файлы*. Затем компоновщик ищет в них неопределенные символы и пытается их разрешить.

Создадим статическую библиотеку для проекта на C/C++ с несколькими исходными файлами. Вначале нужно создать *переносимые объектные файлы*, скомпилировав все исходники.

В системах Unix применительно к статическим библиотекам действует общепринятое соглашение об именовании. Имя файла должно начинаться с `lib` и иметь расширение `*.a`. В разных операционных системах могут действовать разные правила; например, в Microsoft Windows статические библиотеки имеют расширение `*.lib`.

Представьте гипотетический проект на языке Си с исходными файлами вида `aa.c`, `bb.c` и вплоть до `zz.c`. Чтобы сгенерировать из них *переносимые объектные файлы*, их нужно скомпилировать так

```
gcc -c aa.c -c aa.o
gcc -c bb.c -c bb.o
...
```

В результате выполнения этих команд мы получим нужные нам переносимые объектные файлы. Сборку можно существенно ускорить за счет мощного компьютера и параллельной компиляции.

Для создания *статической библиотеки* достаточно выполнять команду

```
$ ar crs libexample.a aa.o bb.o ... zz.o
```

В итоге получится архив `libexample.a` со всеми переносимыми объектными файлами, созданными ранее.

Еще раз, чтобы создать *статическую библиотеку* прежде всего нужно *скомпилировать исходные файлы* в соответствующие *переносимые объектные файлы*. Следует отметить, что если в проекте нет функции `main`, то скомпоновать результаты компиляции в исполняемый файл не получится [3, стр. 134].

Поэтому *переносимые объектные файлы* можно либо оставить как есть, либо объединить в статическую библиотеку.

Теперь можно создать *статическую библиотеку* из *переносимых объектных файлов*

Создание статической библиотеки из переносимых объектных файлов

```
# будет создан файл libgeometry.a (обычный архив)
$ ar crs libgeometry.a geom2d.o geom2d_2.o geom3d.o
$ mkdir -p /opt/geometry
$ mv libgeometry.a /opt/geometry
```

Если передать команде `ar` параметр `t`, можно посмотреть содержимое архива

```
# статическая библиотека состоит из 3 переносимых объектных файлов
$ ar t /opt/geometry/libgeometry.a
geom2d.o
geom2d_2.o
geom3d.o
```

Объявления нужны на этапе компиляции, ведь компилятору нужно знать о существовании типов, сигнатур функций и т.д. Для этого используются заголовочные файлы. При работе со статическими библиотеками в языке Си необходим доступ к предоставляемым ими *объявлениям*. Они известны как *публичный интерфейс* (чаще можно встретить название API).

API в языке Си (интерфейсы, предоставляемые библиотекой) обычно представлены в виде *набора заголовков*. Поэтому для написания новой программы, которая использует наши геометрические функции, достаточно иметь заголовчный файл и статическую библиотеку `libgeometry.a`.

Для использования статической библиотеки нужно написать еще один исходный файл, который будет подключать ее API и вызывать ее функции.

```
#include <stdio.h>
#include "geom.h" // API нашей библиотеки libgeometry.a

int main(int argc, char** argv) {
    cartesian_pos_2d_t cartesian_pos; // объявляем структурную переменную
    cartesian_pos.x = 100;
    cartesian_pos.y = 200;

    polar_pos_2d_t = polar_pos = convert_to_2d_polar_pos(&cartesian_pos);
    printf("Polar Position: Length: %f, Theta: %f (deg)\n", polar_pos.length, polar_pos.theta);
}
```

Следующая команда завершит сборку, выполнив компоновку и создав исполняемый объектный файл `ex3_3.out`; предполагается, что файл `libgeometry.a` находится в каталоге `/opt/geometry`

```
$ gcc main.o -L/opt/geometry -lgeometry -lm -o ex3_3.out
```

Рассмотрим каждый параметр этой команды:

- Параметр `-L/opt/geometry` сообщает компилятору `gcc`, что каталог `/opt/geometry` входит в число тех мест, в которых можно найти *статические* и *разделяемые библиотеки*. По умолчанию компоновщик ищет библиотечные файлы в традиционных каталогах, таких как `/usr/lib` или `/usr/local/lib`. Если не указать параметр `-L`, то компоновщик выполнит поиск только по этим стандартным путям.
- Параметр `-lgeometry` говорит компилятору `gcc`, что ему нужно искать файл `libgeometry.a` или `libgeometry.so`. Если передать параметр `-lxyz`, то компоновщик будет искать в заданных и стандартных каталогах файл `libxyz.a` или `libxyz.so`.

- Параметр `-lm` сообщает компилятору `gcc`, что нужно искать еще одну библиотеку с именем `libm.a` или `libm.so`. Она хранит определения математических функций.
- Параметр `-o ex3_3.out` говорит компилятору `gcc`, что итоговый исполняемый файл должен называться `ex3_3.out`.

Если все пройдет гладко, то после выполнения представленной выше команды у вас получится *исполняемый двоичный файл*, который содержит все *переносимые объектные файлы*, найденные в статической библиотеке `libgeometry.a` (по сути архив переносимых объектных файлов), плюс `main.o`.

После компоновки программа не будет зависеть от наличия статических библиотек, поскольку все их содержимое *встраивается* в ее *исполняемый файл*. Иными словами, *полученная программа является самостоятельной* и для ее запуска не требуется присутствия статической библиотеки [3, стр. 137].

Однако исполняемые файлы, полученные путем компоновки *большого количества статических библиотек*, обычно отличаются *огромными размерами*. Чем больше статических библиотек и переносимых объектных файлов у них внутри, тем крупнее программа. Иногда итоговый размер может достигать сотен мегабайт или даже нескольких гигабайтов.

Динамические (они же *разделяемые*) библиотеки – еще один продукт компиляции с возможностью повторного использования. От статических библиотек они отличаются тем, что *не входят в состав итогового исполняемого файла*. Вместо этого их *необходимо загружать и подключать во время запуска процесса*.

В *динамических* библиотеках неопределенные символы *могут оставаться и после работы компоновщика*; их поиск начинается в момент, когда программа готовится к *загрузке и выполнению*.

Во время загрузки исполняемого файла и его подготовки к выполнению в виде процесса используется *динамический компоновщик*, или просто *загрузчик*.

Перед самым запуском процесса *разделяемый объектный файл* загружается в участок памяти, доступный этому процессу. Данная процедура выполняется динамическим компоновщиком (или загрузчиком), который загружает и выполняет программу.

Скомпилируем тот же исходный код, чтобы получить из него разделяемый объектный файл.

```
$ gcc -c geom2d.c -fPIC -o geom2d.o
$ gcc -c geom3d.c -fPIC -o geom3d.o
...
```

Параметр `-fPIC` обязателен, если требуется создать *динамическую библиотеку* из набора *переносимых объектных файлов* (`*.o`). PIC расшифровывается как *position independent code* (позиционно независимый код). *Позиционная независимость* означает, что *инструкции* внутри *переносимого объектного файла* имеют не фиксированные, а *относительные адреса*; таким образом, в разных процессах они могут находиться на разных участках памяти.

Нет никакой гарантии, что динамический компоновщик будет загружать разделяемый объектный файл по одному и тому же адресу для разных процессов. На самом деле загрузчик отображает разделяемый объектный файл в память, и диапазоны адресов у этих отображений могут различаться. Если бы адреса инструкций были абсолютными, то мы не смогли бы загружать одну и ту же динамическую библиотеку сразу в *несколько участков памяти*, принадлежащих *разным процессам*.

Чтобы создать динамическую библиотеку, нам снова нужно будет воспользоваться компилятором. В отличие от *статической библиотеки*, которая является обычным архивом, разделяемый объектный файл остается *объектным* [3, стр. 140]

Создание динамической библиотеки из переносимых объектных файлов

```
$ gcc -shared 2d.o 3d.o trigon.o -o libgeometry.so
$ mv libgeometry.so /opt/geometry/
```

В первой команде мы передали параметр `-shared`, чтобы *компилятор* создал *разделяемый объектный файл* (`*.so`) из *переносимых* (`*.o`). В результате получилась библиотека под названием `libgeometry.so` (разделяемый объектный файл).

Замечание

Статическая библиотека встраивается в итоговый исполняемый файл (`*.out`). Динамическая библиотека НЕ встраивается в итоговый исполняемый файл (`*.out`), а загружается во время его запуска [3, стр. 141]

```
$ rm -fv /opt/geometry/libgeometry.a
$ gcc -c main.c -o main.o
$ gcc main.o -L/opt/geometry -lgeometry -lm -o ex3_3.out
```

Параметр `-lgeometry` заставляет компилятор найти библиотеку, *статическую* (`*.a`) или *динамическую* (`*.so`), и скомпоновать ее с оставшимися объектными файлами. Даже если заданная библиотека существует в двух вариантах, при компоновке программы `gcc` отдает предпочтение *разделяемому объектному файлу* (динамической библиотеке).

ВАЖНО! После компоновки со *статической библиотекой* исполняемый файл (`*.out`) получается *самодостаточным*, поскольку в него копируется все ее содержимое; таким образом, он больше не зависит от ее существования [3, стр. 141].

Описанное не относится к *разделяемым объектным файлам* (динамическим библиотекам). При запуске исполняемого файла `ex3_3.out`, полученного компоновкой с разделяемым объектным файлом, мы скорее всего получим ошибку

```
./ex3_3.out: error while loading shared libraries: libgeometry.so: cannot open shared object
file: No such file or directory
```

Загрузчику программы (динамическому компоновщику) не удалось найти `libgeometry.so` по своим стандартным поисковым путям. Поэтому нам нужно добавить к ним каталог `/opt/geometry`, в котором находится файл `libgeometry.so`. Для этого нужно *обновить переменную среды* `LD_LIBRARY_PATH` так, чтобы она указывала на текущий каталог.

Загрузчик проверит значение этой переменной среды и выполнит поиск необходимых разделяемых библиотек по заданному пути. Следует отметить, что в одной переменной можно указать несколько путей (используя `:` в качестве разделителя)

```
$ export LD_LIBRARY_PATH=/opt/geometry
$ ./ex3_3.out
```

Переменные среды часто указывают вместе с запуском программы

```
LD_LIBRARY_PATH=/opt/geometry ./ex3_3.out
```

Компонуя программу с несколькими разделяемыми объектными файлами, мы сообщаем системе о том, что в ходе ее запуска нужно найти и загрузить ряд динамических библиотек. Поэтому

при запуске программы загрузчик в первую очередь автоматически ищет эти *разделяемые объектные файлы* и отображает нужные символы в подходящие адреса памяти, доступные процессу. Только после этого процессор начинает выполнять код.

Или можно переместить динамическую библиотеку, например, по пути `/usr/local/lib`. Тогда исполняемый файл можно будет запустить просто как `./ex3_3.out`.

10.3. Ручная загрузка разделяемых библиотек

Разделяемые объектные файлы можно загружать и использовать иначе, не задействуя загрузчик (динамический компоновщик), который делает это *автоматически*. Идея в том, что программист загружает динамическую библиотеку вручную непосредственно перед использованием ее *символов* (функций).

Ручная загрузка динамической библиотеки

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

#include "geom.h"

polar_pos_2d_t (*func_ptr)(cartesian_pos_2d_t*); // указатель на функцию

int main(int argc, char** argv) {
    void* handle = dlopen("/opt/geometry/libgeometry.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }

    // dlsym возвращает указатель на функцию
    func_ptr = dlsym(handle, "convert_to_2d_polar_pos"); // поиск символа
    if (!func_ptr) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }

    cartesian_pos_2d_t cartesian_pos;
    cartesian_pos.x = 100;
    cartesian_pos.y = 200;
    polar_pos_2d_t polar_pos = func_ptr(&cartesian_pos);
    printf("Polar Position: Length: %f, Theta: %f (deg)\n", polar_pos.length, polar_pos.theta);
}
```

Функции `dlopen()` и `dlsym()`, чтобы загрузить разделяемый объектный файл и выполнить в нем поиск *символа* `convert_to_2d_polar_pos`. Функция `dlsym` возвращает указатель, с помощью которого можно вызывать нужную функцию.

Данный метод часто называют *отложенной загрузкой* разделяемых объектных файлов [3, стр. 144].

11. Структура памяти процесса

В языке Си управление *памятью* полностью ручное. Более того, вся ответственность за выделение областей памяти и их освобождение после того, как они больше не нужны, ложиться на программиста.

В высокоуровневых языках программирования, таких как Java или C#, управление памятью частично выполняется внутренней платформой языка – например, Java Virtual Machine (JVM) в случае Java. В таких языках программист занимается лишь выделением памяти, но не ее освобождением. Ресурсы освобождаются автоматически с помощью *сборщика мусора*.

В C/C++ сборщиков мусора нет!

11.1. Внутреннее устройство памяти процесса

При каждом запуске исполняемого файла операционная система создает новый процесс. Процесс – активная запущенная программа, которая загружена в память и имеет уникальный идентификатор (process identifier, PID). ОС полностью контролирует создание и загрузку новых процессов.

Процесс перестает быть активным либо в результате нормального завершения, либо при получении сигнала наподобие SIGTERM, SIGINT или SIGKILL, который в итоге заставляет его прекратить работу. SIGKILL останавливает процесс немедленно и принудительно:

- SIGTERM – сигнал, запрашивающий завершение; дает возможность процессу подготовиться к выходу,
- SIGINT – сигнал прерывания, который обычно отправляется активным процессам путем нажатия Ctrl+C,
- SIGKILL – сигнал немедленного завершения; принудительно закрывает процесс, не давая ему возможности подготовиться.

Когда операционная система создает процесс, одно из первых действий, которые она совершает, – выделяет участок памяти с заранее определенной внутренней структурой.

Память типичного процесса делится на несколько частей, которые называются *сегментами*. Каждый из них представляет собой область памяти с определенной задачей, предназначенную для хранения данных конкретного типа.

Список сегментов активного процесса [3, стр. 148]:

- сегмент неинициализированных данных или BSS (Block Started by symbol – блок, начинающийся с символа),
- сегмент данных,
- текстовый сегмент или сегмент кода,
- сегмент стека,
- сегмент кучи.

Исполняемый объектный файл и процесс – это разные вещи. *Исполняемый объектный файл* содержит *машинные инструкции* и за его создание отвечает компилятор. Но *процесс* – активная программа, созданная путем запуска исполняемого объектного файла.

Процесс – динамическая сущность, выполняемая внутри операционной системы, в то время как исполняемый объектный файл – просто набор данных с подготовленной начальной структурой, и на ее основе создается будущий процесс [3, стр. 148].

Статическая и динамическая схемы размещения в памяти включают определенный набор сегментов. Содержимое статической заранее записывается в исполняемый объектный файл во время компиляции исходного кода. А динамическая схема формируется инструкциями процесса, которые выделяют память для переменных и массивов и изменяют ее в соответствии с логикой программы.

Содержимое *статической памяти* можно предсказать по *исходному коду* или *скомпилированному объектному файлу*. Но с *динамической схемой* все сложнее, поскольку ее можно узнать, *только запустив программу*. Кроме того, при каждом выполнении одной и той же программы *содержимое динамической памяти может меняться*. Иными словами, динамическое содержимое любого процесса уникально, и исследовать нужно, пока он работает.

Инструменты, которые используются для исследования статической памяти, обычно рассчитаны на объектные файлы. Пример

simple.c

```
int main(int argc, char** argv) {  
    return 0;  
}
```

Нужно скомпилировать эту программу, чтобы получить итоговый исполняемый объектный файл

```
$ gcc simple.c -o ex_macos.out
```

Исполняемый объектный файл содержит предустановленную статическую схему размещения в памяти. Вывести статическую схему размещения в памяти можно так

```
# просмотр статических сегментов  
# размеры в байтах  
$ size ./ex_macos.out  
__TEXT __DATA __OBJC  others  dec hex  
16384 0 0 4294983680  4295000064  100008000
```

11.1.1. Сегмент BSS

Сегмент BSS (Block Started by symbol; блок, начинающийся с символа) фактически предназначен для хранения либо *неинициализированных*, либо *обнуленных глобальных переменных*.

С архитектурной точки зрения в алгоритмах обычно лучше использовать *локальные переменные*. Слишком большое количество глобальных переменных может увеличить размер двоичного файла. То есть объявление *неинициализированных* или *обнуленных* глобальных переменных увеличивает сегмент BSS [3, стр. 152].

Расширим предыдущий пример

simple.c

```
int global_var1; // неинициализированная глобальная переменная  
int global_var2; // неинициализированная глобальная переменная  
int global_var3 = 0; // обнуленная глобальная переменная  
  
int main(int argc, char** argv) {  
    return 0;  
}
```

Если теперь посмотреть статические сегменты

```
$ size -m ex-macos.out
Segment __PAGEZERO: 4294967296
Segment __TEXT: 16384
  Section __text: 22
  Section __unwind_info: 72
  total 94
Segment __DATA: 16384
  Section __common: 12 # это по сути сегмент BSS
  total 12
Segment __LINKEDIT: 16384
total 4295016448
```

Неинициализированные глобальные переменные по умолчанию *обнуляются*.

11.1.2. Сегмент Data

Чтобы продемонстрировать, какого рода содержимое находится в этом сегменте, объявим больше глобальных переменных, но на сей раз инициализируем их с помощью ненулевых значений.

```
int global_var1;
int global_var2;
int global_var3 = 0;

double global_var4 = 4.5;
char global_var5 = 'A';

int main(int argc, char** argv) {
    return 0;
}
```

Сегмент DATA предназначен для хранения *инициализированных* глобальных переменных с *ненулевыми* значениями.

11.1.3. Сегмент Text (или сегмент Code)

В *итоговый исполняемый файл* записываются *инструкции машинного уровня*. Поскольку *все машинные инструкции* программы находятся в *сегменте Text* (или *Code*), он должен находиться в исполняемом объектном файле – а именно, в его статической схеме размещения.

Процессор извлекает эти инструкции и *выполняет* их во время работы процесса.

11.2. Исследование динамической схемы размещения в памяти

Динамическая схема размещения находится в памяти процесса и существует до тех пор, пока он не завершится. Процедурой запуска исполняемого объектного файла занимается программа под названием «*загрузчик*». Он создает новый процесс и его начальную схему размещения в памяти, которая должна быть динамической. Для этого копируются сегменты, найденные в статической схеме размещения исполняемого объектного файла. Затем к ним добавляется два новых сегмента. И только после этого процесс может приступить к выполнению.

Если коротко, то в памяти активного процесса должно быть 5 сегментов, три из которых *копируются* непосредственно из статической схемы размещения исполняемого объектного файла, а остальные два создаются с нуля и называются *стеком* и *кучей*. Последние являются динамическими сегментами и существуют только в период работы процессы.

Стек – область памяти, в которой по умолчанию выделяется память для переменных. Стек имеет ограниченный размер, поэтому большие объекты в нем хранить нельзя.

Для сравнения, *куча* – более крупная и гибкая область памяти, в которой могут поместиться большие объекты и массивы.

Замечание

Динамическая схема размещения – не то же самое, что динамическое выделение памяти

Пять сегментов, из которых состоит динамическая память (то есть, как я понял, память активного процесса), ссылаются на разные участки основной памяти, уже выделенные и доступные только соответствующему процессу. Все эти сегменты – динамические в том смысле, что во время выполнения их содержимое постоянно меняется; исключение составляет только сегмент *Text* (сегмент кода), который является статическим и неизменяемым в буквальном смысле слова.

Сегмент BSS, сегмент данных (Data) и сегмент кода (Text/Code) – это статическая схема размещения в памяти, а стек и куча – это динамическая схема размещения в памяти [3, стр. 149]

11.3. Отражение памяти

Пример

sample.c

```
#include <unistd.h>

int main(int argc, char** argv) {
    while (1) {
        sleep(1);
    };
}
```

Компилируем пример (делаем из исходного файла *.c исполняемый объектный файл *.out)

```
$ gcc sample.c -o ex.out
```

Теперь можно запустить процесс в фоновом режиме

```
$ ./ex.out &
[1] 21881  # PID
```

Чтобы завершить процесс, следует воспользоваться командой kill

```
$ kill -9 <PID>
```

На компьютере под управлением Linux сведения о процессе можно найти в файлах внутри каталога /proc. Он находится в специальной файловой системе под названием *procfs*, которая отличается от обычных ФС тем, что не предназначена собственно для хранения данных; это скорее иерархический интерфейс для получения информации о разных свойствах отдельных процессов или системы в целом.

Файловая система *procfs* есть не только в Linux. Она обычно является частью Unix-подобных операционных систем, хотя используют ее не все они. Например, в FreeBSD она применяется, а в macOS – нет.

Стек обычно имеет ограниченный размер и плохо подходит для хранения крупных объектов. Если в этом сегменте закончится свободное место, то процесс больше не сможет вызывать функции, поскольку стек активно используется механизмом вызовов. В таких случаях процесс

принудительно завершается операционной системой. *Переполнение стека* – широко известная ошибка, которая возникает при полном заполнении стека.

Когда объявляется локальная переменная, она создается на *вершине стека*. При выходе из функции компилятор снимает со стека ее локальные переменные, в результате чего на вершину поднимаются значения внешней области видимости.

В стеке хранятся не только переменные. При каждом вызове функции на стек кладется новая запись под названием «*стековый фрейм*».

Поскольку стек ограничен в размере, в нем рекомендуется объявлять небольшие переменные. Кроме того, не следует создавать слишком много стековых фреймов в результате бесконечной рекурсии или множества вызовов функции.

Куча может достигать десятков гигабайтов. В ней обычно размещают постоянные, глобальные и очень большие объекты, такие как массивы и битовые потоки.

С локальными переменными, созданными на вершине стека, можно взаимодействовать напрямую, тогда как для доступа к содержимому кучи необходимо использовать указатели.

Статическая схема размещения в памяти находится в исполняемом объектном файле (*.out) и разбита на части, которые называются *сегментами*. В состав статической схемы размещения входят такие сегменты, как Text, Data и BSS [3, стр. 169].

Сегмент Text (Code) используется для хранения *инструкций машинного уровня*, предназначенных для выполнения, когда на основе текущего исполняемого файла создается новый процесс.

12. Стек и куча

12.1. Стек

Куча и стек – основные сегменты, с которыми работает программист. Data, Text и BSS используются реже, и доступ к ним ограничен. Причину тому факт, что данные сегменты генерируются компилятором и зачастую занимают небольшую долю в общем объеме памяти запущенного процесса. Это не значит, что они важны, но большую часть времени придется работать со стеком и кучей.

Стек и его содержимое тщательно спроектированы для обеспечения бесперебойной работы процесса. *Выделение памяти в стеке* происходит *быстро* и не требует применения никаких специальных функций. Более того, освобождение ресурсов и все действия по управлению памятью происходят *автоматически*.

Стек не очень большой, поэтому в нем нельзя хранить крупные объекты.

Объявление массива, выделенного на *вершине стека*

sample.c

```
#include <string.h>

int main(int argc, char** argv) {
    char arr[4];
    arr[0] = 'A';
    arr[1] = 'B';
    arr[2] = 'C';
    arr[3] = 'D';
}
```

Это простая программа и в ней легко разобраться, но в ее памяти происходит нечто интересное. Прежде всего, память, необходимая для массива `arr`, выделяется в *стеке*, а не в *куче* просто потому, что мы не использовали функцию `malloc` [3, стр. 173].

По умолчанию в *стеке* выделяются все *переменные* и *массивы*. Чтобы выделить место в *куче*, необходимо воспользоваться функцией `malloc` или ее аналогом `calloc`. В противном случае память будет выделена в *стеке*, а точнее, на его вершине [3, стр. 173].

Чтобы программу можно было отлаживать, ее двоичный файл должен быть собран соответствующим образом. То есть нам нужно сообщить компилятору о том, что в исполняемом файле должны быть *отладочные символы*.

Скомпилируем описанный выше пример с отладочным параметром `-g`

```
$ gcc -g sample.c -o ex_dbg.out
```

Параметр `-g` говорит компилятору о том, что итоговый исполняемый объектный файл должен содержать отладочную информацию. Он влияет на размер программы.

Замечание

По сравнению с другими сегментами *стек* заполняется в обратном порядке. Другие области памяти заполняются, начиная с младшего адреса, но со стеком все не так

Можно модифицировать данные, помещенные в стек. Если продолжить выполнение и если была модифицирована важная часть стека, то можно ожидать сбоя; в крайнем случае это изменение будет обнаружено неким механизмом, который прервет работу программы.

Значения нельзя записывать за рамками переменных и массивов, поскольку в стеке адреса растут в обратном направлении, что повышает вероятность модификации уже записанных байтов.

12.1.1. Рекомендации по использованию стековой памяти

Каждая переменная имеет свою область видимости, которая определяет ее время жизни. Это значит, что вместе с данной областью исчезают и все переменные, которые были в ней созданы.

Кроме того, только переменные, находящиеся в стеке, выделяются и освобождаются автоматически. Автоматическое управление памятью обусловлено природой стекового сегмента.

Любая переменная, которую вы объявляете в стеке, автоматически создается на его вершине. Выделение памяти происходит автоматически и может считаться началом жизни переменной. После этого поверх нее будет записано множество других переменных и стековых фреймов. Но пока она находится в стеке и над ней есть другие переменные, ее существование продолжается.

Однако рано или поздно программа должна завершиться, и перед ее выходом стек должен быть пустым. Поэтому в какой-то момент наша переменная будет извлечена из стека. Таким образом, освобождение ее памяти происходит автоматически и знаменует конец ее жизненного цикла. Вот почему мы говорим, что управление памятью стековых переменных происходит автоматически, без участия программиста.

Пример

```
int main(int argc, char** argv) {
    int a;
    ...
    return 0;
}
```

```
}
```

Эта переменная будет оставаться в стеке, пока не завершится функция `main`. Иными словами, переменная существует, пока остается действительной ее область видимости (функция `main`). И, поскольку в этой функции происходит выполнение всей программы, время жизни данной переменной почти такое же, как у глобального значения, доступного на протяжении всей работы процесса.

Тем не менее *глобальное значение* всегда остается в памяти, даже когда завершаются главная функция и сама программа, в то время как наша переменная будет извлечена из стека. Кроме того, стоит отметить, что глобальные переменные выделяются в другом сегменте памяти, `Data` или `BSS`, который ведет себя не так, как стек [3, стр. 180].

Чтобы компилятор `gcc` считал ошибками любые предупреждения, то передайте ему параметре `-Werror`. То же самое можно сделать и для отдельных предупреждений; например, в случае возвращения адреса локальной переменной, достаточно указать параметр `-Werror=return-local-addr`.

```
int* get_integer() {
    int var = 10; // объявление переменной на вершине стека
    return &var;
}

int main(int argc, char** argv) {
    int* ptr = get_integer();
    *ptr = 5;
    return 0;
}
```

Переменная `var` была локальной для функции `get_integer` и перестала существовать просто потому, что мы уже вышли из функции `get_integer` и соответствующей области видимости. Таким образом, возвращенный указатель получился *висячим*.

Обычно *указатели на переменные* в текущей области видимости рекомендуется передавать *другим функциям*, но не наоборот, поскольку они существуют, пока область видимости остается действительной. Дальнейшие вызовы кладут на стек новые значения, и текущая область видимости не исчезнет раньше них [3, стр. 183].

Ключевые характеристики стека [3, стр. 183]:

- стековая память имеет ограниченный размер, поэтому не подходит для хранения крупных объектов,
- адреса в стеке увеличиваются в обратном порядке, поэтому, перемещаясь вперед по стеку, мы читаем уже сохраненные байты,
- управление памятью в стеке происходит автоматически. Это касается как выделения, так и освобождения места,
- каждая переменная в стеке обладает областью видимости, которая определяет ее время жизни,
- указатели должны ссылаться только на те стековые переменные, которые все еще находятся в области видимости,
- освобождение памяти *стековых переменных* происходит автоматически, прямо *перед исчезновением области видимости*, и вы не можете на это повлиять,
- указатели на переменные, существующие в текущей области видимости, можно передавать в другие функции в качестве аргументов.

12.2. Куча

Почти любой код, написанный на любом языке программирования, так или иначе использует кучу. Это вызвано тем, что она имеет уникальные преимущества, недоступные при работе со стеком.

Но у этого сегмента памяти есть и недостатки. Например, выделение места в нем происходит медленней, чем в стеке.

Особенности кучи [3, стр. 184]:

- В куче никакие блоки памяти не выделяются автоматически. Вместо этого для выделения каждого участка памяти программист должен использовать `malloc` или аналогичную функцию.
- Куча большая. В то время как стек имеет ограниченный размер и не подходит для размещения крупных объектов, *куча* позволяет хранить *огромные объемы данных!*, достигающие десятков гигабайтов. По мере увеличения кучи аллокатору нужно запрашивать у операционной системы все больше страниц памяти, по которым распределяются блоки этого сегмента. Стоит отметить, что, в отличие от стека, куча выделяет адреса по направлению от младшего к старшему.
- Выделением и освобождением памяти в куче занимается программист. Во многих современных языках программирования *освобождение блоков кучи* выполняется *автоматически* параллельным компонентом под названием «сборщик мусора». Но в C/C++ такого механизма нет, и потому освобождать блоки кучи нужно вручную. Если не освободить выделенный блок, то может произойти утечка памяти.
- Переменные, выделенные в куче, не имеют никакой области видимости, в отличие от стековых переменных. Такое свойство можно считать отрицательным, поскольку оно существенно усложняет управление памятью. Вы не знаете, когда нужно освободить переменную, поэтому для эффективного использования кучи необходимо выработать собственные понятия области видимости и владельца.
- К блокам кучи можно обращаться только с помощью указателей. Иными словами, нет такого понятия как «переменная кучи». Для навигации по куче используются указатели.
- Поскольку куча доступна только владеющему ей процессу, для ее исследования нужен отладчик. К счастью, в языке Си указатели работают одинаково как со стеком, так и с блоками кучи. Таким образом, для исследования кучи и стека можно применять одни и те же методы.

12.2.1. Выделение и освобождение памяти в куче

Для получения блока памяти в куче используются функции `malloc`, `calloc` и `realloc`, а для освобождения памяти предусмотрена только одна функция – `free`.

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

void print_mem_maps() {
#ifdef __linux__
    FILE* fd = fopen("/proc/self/maps", "r");
    if (!fd) {
        printf("Could not open maps file.\n");
        exit(1);
    }
}
```

```

    char line[1024];
    while (!feof(fd)) {
        fgets(line, 1024, fd);
        printf("> %s", line);
    }
    fclose(fd);
#endif
}

int main(int argc, char** argv) {
    // Выделяем 10 байт без инициализации
    char* ptr1 = (char*)malloc(10 * sizeof(char));
    printf("Address of ptr1: %p\n", (void*)&ptr1);
    printf("Memory allocated by malloc at %p: ", (void*)ptr1);

    for (int i = 0; i < 10; i++) {
        printf("0x%02x ", (unsigned char)ptr1[i]);
    }
    printf("\n");

    // Выделяем 10 байт, каждый из которых обнулен
    char* ptr2 = (char*)calloc(10, sizeof(char));
    printf("Address of ptr2: %p\n", (void*)&ptr2);
    printf("Memory allocated by calloc at %p: ", (void*)ptr2);

    for (int i = 0; i < 10; i++) {
        printf("0x%02x ", (unsigned char)ptr2[i]);
    }
    printf("\n");

    print_mem_maps();

    free(ptr1);
    free(ptr2);

    return 0;
}

```

Указатели выделены в стеке, но ссылаются на какой-то другой сегмент памяти – в данном случае на кучу. Использование *стекового указателя* для работы с *блоком кучи* – распространенная практика.

Указатели `ptr1` и `ptr2` находятся в одной области видимости и освобождаются при завершении функции `main`, однако блоки памяти, полученные в куче, не входят ни в какую область видимости. Они будут существовать, пока их не освободят вручную.

`calloc` расшифровывается как *clear and allocate*, а `malloc` – *memory allocation*. То есть `calloc` очищает выделенный блок памяти, а `malloc` оставляет его инициализацию самой программе (то есть `malloc` сама *НЕ инициализирует* выделяемый блок памяти).

В спецификации языка сказано, что функция `malloc` не инициализирует выделяемый блок памяти. Значит исходить нужно из того, блок памяти, выделенный `malloc`, является неинициализированным и в случае необходимости его нужно инициализировать вручную.

Обратите внимание: благодаря этому `malloc` обычно работает быстрее, чем `calloc`. На самом деле некоторые реализации `malloc` *откладывают* выделение блока памяти до тех пор, пока к нему кто-то не обратится (для чтения или записи). Таким образом, ускоряется выделение памяти.

Если требуется инициализировать память, выделенную с помощью `malloc`, то можно воспользоваться функцией `memset`

```
#include <stdlib.h> // malloc
#include <string.h> // memset

int main(int argc, char** argv) {
    char* ptr = (char*)malloc(16 * sizeof(char));
    memset(ptr, 0, 16 * sizeof(char)); // заполняем нулями
    memset(ptr, 0xff, 16 * sizeof(char)); // заполняем байтами 0xff
}
```

Еще одно средство выделения памяти в куче – функция `realloc`. Она перераспределяет память, *изменяя размер* уже выделенного блока

```
int main(int argc, char** argv) {
    char* ptr = (char*)malloc(16 * sizeof(char));
    ...
    ptr = (char*)realloc(32 * sizeof(char));
    ...
    free(ptr);
}
```

Функция `realloc` *не изменяет* содержимое выделенной памяти, а просто *расширяет старый блок*. Если это не удастся сделать из-за фрагментации, то она находит другой блок подходящего размера и копирует в него данные из старого блока. Такое перераспределение может оказаться недешевой операцией, состоящей из множества этапов, и потому ее следует использовать с осторожностью.

Для поиска проблем с памятью в активном процессе можно использовать утилиту `valgrind`. Но сначала нужно собрать его с параметром `-g`.

```
$ gcc -g main.c -o ex.out
$ valgrind --leak-check=full ./ex.out
```

Каждый блок памяти (или переменная) *в стеке* имеет область видимости, поэтому определить его время жизни не составляет труда. Каждый раз, когда мы покидаем область видимости, все ее переменные исчезают. Но с кучей все намного сложнее.

У блока кучи *нет никакой области видимости*, и потому его время жизни является неочевидным и должно быть определено вручную [3, стр. 194].

Одна из лучших стратегий по преодолению сложностей, связанных с временем жизни кучи, состоит в определении *владельца блока памяти*, а не области видимости, которая вмещает в себя данный блок. Конечно, это нельзя назвать полноценным решением.

```
#include <stdio.h>
#include <stdlib.h> // для функций по работе с кучей

#define QUEUE_MAX_SIZE 100

typedef struct {
    int front;
    int rear;
    double* arr;
} queue_t;

void init(queue_t* q) {
    q->front = q->rear = 0;
```

```

    // выделенными здесь блоками кучи владеет ОБЪЕКТ ОЧЕРЕДИ (СТРУКТУРА queue_t)
    q->arr = (double*)malloc(QQUEUE_MAX_SIZE * sizeof(double));
}

void destroy(queue_t* q) {
    free(q->arr);
}

int size(queue_t* q) {
    return q->rear - q->front;
}

void enqueue(queue_t* q, double item) {
    q->arr[q->rear] = item;
    q->rear++;
}

double dequeue(queue_t* q) {
    double item = q->arr[q->front];
    q->front++;
    return item;
}

int main(int argc, char** argv) {
    // выделенными здесь блоками кучи владеет функция main
    queue_t* q = (queue_t*)malloc(sizeof(queue_t));

    // выделяем необходимую память для объекта очереди
    init(q);

    enqueue(q, 6.5);
    enqueue(q, 1.3);
    enqueue(q, 2.4);

    printf("%.1f\n", dequeue(q)); // 6.5
    printf("%.1f\n", dequeue(q)); // 1.3
    printf("%.1f\n", dequeue(q)); // 2.4

    destroy(q);
    free(q);
}

```

В этом примере используется два экземпляра владения для двух разных объектов. Первый относится к блоку кучи, на который ссылается указатель **arr** в структуре **queue_t**, принадлежащий объекту очереди. Пока этот объект существует, данный блок будет оставаться в памяти.

Второй экземпляр владения относится к блоку кучи, который функция **main** выделила в качестве заглушки для объекта очереди **q** (и владельцем которого является). Очень важно отличать блоки кучи, принадлежащие объекту очереди и функции **main**, поскольку освобождение одного не приводит к освобождению другого.

Факт владения блоком кучи со стороны объекта или функции следует отметить в комментариях. Код, которому этот блок не принадлежит, не должен его освобождать.

Кроме того, нужно сказать, что повторное освобождение одного и того же блока кучи приводит к повреждению памяти, и потому эту и любую другую проблему подобного рода следует исправлять сразу после обнаружения. В противном случае она может повлечь за собой внезапные сбои программы.

Помимо *стратегии владения*, можно также использовать *сборщик мусора* – автоматический механизм, встроенный в программу, пытающийся освободить блоки памяти, на которые не ссылается ни один указатель. В языке Си один из традиционных и широко известных инструментов этого типа является *консервативный сборщик мусора Бема-Демерса-Вайзера*; он представляет набор функций для выделения памяти, которые нужно вызывать вместо `malloc` и других стандартных операций языка Си.

Еще один подход к управлению *временем жизни блоков кучи* – использование *объекта RAII* (resource acquisition is initialization – «получение ресурса есть инициализация»). Это идиома объектно-ориентированного программирования, согласно которой время жизни ресурса (такого как выделенный блок кучи) можно привязать к времени жизни объекта. Иными словами, мы задействует объект, который во время своего создания инициализирует ресурс, а во время своего удаления – освобождает его. **К сожалению, данный метод нельзя реализовать в Си, поскольку в этом языке программиста не уведомляют об уничтожении объектов** Но можно эффективно применять в C++ с помощью деструкторов. Объект RAII инициализирует ресурс в своем конструкторе, а тот содержит код, отвечающий за деинициализацию. Следует отметить, что в C++ деструктор вызывается автоматически, когда объект удаляется или выходит из области видимости.

Факты и рекомендации по работе с кучей [3, стр. 196]:

- Выделение памяти в куче отнимает определенные ресурсы. Самой «дешевой» является функция `malloc`.
- Все блоки памяти, выделенные в пространстве кучи, должны быть освобождены либо сразу после того, как в них пропадает необходимость, либо перед завершением программы.
- Поскольку *у блоков кучи нет области видимости*, во избежание потенциальных утечек программа должна уметь управлять памятью.
- Выбранную стратегию управления памятью и предположения, на которых она основана, следует документировать на любом участке кода, где происходит доступ к блоку, чтобы в будущем программисты об этом знали.

У центрального процессора есть ряд внутренних регистров, которые работают очень быстро на чтение и запись. Кроме того, процессор должен копировать данные из оперативной памяти, которая во много раз медленнее его регистров. Здесь нужен механизм кэширования, иначе низкая скорость оперативной памяти станет доминирующей и нивелирует высокую производительность вычислений, свойственную процессору [?, стр. [201]amini-extreme-c:2022.

Файлы БД обычно хранятся на внешнем жестком диске, который на несколько порядков медленнее оперативной памяти. Очевидно, что здесь нужен механизм кэширования, иначе самая низкая скорость станет общим знаменателем и будет определять производительность всей системы [3, стр. 201].

В *кэше* центрального процессора находятся *последние инструкции и данные*, прочитанные из *более медленной оперативной памяти* [3, стр. 201].

При выполнении инструкций процессору сначала нужно получить *все необходимые данные*. Они находятся в *оперативной памяти* по определенному адресу, который указан в инструкции.

Перед вычислением *данные должны быть скопированы в регистры процессора*. Однако он обычно копирует больше блоков, чем ожидалось, и помещает их в свой *кэш*.

В следующий раз, если ему понадобится значение, находящееся недалеко от предыдущего адреса, он сможет найти его в кэше и избежать обращения к оперативной памяти, что намного

повысит скорость чтения. Если значение не удастся найти, то это промах кэша, в результате которого процессору придется считывать и копировать нужный адрес из оперативной памяти, что довольно медленно.

Элементы матрицы *развертываются в памяти по строкам*. Это значит, они сохраняются строка за строкой. Поэтому если процессор копирует из строки один элемент, то все остальные элементы в данной строке тоже, скорее всего, копируются *в кэш*. И, как следствие, *сложение лучше выполнять по строкам, а не по столбцам*.

В отличие от *стека*, в котором *выделение памяти* происходит *относительно быстро* и не требует получения дополнительного диапазона адресов, *куча* должна сначала найти *свободный блок памяти достаточного размера*, и это может быть затратной операцией [3, стр. 205].

12.3. ООП

В Си инкапсуляция происходит за счет *структур* [3, стр. 224]

```
typedef struct {
    int x, y; // будут проинициализированы нулями
    int red, green, blue; // будут проинициализированы нулями
} pixel_t; // к именам новых типов добавляется _t

pixel_t p1, p2; // объявление структурных переменных

p1.x = 56;
p1.y = 34;
p1.red = 123;
p1.green = 37;
p1.blue = 127;

p2.x = 212;
p2.y = 994;
p2.red = 127;
p2.green = 127;
p2.blue = 0;
```

Структуры Си не могут быть аналогами классов. К сожалению, с этим ничего нельзя сделать; *атрибуты* и *операции* существуют *отдельно* и в коде *связываются косвенно*. В Си любой *класс* является *неявным* и представляет собой совокупность структуры и списка функций [3, стр. 225].

Структура хранит только атрибуты, но не операции. Создание объекта очень похоже на объявление новой переменной. Сначала идет тип, затем имя (в данном случае имя объекта). При объявлении объекта почти одновременно происходит две вещи:

- о сначала для него выделяется память,
- о после чего его атрибуты инициализируются с помощью значений по умолчанию.

В Си, как и во многих других языках программирования, для доступа к атрибутам внутри объекта используется точка; если доступ к атрибутам структуры осуществляется не напрямую, а через *адрес внутри указателя*, то применяется стрелка (->). Выражение `p1.x` (или `p1->x`, если `p1` является *указателем*) следует читать как «*атрибут x в объекте p1*».

Неявная инкапсуляция (неявной ее делает то, что поведение в ней инкапсулируется так, что язык Си не знает об этом) предполагает следующее [3, стр. 227]:

- о Использование структур языка Си для атрибутов объекта (явная инкапсуляция атрибутов). Будем называть их *структурами атрибутов*.

- Для инкапсуляции поведения в Си применяются функции. **Язык Си не позволяет размещать функции в структурах.** Поэтому подобные функции должны существовать *за пределами структуры атрибутов* (неявная инкапсуляция поведения).
- В качестве одного из аргументов (обычно первого или последнего) поведенческие функции должны принимать *указатель на структуру*. Данный указатель ссылается на *структуру атрибутов объекта*.
- *Объявления поведенческих функций* обычно размещаются в одном заголовочном файле с *объявлением структуры атрибутов*.
- *Определения поведенческих функций* обычно находятся в одном или нескольких отдельных исходных файлах, которые подключают заголовок объявления.

ВАЖНО! При использовании неявной инкапсуляции у нас *нет никаких классов*, но их существование подразумевается самим программистом.

Заголовочный файл, приведенный ниже, содержит объявление нового типа, `car_t`, который представляет собой структуру атрибутов класса `Car`. Там же находятся объявления поведенческих функций этого класса. Под *классом Car* мы понимаем неявный класс, который на самом деле в коде отсутствует, но объединяет в себе *структуру атрибутов* и *поведенческие функции*.

class_car.h

```
#ifndef CLASS_CAR_H
#define CLASS_CAR_H

// эта структура содержит все атрибуты, относящиеся к объекту car
typedef struct {
    char name[32];
    double speed;
    double fuel;
} car_t; // постфикс _t для имени нового типа

// это объявления поведенческих функций
void car_construct(car_t*, const char*);
void car_destruct(car_t*);
void car_accelerate(car_t*);
void car_refuel(car_t*, double);

#endif
```

В Python класс выглядел бы так

```
class Car:
    def __init__(self, name: str, speed: float, fuel: float) -> None:
        self.name = name
        self.speed = speed
        self.fuel = fuel

    def construct(self, name: str):
        pass

    ...
```

У неявной инкапсуляции есть очень важная особенность: каждая структура атрибутов относится к отдельному объекту, но все объекты разделяют одни и те же поведенческие функции. Иными словами, для каждого объекта создается своя переменная на основе структуры атрибутов, но *поведенческие функции* существуют в *единственном экземпляре* и вызываются из разных объектов.

Сама по себе структура атрибутов `car_t` не является классом `Car`. Она просто содержит его атрибуты. Класс `Car` – совокупность всех объявлений.

```
#include <string.h>
#include "class_car.h"

// определения (реализации) подключенных выше функций
void car_construct(car_t* car, const char* name) {
    strcpy(car->name, name);
    car->speed = 0.0;
    car->fuel = 0.0;
}

void car_destruct(car_t* car) {
    // ...
}

void car_accelerate(car_t* car) {
    car->speed += 0.5;
    car->fuel -= 1.0;
    if (car->fuel < 0.0) {
        car->fuel = 0.0;
    }
}

void car_brake(car_t* car) {
    car->speed -= 0.07;
    if (car->speed < 0.0) {
        car->speed = 0.0;
    }

    car->fuel -= 2.0;
    if (car->fuel < 0.0) {
        car->fuel = 0.0;
    }
}

void car_refuel(car_t* car, double amount) {
    car->fuel = amount;
}
```

Все эти функции принимают в качестве первого аргумента *указатель* на `car_t` (что-то вроде `self` в Python). Если функция не принимает *указатель на структуру атрибутов*, то ее можно считать обычной функцией языка Си, которая не представляет поведение объекта.

Объявления поведенческих функций и соответствующей структуры атрибутов обычно находятся рядом. Причина в том, что за соответствие этих двух сущностей полностью отвечает программист, и поддержка данного кода должна быть достаточно легкой. Вот почему совместное хранение этих объявлений, обычно в одном заголовочном файле, помогает поддерживать в порядке общую структуру класса и облегчает его обслуживание в будущем.

Точка входа

```
#include <stdio.h>
#include "class_car.h"

int main(int argc, char** argv) {
    // создаем переменную объекта
    car_t car;
```

```

// создаем объект
car_construct(&car, "Renault");

car_refuel(&car, 100.0);
printf("Car is refueled, the correct fuel level is %f\n", car.fuel);
while (car.fuel > 0) {
    printf("Car fuel level: %f\n", car.fuel);
    if (car.speed < 80) {
        car_accelerate(&car);
        printf("Car has been accelerated to the speed: %f\n", car.speed);
    } else {
        car_brake(&car);
        printf("Car has been slowed down to the speed: %f\n", car.speed);
    }
}

printf("Car ran out of the fuel! Slowing down ... \n");
while (car.speed > 0) {
    car_brake(&car);
    printf("Car has been slowed down to the speed: %f\n", car.speed);
}

car_destruct(&car);
}

```

Объект можно инициализировать только после выделения памяти для его атрибутов.

Чтобы сделать атрибуты приватными, можно объявить структуру атрибутов без полей

ExtremeC_examples_chapter_3.h

```

// Публичный интерфейс класса List
#ifndef EXTREME_C_EXAMPLE_C_EXAMPLES_CHAPTER_6_3_H
#define EXTREME_C_EXAMPLE_C_EXAMPLES_CHAPTER_6_3_H

#include <unistd.h>

// объявление структуры атрибутов БЕЗ публично доступных полей
struct list_t;

// функция выделения памяти
struct list_t* list_malloc();

// конструктор и деструктор
void list_init(struct list_t*);
void list_destroy(struct list_t*);

// публичные поведенческие функции
int list_add(struct list_t*, int);
int list_get(struct list_t*, int, int*);
void list_clear(struct list_t*);
size_t list_size(struct list_t*);
void list_print(struct list_t*);

#endif

```

Здесь атрибуты становятся *приватными*. Если другой исходный файл (например, содержащий функцию main) подключит данный заголовочный файл, то не будет иметь доступ к атрибутам внутри типа list_t. Это легко объяснить: list_t – просто объявление без определения, которое не позволяет обращаться к полям структуры. Таким образом, мы соблюдаем принцип сокрытия.

Здесь приводится определение структуры атрибутов `list_t`

ExtremeC_examples_chapter6_3.c

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

#define MAX_SIZE 10

// создаем псевдоним bool_t
typedef int bool_t;

// определяем тип list_t
typedef struct {
    size_t size;
    int* items; // для массива целых чисел
} list_t;

// приватное поведение, которое проверяет, заполнен ли список
bool_t __list_is_full(list_t* list) {
    return (list->size == MAX_SIZE);
}

// еще одно приватное поведение для проверки индекса
bool_t __check_index(list_t* list, const int index) {
    return (index >= 0 && index <= list->size);
}

// выделяет память для объекта list
list_t* list_malloc() {
    return (list_t*)malloc(sizeof(list_t));
}

// конструктор объекта list
void list_init(list_t* list) {
    list->size = 0;
    // выделяет из кучи
    list->items = (int*)malloc(MAX_SIZE * sizeof(int));
}

// деструктор объекта list
void list_destroy(list_t* list) {
    // освобождает выделенную память
    free(list->items);
}

int list_add(list_t* list, const int item) {
    // использование приватного поведения
    if (__list_is_full(list)) {
        return -1;
    }

    list->items[list->size++] = item; // (1)(value <- size) => (2)(size++)
    return 0;
}

int list_get(list_t* list, const int index, int* result) {
    if (__check_index(list, index)) {
        *result = list->items[index]; // изменяем значение через адрес переменной
        return 0;
    }
}
```

```

}

void list_clear(list_t* list) {
    list->size = 0;
}

size_t list_size(list_t* list) {
    return list->size;
}

void list_print(list_t* list) {
    printf("[");
    for (size_t i = 0; i < list->size; i++) {
        printf("%d ", list->items[i]);
    }
    printf("]\n");
}

```

Все определения, приведенные в этом листинге, являются *приватными*. Внешней логике, которая будет использовать объект `list_t`, ничего о них не известно, и полагаться она может только на код в заголовочном файле.

Здесь не было подключено никаких заголовков! Совпадений сигнатур функций с объявлениями в заголовочном файле достаточно. Но все же заголовки *рекомендуется подключить*, поскольку это *гарантирует совместимость* между объявлениями и соответствующими определениями. Исходные файлы компилируются отдельно и только потом объединяются.

Приватные поведенческие функции можно обозначать и по-другому. Мы можем использовать в их именах префикс `__`. Например функция `__check_index` – приватная. Обратите внимание: у приватных функций нет соответствующих объявлений в заголовочном файле.

Точка входа

```

#include <stdlib.h>
#include "ExtremeC_examples_chapter6_3.h"

int reverse(struct list_t* source, struct list_t* dest) {
    list_clear(dest);
    for (size_t i = list_size(source) - 1; i >= 0; i--) {
        int item;
        if (list_get(source, i, &item)) {
            return -1;
        }
        list_add(dest, item);
    }
    return 0;
}

int main(int argc, char** argv) {
    struct list_t* list1 = list_malloc();
    struct list_t* list2 = list_malloc();

    // создание объектов
    list_init(list1);
    list_init(list2);

    list_add(list1, 4);
    list_add(list1, 6);
    list_add(list1, 1);
    list_add(list1, 5);
}

```

```

list_add(list2, 9);

reverse(list1, list2);

list_print(list1);
list_print(list2);

// уничтожение объектов
list_destroy(list1);
list_destroy(list2);

free(list1);
free(list2);

return 0;
}

```

Функции `main` и `reverse` используют только публичный API класса `List`: объявления структуры атрибутов `list_t` и его поведенческих функций.

Чтобы собрать пример, нужно сначала скомпилировать его исходные файлы.

```

$ gcc -c ExtremeC_examples_chapter6_3.c -o private.o
$ gcc -c ExtremeC_examples_chapter6_3_main.c -o main.o

```

Мы скомпилировали приватную часть кода в `private.o`, а главную – в `main.o`. Заголовочные файлы не компилируются! Публичные объявления, которые в них содержатся, становятся частью объектного файла `main.o`.

Скомпоновать этот пример нужно так

```

$ gcc main.o private.o -o ex.out
$ ./ex.out

```

Если не хочется повторять этап компоновки при каждом изменении реализации списка, то для хранения *приватного объектного файла* можно использовать разделяемую библиотеку (файл `*.so`). Ее можно загружать динамически во время выполнения, что избавляет от необходимости заново компоновать исполняемый файл [3, стр. 242].

13. Композиция и агрегация

Существует два подхода к созданию объектов: *прототипный* и *классовый*. В рамках прототипного подхода объект изначально создается пустым (без каких-либо атрибутов и без поведения) или копируется из существующего объекта. В этом контексте *экземпляр* и *объект* означают одно и то же. Таким образом, можно сказать, что прототипный подход основан на объектах; то есть все начинается с *пустых объектов*, а не с *классов*.

В случае с классовым подходом мы не можем создать объект без «схемы», известной как *класс*. Поэтому сначала пишем класс, а затем создаем из него объект.

Пусть требуется определить класс `Person`

```

typedef struct {
    char name[32]; // C-строка
    char surname[32];
    unsigned int age;
} person_t;

```

В Python можно было бы создать именованный кортеж

```
import typing as t

class Person(t.NameTuple):
    name: str
    surname: str
    age: int
```

Когда объект создается из класса, для него сначала выделяется память. Это место, в котором будут размещаться значения атрибутов. Затем атрибуты нужно инициализировать.

За создание объекта обычно отвечает отдельная функция, которую называют *конструктором*. Есть еще одна функция, которая отвечает за освобождение выделенных ресурсов. Она называется *деструктор*. После уничтожения объекта освобождается его память, но перед этим необходимо позаботиться об освобождении памяти, которую занимают все его ресурсы.

Сам *класс* не потребляет никакой памяти (по крайней мере в Си или C++) и существует лишь в виде исходного кода и только на этапе компиляции. Но *объекты существуют во время выполнения и потребляют память*.

Создание объекта начинается с выделения памяти. Освобождение памяти – последняя операция, которую выполняет объект.

13.1. Композиция

Отношение «композиция» между двумя объектами означает, что один из них обладает другим. Иными словами, один объект состоит из другого.

Время жизни внутреннего объекта привязано ко времени жизни внешнего объекта (контейнера). Если существует контейнер, то должен существовать и внутренний объект.

Пример композиции. Два заголовочных файла, которые объявляют публичные интерфейсы классов `Car` и `Engine`, двух исходных, содержащих реализацию этих классов, и еще одного исходного файла с функцией `main`.

Публичный интерфейс класса *Car* (ExtremeC_examples_chapter7_1_car.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_7_1_CAR_H
#define EXTREME_C_EXAMPLES_CHAPTER_7_1_CAR_H

// объявление структуры атрибутов
struct car_t;

// аллокатор памяти
struct car_t* car_new();

// конструктор
void car_ctor(struct car_t*);

// деструктор
void car_dtor(struct car_t*);

// поведенческие функции
void car_start(struct car_t*);
void car_stop(struct car_t*);
double car_get_engine_temperature(struct car_t*);

#endif
```

Здесь использовано предварительное объявление структуры атрибутов `car_t`. Определение структуры `car_t` будет находиться в исходном файле. Структура `car_t` считается *неполным* типом, который еще не определен.

Публичный интерфейс класса *Engine* (ExtremeC_examples_chapter7_1_engine.h)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_7_1_ENGINE_H
#define EXTREME_C_EXAMPLES_CHAPTER_7_1_ENGINE_H

// предварительное объявление структуры атрибутов
struct engine_t;

// аллокатор памяти
struct engine_t* engine_new();

// конструктор
void engine_ctor(struct engine_t*);

// деструктор
void engine_dtor(struct engine_t*);

// поведенческие функции
void engine_turn_on(struct engine_t*);
void engine_turn_off(struct engine_t*);
double engine_get_temperature(struct engine_t*);

#endif
```

В следующих листингах содержатся реализации классов `Car` и `Engine`.

Определение класса *Car* (ExtremeC_examples_chapter7_1_car.c)

```
#include <stdlib.h>
// Car может работать только с публичным интерфейсом Engine
#include "ExtremeC_examples_chapter7_1_engine.h"

typedef struct {
    // благодаря этому атрибуту устанавливается отношение композиции
    struct engine_t* engine;
} car_t;

car_t* car_new() {
    return (car_t*)malloc(sizeof(car_t));
}

void car_ctor(car_t* car) {
    // выделяем память для объекта engine
    car->engine = engine_new();

    // создаем объект engine
    engine_ctor(car->engine);
}

void car_dtor(car_t* car) {
    // уничтожаем объект engine
    engine_dtor(car->engine);

    // освобождаем память, выделенную для объекта engine
    free(car->engine);
}
```



```

void car_start(car_t* car) {
    engine_turn_on(car->engine);
}

void car_stop(car_t* car) {
    engine_turn_off(car->engine);
}

double car_get_engine_temperature(car_t* car) {
    return engine_get_temperature(car->engine);
}

```

Объект `car` содержит `engine`. В структуре атрибутов `car_t` есть новое поле типа `struct engine_t*`. Благодаря ему устанавливается отношение *композиции*.

Внутри этого исходного файла тип `struct engine_t*` по-прежнему остается *неполным*, но во время выполнения сможет сослаться на объект полного типа `engine_t`.

Указатель `engine` является *приватным*, доступным только внутри реализации. Когда вы реализуете *композицию*, ни один указатель не должен быть виден снаружи, иначе внешний код сможет изменять состояние внутреннего объекта [3, стр. 250]

Большинство случаев два объекта разных типов не должны знать о деталях реализации друг друга. Это продиктовано принципом сокрытия.

Теперь посмотрим на реализацию класса `Engine`

```

#include <stdlib.h>

typedef enum {
    ON,
    OFF
} state_t;

typedef struct {
    state_t state;
    double temperature;
} engine_t;

// аллокатор памяти
engine_t* engine_new() {
    return (engine_t*)malloc(sizeof(engine_t));
}

// конструктор
void engine_ctor(engine_t* engine) {
    engine->state = OFF;
    engine->temperature = 15;
}

// деструктор
void engine_dtor(engine_t* engine) {
    // Здесь ничего не происходит
}

// поведенческие функции
void engine_turn_on(engine_t* engine) {
    if (engine->state == ON) {
        return;
    }
    engine->state = ON;
}

```

```

    engine->temperature = 75;
}

void engine_turn_off(engine_t* engine) {
    if (engine->state == OFF) {
        return;
    }
    engine->state = OFF;
    engine->temperature = 15;
}

double engine_get_temperature(engine_t* engine) {
    return engine->temperature;
}

```

Точка входа

```

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

#include "ExtremeC_examples_chapter7_1_car.h"

int main(int argc, char** argv) {
    // выделяем память для объекта car
    struct car_t* car = car_new();

    // создаем объект car
    car_ctor(car);

    printf("Engine temperature before starting the car: %f\n", car_get_engine_temperature(car));
    car_start(car);
    printf("Engine temperature after starting the car: %f\n", car_get_engine_temperature(car));
    car_stop(car);
    printf("Engine temperature after stopping the car: %f\n", car_get_engine_temperature(car));

    // уничтожаем объект
    car_dtor(car);

    // освобождаем память, выделенную для объекта car
    free(car);
}

```

Чтобы собрать этот пример, нужно сначала скомпилировать три предыдущих исходных файла. Затем их следует компоновать, чтобы сгенерировать итоговый исполняемый объектный файл.

```

$ gcc -c ExtremeC_examples_chapter7_1_engine.c -o engine.o
$ gcc -c ExtremeC_examples_chapter_7_1_car.c -o car.o
$ gcc -c main.c -o main.o
$ gcc engine.o car.o main.o -o ex.out
$ ./ex.out

```

13.2. Агрегация

В агрегации тоже применяется контейнер, который содержит другой объект. Основное отличие в том, что *время жизни контейнера никак не связано с временем жизни содержащегося в нем объекта* [3, стр. 253].

Для сравнения, *композиция* подразумевает, что время жизни контейнера должно быть не короче времени жизни внутреннего объекта.

Объект `player` будет какое-то время играть роль контейнера, а `gun` будет выступать внутренним объектом, пока игрок держит его в руках. Эти два объекта имеют независимое время жизни.

Публичный интерфейс класс `Gun` (`ExtremeC_examples_chapter7_2_gun.h`)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_7_2_GUN_H
#define EXTREME_C_EXAMPLES_CHAPTER_7_2_GUN_H

typedef int bool_t;

// предварительное объявление структуры атрибутов
struct gun_t;

// аллокатор
void gun_ctor(struct gun_t*, int);

// деструктор
void gun_dtor(struct gun_t*);

// поведенческие функции
bool_t gun_has_bullets(struct gun_t*);
void gun_trigger(struct gun_t*);
void gun_refill(struct gun_t*);

#endif
```

Здесь находится только *объявление структуры атрибутов* `gun_t`, поскольку мы еще не определили ее поля. Это называется предварительным объявлением, и в итоге получается *неполный тип*, из которого нельзя создать объект.

Публичный интерфейс класса `Player` (`ExtremeC_examples_chapter7_2_player.h`)

```
#ifndef EXTREME_C_EXAMPLES_CHAPTER_7_2_PLAYER_H
#define EXTREME_C_EXAMPLES_CHAPTER_7_2_PLAYER_H

// предварительное объявление типа
struct player_t;
struct gun_t;

// аллокатор памяти
struct player_t* player_new();

// конструктор
void player_ctor(struct player_t*, const char*);

// деструктор
void player_dtor(struct player_t*);

// поведенческие функции
void player_pickup_gun(struct player_t*, struct gun_t*);
void player_shoot(struct player_t*);
void player_drop_gun(struct player_t*);

#endif
```

Тип `gun_t` нужно объявить в связи с тем, что некоторые поведенческие функции класса `Player` принимают аргументы этого типа.

Определение класса `Player` (`ExtremeC_examples_chapter7_2_player.c`)

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

#include "ExtremeC_examples_chapter7_2_gun.h"

// структура атрибутов
typedef struct {
    char* name;
    struct gun_t* gun; // атрибут-указатель
} player_t;

// аллокатор памяти
player_t* player_new() {
    return (player_t*)malloc(sizeof(player_t));
}

// конструктор
void player_ctor(player_t* player, const char* name) {
    player->name = (char*)malloc((strlen(name) + 1) * sizeof(char));
    strcpy(player->name, name);
    // Это важно! Указатели агрегации, которые не должны быть заданы в конструкторе, необходимо о
    бнулить
    player->gun = NULL;
}

// деструктор
void player_dtor(player_t* player) {
    free(player->name);
}

// поведенческие функции
void player_pickup_gun(player_t* player, struct gun_t* gun) {
    // после следующей строчки начинается отношение типа <<агрегация>>
    player->gun = gun;
}

void player_shoot(player_t* player) {
    // нам нужно проверить, подобрал ли игрок ружье,
    // иначе стрельба не имеет смысла
    if (player->gun) {
        gun_trigger(player->gun);
    }
    else {
        printf("Player wants to shoot but he doesn't have a gun!");
        exit(1);
    }
}

void player_drop_gun(player_t* player) {
    // После следующей строчки завершается агрегация двух объектов.
    // Обратите внимание, что объект gun не нужно освобождать, поскольку данный объект не является его вл
    адедцем (как в композиции)/
    player->gun = NULL;
}
```

Внутри структуры `player_t` объявлен атрибут-указатель `gun`, который скоро будет ссылаться на одноименный объект. В конструкторе его нужно *обнулить*, а не присваивать ему значение, как в случае с композицией.

Следует отметить, что после разрыва отношения типа «агрегация» указатель `gun` следует обнулить. В отличие от композиции, контейнер здесь не является *владельцем* внутреннего объекта, поэтому не может контролировать его жизненный цикл.

Реализация класса `Gun`

Определение класса `Gun` (`ExtremeC_examples_chapter7_2_gun.c`)

```
#include <stdlib.h>

typedef int bool_t;

// структура атрибутов
typedef struct {
    int bullets;
} gun_t;

// аллокатор памяти
gun_t* gun_new() {
    return (gun_t*)malloc(sizeof(gun_t));
}

// конструктор
void gun_ctor(gun_t* gun, int initial_bullets) {
    gun->bullets = 0;
    if (initial_bullets > 0) {
        gun->bullets = initial_bullets;
    }
}

// деструктор
void gun_dtor(gun_t* gun) {
    // здесь ничего не происходит
}

// поведенческие функции
bool_t gun_has_bullets(gun_t* gun) {
    return (gun->bullets > 0);
}

void gun_trigger(gun_t* gun) {
    gun->bullets--;
}

void gun_refill(gun_t* gun) {
    gun->bullets = 7;
}
```

Точка входа

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

#include "ExtremeC_examples_chapter7_2_player.h"
#include "ExtremeC_examples_chapter7_2_gun.h"

int main(int argc, char** argv) {
```

```

// создаем и инициализируем объект gun
struct gun_t* gun = gun_new();
gun_ctor(gun, 3);

// создаем и инициализируем объект player
struct player_t* player = player_new();
player_ctor(player, "Billy");

// начинаем агрегацию
player_pickup_gun(player, gun);

// стреляем, пока не кончатся патроны
while (gun_has_bullets(gun)) {
    player_shoot(player);
}

// перезаряжаем ружье
gun_refill(gun);

// стреляем, пока не закончатся патроны
while (gun_has_bullets(gun)) {
    player_shoot(player);
}

// завершаем агрегацию
player_drop_gun(player);

// уничтожаем и освобождаем объект player
player_dtor(player);
free(player);

// уничтожаем и освобождаем объект gun
gun_dtor(gun);
free(gun);
}

```

Важной особенностью агрегации является то, что **контейнер не должен влиять на время жизни внутреннего объекта**; если следовать данному правилу, то никаких проблем с памятью возникнуть не должно [3, стр. 258].

Сборка примера

```

$ gcc -c gun.c -o gun.o
$ gcc -c player.c -o player.o
$ gcc -c main.c -o main.o
$ gcc gun.o player.o main.o -o ex.out

```

В объектной модели реального проекта *агрегация* обычно применяется чаще, чем композиция. В отличие от композиции, агрегация является временным отношением между двумя объектами.

14. Наследование и полиморфизм

14.1. Наследование

Пример расширения через структуры атрибутов

```

typedef struct {
    char first_name[32];
    char last_name[32];
}

```

```

    unsigned int birth_year;
} person_t;

typedef struct {
    person_t person;
    char student_number[16];
    unsigned int passed_credits;
} student_t;

```

В показанном выше листенге наличие первого поля типа `person_t` позволяет привести указатель `student_t` к этому типу и сделать так, чтобы они *указывали* на один и тот же *адрес в памяти*.

Это так называемое *восходящее приведение* – то есть приведение структуры атрибутов потомка к типу структуры атрибутов родителя.

Указатели указывают на адрес в памяти. Указатели ссылаются на участок памяти [3, стр. 263].

```

#include <stdio.h>

typedef struct {
    char first_name[32];
    char last_name[32];
    unsigned int birth_year;
} person_t;

typedef struct {
    person_t person;
    char student_number[16]; // дополнительный атрибут
    unsigned int passed_credits; // дополнительный атрибут
} student_t;

int main(int argc, char** argv) {
    student_t s; // объявляем структурную переменную
    student_t* s_ptr = &s;
    person_t* p_ptr = (person_t*)&s;
}

```

Указатели `s_ptr` и `p_ptr` *указывают* на один и тот же *адрес*. Это значит, что структурная переменная типа `student_t` в действительности наследует структуру `person_t` в своей схеме размещения. Отсюда следует, что поведенческие функции класса `Person` можно вызывать с помощью указателя на объект `student`. Иными словами, поведенческие функции класса `Person` могут использоваться объектами `student`.

Неполный тип – это результат предварительного объявления структуры. Его могут иметь *только указатели*, но не переменные. *Память для неполного типа нельзя выделить даже в куче* [3, стр. 263].

Наследование можно реализовать двумя путями:

- сделать так, чтобы дочерний класс имел доступ к приватной реализации (определению) базового класса,
- сделать так, чтобы дочерний класс мог обращаться только к публичному интерфейсу базового класса.

Если требуется, чтобы одна часть кода знала об определении структуры, а другая – нет, то можно воспользоваться *приватными заголовочными файлами*. Приватным называют заголовочный файл, который должен подключаться и использоваться только на определенных участках кода или в классах, которым он действительно нужен.

```

#ifndef EXTREME_C_EXAMPLES_CHAPTER_8_2_PERSON_P_H
#define EXTREME_C_EXAMPLES_CHAPTER_8_2_PERSON_P_H

// приватное определение
typedef struct {
    char first_name[32];
    char last_name[32];
    unsigned int birth_year;
} person_t;

#endif

```

Это часть класса **Person**, которая должна оставаться приватной, но притом быть доступной классу **Student**.

NB: *неполный тип* как результат предварительного объявления могут иметь только указатели, но не переменные! [3, стр. 267]. Причем в расширяющей (дочерней) структуре атрибутов эта структурная переменная должна быть первым полем, иначе мы теряем возможность использования поведенческих функций базового класса [3, стр. 268].

При использовании первого подхода к наследованию мы хранили структурную переменную в качестве первого поля структуры атрибутов потомка. Во втором подходе будем использовать указатель на структурную переменную родителя. Это позволит сделать дочерний класс независимым от реализации родительского [3, стр. 270].

Основное отличие второго подхода в том, что класс **Student** зависит только от публичного интерфейса класса **Person** и не требует доступа к его приватному определению. Это важно, поскольку так мы можем изолировать классы друг от друга и легко изменять реализацию родителя, не меняя реализацию потомка.

Приватное определение **person_t** теперь находится в исходном файле, и мы больше не используем приватный заголовок. Это значит, что мы не станем делать его доступным ни для каких других классов, включая **Student**. Мы хотим добиться полноценной инкапсуляции класса **Person** и скрыть все детали его реализации.

Два подхода к реализации наследования в Си [3, стр. 276]:

- Оба подхода, по сути, демонстрируют отношение типа «композиция»,
- В первом подходе структурная переменная находится в структуре атрибутов потомка. Здесь требуется доступ к приватной реализации родительского класса. Но во втором подходе используется структурный указатель неполного типа, принадлежащего структуре атрибутов родителя, поэтому теряется зависимость от приватной реализации родительского класса.
- В первом подходе родительский и дочерний типы тесно связаны. Во втором классы независимы друг от друга и все, что находится в родительской реализации, скрыто от потомка.
- В первом подходе может быть только один родитель. То есть это реализация *одиночного наследования* в Си. Во втором же подходе родителей может быть сколько угодно; так выглядит концепция *множественного наследования*.
- В первом подходе структурная переменная родителя должна быть первым полем, структуры атрибутов дочернего класса. Во втором указатели на родительские объекты могут находиться в любом месте структуры.

14.2. Полиморфизм

Полиморфизм это метод использования одного и того же кода для реализации разного поведения. С его помощью можно расширить код и добавлять новые возможности, не прибегая к перекомпиляции всей кодовой базы.

Полиморфизм – просто предоставление разного поведения с помощью одного и того же публичного интерфейса (или набора поведенческих функций) [3, стр. 277].

Пример. Пусть есть два класса `Cat`, `Duck`, каждый из которых имеет поведенческую функцию `sound` для вывода издаваемых ими звуков. Мы исходим из того, что `Cat` и `Duck` – потомки класса `Animal`.

Без полиморфизма операцию `sound` пришлось бы вызывать из каждого объекта

```
// без полиморфизма  
animal_sound(animal);  
cat_sound(cat);  
duck_sound(duck);
```

На отсутствие полиморфизма в приведенных выше листингах указывает тот факт, что для вызова определенных операций из объектов `Cat` и `Duck` использовались разные функции: `cat_sound` и `duck_sound`.

Пример полиморфного кода

```
// это полиморфизм  
animal_sound(animal);  
animal_sound((struct animal_t*)cat);  
animal_sound((struct animal_t*)duck);
```

В полиморфном коде подразумевается наличие отношения наследования между классом `Animal` и двумя другими классами, `Cat` и `Duck`, поскольку мы должны иметь возможность приводить указатели `duck_t` и `cat_t` к типу `animal_t`. Структурная переменная `animal_t` должна идти первым полем в структурах атрибутов `duck_t` и `cat_t`.

Определение структур атрибутов для классов `Animal`, `Cat` и `Duck`

```
typedef struct {  
    ...  
} animal_t;  
  
typedef struct {  
    animal_t animal;  
    ...  
} cat_t;  
  
typedef struct {  
    animal_t animal;  
    ...  
} duct_t;
```

Благодаря такой конфигурации мы можем приводить указатели `duck_t` и `cat_t` к типу `animal_t` и затем использовать одни и те же поведенческие функции для обоих дочерних классов.

Полиморфизм прежде всего нужен потому, что мы хотим использовать *один и тот же код при работе с разными подтипами базового типа* [3, стр. 280].

Нам не хочется модифицировать нашу логику при добавлении в систему новых подтипов или когда один из подтипов меняет свое поведение. Конечно, при появлении новой возможности нельзя совсем обойтись без обновления кода – какие-то изменения обязательно потребуются. Но

благодаря полиморфизму мы можем существенно уменьшить количество необходимых изменений.

Еще один повод для использования полиморфизма связан с понятием *абстракции*. *Абстрактные типы* (или *классы*) обычно содержат какие-то неопределенные или нереализованные поведенческие функции, которые необходимо переопределять в дочерних классах, и полиморфизм играет в этом ключевую роль.

14.3. Полиморфное поведение в языке Си

Приватный заголовок класса Animal

```
#ifndef EXTEME_C..._P_H
#define EXTEME_C..._P_H

// тип указателя, необходимого для обращения к разным версиям animal_sound
typedef void (*sound_func_t)(void*);

// предварительное объявление
typedef struct {
    char* name;
    // этот член класса является указателем на функцию,
    // которая отвечает за вывод звуков
    sound_func_t sound_func;
} animal_t;

#endif
```

При использовании полиморфизма каждый дочерний класс может предоставить собственную версию функции `animal_sound`. Иными словами, каждый подкласс может переопределить функцию, унаследованную от родительского.

14.4. Абстракция данных

Интерфейс – класс без каких-либо атрибутов или определений по умолчанию, содержащий только виртуальные функции.

Чтобы избежать создания объекта абстрактного типа, из публичного интерфейса класса можно убрать *функцию-аллокатор*. Если убрать аллокатор создание объектов из структуры атрибутов родителя становится доступным только дочерним классам.

Для получения *абстрактного класса* в Си необходимо *обнулить указатели на виртуальные функции*, у которых не должно быть определений по умолчанию на абстрактном уровне. Чтобы внешний код не мог создавать объекты абстрактных типов, следует удалить функцию-аллокатор из публичного интерфейса [3, стр. 293].

Пример наследования в Си

```
#include <string.h>

// родительская структура атрибутов
typedef struct {
    char c;
    char d;
} a_t;

// дочерняя структура атрибутов
typedef struct {
```

```

    a_t parent; // NB
    char str[5];
} b_t;

int main(int argc, char** argv) {
    b_t b;
    b.parent.c = 'A';
    b.parent.d = 'B';
    strcpy(b.str, "1234");

    return 0;
}

```

Демонстрация того, как следует приводить типы, чтобы указатели ссылались на правильные поля

```

typedef struct {
    ...
} a_t;
typedef struct {
    ...
} b_t;

typedef struct {
    a_t a;
    b_t b;
    ...
} c_t;

c_t c_obj;
a_t* a_ptr = (a_t*)&c_obj;
b_t* b_ptr = (b_t*)&c_obj + sizeof(a_t); // NB
c_t* c_ptr = &c_obj;

```

Мы прибавили размер объекта `a_t` к адресу `c_obj`; благодаря этому *указатель* теперь *ссылается* на поле `b` внутри `c_t`

14.5. Интерфейса командной оболочки для пользовательских приложений

Unix-совместимая система полностью соответствует стандартам SUS (Simple Unix Specification – простая спецификация Unix), чего нельзя сказать о *Unix-подобной системе*, которая имеет лишь частичную совместимость. Это означает, что *Unix-подобные системы* соответствуют только определенному *подмножеству стандартов SUS*. Следовательно, программы, разработанные для одной Unix-совместимой системы, теоретически можно перенести на другую, но перенос на Unix-подобную ОС не гарантирован. Это особенно касается программ, которые переносятся с Linux на другие Unix-совместимые системы или наоборот [3, стр. 321].

POSIX – подмножество SUS, с которым совместимы Unix-подобные системы [3, стр. 321].

Системные вызовы инициируются кодом, написанным в реализации `libc`. На самом деле именно так вызываются процедуры ядра. В SUS и впоследствии в POSIX-совместимых системах была предусмотрена программа, которая позволяла отслеживать системные вызовы во время выполнения кода.

Процесс ядра – это первое, что загружается и выполняется в системе. Только вслед за ним можно запускать пользовательские процессы. Процесс ядра обрабатывает и выполняет системные вызовы, в то время как пользовательский их инициирует и ждет выполнения.

Процесс ядра имеет *привилегированный* доступ к *физической памяти* и всему подключенному оборудованию, в то время как пользовательский работает с *виртуальной памятью*, которая является отражением части физической памяти, и ничего не знает о ее физической структуре. Пользовательский процесс имеет управляемый и наблюдаемый доступ к ресурсам и оборудованию. Можно сказать, что он выполняется в изолированной среде, которая симулируется операционной системой. Это также означает, что пользовательские процессы не могут видеть память друг друга [3, стр. 329].

Среда выполнения операционной системы имеет два разных режима. Один предназначен для процесса ядра, а второй – для пользовательских процессов.

Первый режим выполнения называется *пространством ядра*, а второй – *пользовательским пространством*. Оба пространства взаимодействуют с помощью предусмотренных системных вызовов. В целом причиной появления системных вызовов послужила необходимость в изоляции между пространствами ядра и пользователя. Пространство ядра обладает максимально привилегированным доступом к системным ресурсам, а доступ пользовательского пространства максимально ограниченный и контролируемый.

NB: никакой пользовательский процесс не должен иметь прямого доступа к оборудованию, а также к внутренним структурам данных и алгоритмам ядра. Обращение к ним должно происходить через системные вызовы [3, стр. 336].

Пока системный вызов делает свою работу, пользовательский процесс обычно находится в *ожидании*. В таком случае *системный вызов* называют *блокирующим*.

Если системному вызову этого мало, то мы можем передать ему указатели на структурные переменные, выделенные в памяти пользовательского процесса.

15. Нововведения в Си

Стандарт C11 пришел на смену C99 и позже был заменен на C18. В C18 не появилось никаких новых возможностей; эта версия содержит лишь исправления ошибок, найденных в C11 [3, стр. 365].

Открытые компиляторы, такие как `gcc` и `clang`, имеют полную поддержку C11, но при необходимости могут переключаться на C99 и даже более ранние версии Си.

Каждый стандарт Си определяет специальный макрос, позволяющий сделать это

```
#include <stdio.h>

int main(int argc, char** argv) {
    printf("VERSION: %ld", __STDC_VERSION__);
}
```

15.1. Удаление функции `gets`

Из C11 была убрана знаменитая функция `gets`. Она была подвержена атакам с переполнением буфера, и в предыдущих версиях ее решили сделать *нерекомендуемой*. Вместо `gets` можно использовать `fgets`.

Функция `gets` не подходит для безопасного использования. Ввиду отсутствия проверки диапазона и неспособности вызывающей программы надежно определить длину следующей входной строки.

15.2. Функции с проверкой диапазона

Программы на языке Си, которые работают с массивами строк и байтов, присуща одна серьезная проблема: они могут легко *выйти за пределы диапазона*, определенного для буфера или байтового массива [3, стр. 370].

Буфер – область памяти, которая служит для хранения массива байтов или строковой переменной. Выход за границы приводит к *переполнению буфера*, чем могут воспользоваться злоумышленники, чтобы организовать атаку (которую обычно называют *атакой переполнения буфера*). В число уязвимых попадают функции обработки строк наподобие `strcpy` и `strcat`, находящиеся в `string.h`. У них нет механизмов проверки границ, которые могли бы предотвратить атаки переполнения буфера.

Среди функций с проверкой диапазона, появившихся в C11, можно выделить `strcpy_s` и `strcat_s`. Эти функции предотвращают запись в невыделенную память.

15.3. Невозвращаемые функции

Вызов функции можно завершить либо с помощью ключевого слова `return`, либо при достижении конца ее блока. Бывают также ситуации, в которых вызов функции никогда не завершается, и обычно это делается намеренно.

В C11 можно указать, что функция является *невозвращаемой* и никогда не прекращает работу. Для этого можно использовать ключевое слово `_Noreturn` из заголовочного файла `stdnoreturn.h`. Ситуация, когда функция, помеченная как `_Noreturn`, возвращается, является примером неопределенного поведения и ни в коем случае не приветствуется.

15.4. Макросы для обобщенных типов

В C11 появилось новое ключевое слово: `_Generic`. С его помощью можно писать макросы, которые получают информацию о типе на этапе выполнения. Иными словами, вы можете написать макрос, который меняет свое значение в зависимости от типов своих аргументов.

Пример

```
#include <stdio.h>
#define abs(x) _Generic((x), \
                        int: absi, \
                        double: absd)(x)

int absi(int a) {
    return a > 0 ? a : -a;
}

double absd(double a) {
    return a > 0 ? a : -a;
}

int main(int argc, char** argv) {
    printf("abs(-2): %d\n", abs(-2));
    printf("abs(2.5): %f\n", abs(2.5));
    return 0;
}
```

В определении макроса есть два разных выражения, которые выбираются в зависимости от типа аргумента `x`. Для значений `int` используется `absi`, а для значений `double` – `absd`.

15.5. Unicode

Одним из самых важных нововведений в стандарте C11 стала поддержка Unicode (в виде кодировок UTF-8, UTF-16 и UTF-32). До выхода C11 нам были доступны только восьмибитные типа `char` и `unsigned char`, предназначенные для хранения символов кодировок ASCII и Extended ASCII. Строки имели вид массивов этих ASCII-символов.

15.6. Анонимные структуры и анонимные объединения

Анонимные структуры и анонимные объединения – определения типов *без имени*; обычно они используются в качестве *вложенных типов*.

```
typedef struct {
    union { // анонимное объединение
        struct { // анонимная структура
            int x;
            int y;
        };
        int data[2]; // двухэлементный целочисленный массив
    };
} point_t;

int main(int argc, char** argv) {
    point_t p;
    p.x = 10; // то же что и p.data[0] = 10;
    p.data[1] = -5; // то же что и p.y = -5;
    printf("Point (%d, %d) using anonymous structure.\n", p.x, p.y);
    printf("Point (%d, %d) using anonymous byte array.\n", p.data[0], p.data[1]);
}
```

Одна и та же область памяти используется для хранения экземпляра анонимной структуры и двухэлементного целочисленного массива [3, стр. 379].

16. Конкурентность

Параллелизм означает, что два задания выполняются одновременно, или в параллель. Именно словосочетание «в параллель» – ключевое отличие между параллелизмом и конкурентностью. Потому что параллельность подразумевает протекание двух событий в один и тот же момент. В случае с *конкурентной* системой это не так; прежде чем выполнять другое задание, она *должна остановить текущее*.

Для параллельного выполнения двух задач компьютерной системе нужно по меньшей мере два *отдельных и независимых вычислительных блока*. Современные процессоры содержат внутри несколько *ядер*, которые являются этими самыми блоками. Например, четырехядерный процессор имеет четыре вычислительных блока; следовательно, может выполнять сразу четыре параллельных задания.

Даже имея многоядерный процессор, вы все равно должны назначать *каждый поток выполнения* определенному ядру. И если одно ядро получит несколько потоков, то их *нельзя будет выполнить параллельно*. В результате вы сможете наблюдать конкурентное выполнение [3, стр. 384].

Если коротко, то наличие двух потоков, назначенных двум разным ядрам, несомненно, может привести к созданию *двух параллельных потоков*. Но если назначить их одному ядру, то

получатся два конкурентных потока. В многоядерном процессоре мы фактически наблюдаем смешанное поведение: как параллелизм между ядрами, так и конкурентность в рамках одного ядра [3, стр. 384].

Если планировщик заданий (он просто очень быстро переключается между заданиями, выполняя некую крошечную часть каждого из них за довольно короткий промежуток времени) достаточно быстрый и беспристрастный, то вы не заметите переключения между заданиями; вам будет казаться, что они работают параллельно.

Конкурентность можно считать *имитацией параллельного выполнения заданий на одном вычислительном блоке (ядре)* [3, стр. 385].

16.1. Процессы и потоки

В ОС роль заданий играют либо *процессы*, либо *потоки*. Большинство ОС обращается с ними практически одинаковым образом: как с некими заданиями, которые необходимо выполнять конкурентно.

ОС должна использовать планировщик в целях разделения ядер процессора между множеством *заданий*, будь то *процессы* или *потоки*, которым для выполнения нужно процессорное время. В момент создания новый процесс или поток поступает в очередь планировщика в качестве нового задания и, прежде чем начинать работу, ждет, когда ему выделят *процессорное ядро*.

Если у вас установлен вытесняющий планировщик (с разделением времени) и задание не успевает закончить работу в отведенное ему время, то он принудительно освобождает ядро процессора, а задание опять попадает в очередь.

Ситуация, когда вытесняющий планировщик останавливает процесс посреди выполнения и запускает вместо него другой, называется *переключением контекста*. Переключение контекста невозможно предсказать!

Если подытожить, то в конкурентной системе с несколькими заданиями, каждое из которых может читать и записывать разделяемый ресурс, разные прогоны будут давать разные результаты.

16.2. Синхронизация

Важное свойство состояния гонки: их возникновение в конкурентной системе не требует разделяемого состояния. Чтобы их избежать, некоторые инструкции должны всегда выполняться в строго определенном порядке. Следует отметить: состояние гонки встречается только когда небольшая группа инструкций, известная под названием «*критический участок*», выполняется не по порядку, в то время как другие инструкции могут выполняться в любом порядке [3, стр. 412].

Если есть изменяемое разделяемое состояние с инвариантным ограничением, то операции чтения и записи в отношении данного состояния, возможно, придется выполнять в строгом порядке. Это просто означает, что у всех заданий всегда должна быть возможность прочитать последнее, самое свежее значение разделяемого состояния.

Исходная конкурентная система носила естественный характер, и *планировщик заданий* был единственным ее компонентом, который инициировал *переключение контекста* [3, стр. 418].

16.3. Семафоры и мьютексы

В любой момент времени семафор позволяет находиться на защищаемом им критическом участке только одному заданию. Задания, ожидающие входа в критический участок, обычно «засыпают».

Семафоры могут позволить находиться на критических участках сразу нескольким заданиям. Семафор, который разрешает входить на критический участок по одному, обычно называют *двоичным семафором* или *мьютексом* [3, стр. 427].

Мьютексы куда более распространены, чем семафоры общего вида, и их всегда можно встретить в конкурентном коде.

Термин мьютекс означает *mutual exclusion* (взаимное исключение). В любой момент времени на критическом участке может находиться только одно из них, а другое должно ждать, пока критический участок не освободится [3, стр. 427].

Когда два разных задания выполняются на двух разных ядрах процессора и обращаются к одному и тому же адресу в основной памяти, каждое ядро заносит значение данного адреса в свой *локальный кэш*. Это значит, если одно из заданий попытается выполнить запись в разделяемый адрес, то изменения проявятся только в его локальном кэше, а не в основной памяти или в кэше других ядер процессора.

Это приводит к множеству разных проблем, и вот почему: когда задание, работающее на другом ядре, попытается прочитать значение из разделяемого адреса памяти, оно не сможет увидеть последние изменения, поскольку чтение будет выполнено из его локального кэша.

Чтобы решить проблему, которая возникает из-за наличия разных локальных кэшей в каждом ядре процессора, используется *протокол когерентности памяти*. В соответствии с ним все задания, работающие на разных ядрах, видят в своих локальных кэшах, одно и то же значение, даже если одно из ядер изменило его. То есть можно сказать, что адрес памяти виден всем вычислительным блокам. Соблюдение протокола когерентности памяти делает память *видимой* для всех заданий, выполняющихся в разных вычислительных блоках [3, стр. 430].

Барьеры памяти синхронизируют локальные кэши всех ядер процессора и основную память. Значения, измененные в одном локальном кэше, распространяются по основной памяти и других локальных кэшах. Они становятся видимыми для всех заданий, выполняющихся на других процессорных ядрах.

Стоит отметить, что создание задания, а также блокировка и разблокировка *семафора* – три разные операции, которые играют *роль барьеров памяти* и синхронизируют локальные кэши всех процессорных ядер и основную память, распространяя последние изменения, внесенные в разделяемое состояние [3, стр. 432].

Мьютексы – семафоры, которые позволяют находиться на критическом участке только одному заданию, и, как с любым семафором, их *блокировка* и *разблокировка* может служить *барьером памяти*.

16.4. Условные переменные

Условными называются обычные переменные (или объекты), которые позволяют перевести задание в спящий режим или оповестить другие задания, чтобы те «проснулись». По аналогии с мьютексами, которые защищают критический участок, условные переменные используются для организации *обмена сигналами* разными заданиями.

Опять же, как и мьютексы, у которых есть операции *блокировки* и *разблокировки*, условные переменные умеют *уведомлять* задания и переводить их в *спящий режим*.

Условную переменную необходимо использовать в связке с мьютексом. Помните, что условная переменная должна разделяться между несколькими заданиями, как разделяемый ресурс, иначе будет бесполезной. Вот почему доступ к ней нужно синхронизировать. Это зачастую достигается за счет применения мьютекса, который защищает критические участки.

При каждом запуске программы создается новый процесс, в котором выполняется ее логика. Процессы изолированы друг от друга; то есть один не имеет доступа к содержимому (например, к памяти) другого.

Похожим образом работают и потоки выполнения, но они находятся в рамках определенного процесса. Отдельно взятый поток нельзя разделить между двумя процессами. Все *потоки* процесса-владельца имеют доступ к его памяти, работая с ней как с разделяемым ресурсом. В то же время поток имеет собственный стек, доступный для других потоков того же процесса. Кроме того, и процессы, и потоки могут использовать разделяемые ресурсы центрального процессора, и планировщик заданий в большинстве операционных систем задействует один и тот же алгоритм для распределения между ними процессорных ядер.

В разных ядрах планировщики используют разные стратегии и алгоритмы для управления заданиями. Но большинство из них можно отнести к двум основным категориям:

- совместное планирование,
- вытесняющее планирование.

Совместное (или кооперативное) планирование состоит в том, что задание, которому выделили ядро процессора, должно добровольно его освободить. Этот подход не является вытесняющим в том смысле, что в большинстве штатных ситуаций не применяется никаких мер по принудительному освобождению процессорного ядра от задания.

Вытесняющее планирование – противоположность совместного. Оно позволяет заданию занимать ядро процессора до тех пор, пока его не освободит планировщик. В некоторых разновидностях вытесняющего планирования заданию позволено использовать ресурсы процессора на протяжении определенного времени. Такой подход называется *разделением времени* и является самой популярной стратегией планирования в современных операционных системах. Промежуток времени, на протяжении которого задание может занимать процессор, называют: интервал, период или квант.

Когда *процессы* или *потоки* не имеют разделяемого состояния, между многопроцессностью и многопоточностью нет принципиальной разницы. И можно одни заменить на другие. Но при наличии разделяемого состояния разница между процессами, потоками и даже их сочетанием становится огромной.

Все потоки внутри одного процесса имеют доступ к общей памяти, поэтому могут применять ее для хранения разделяемого состояния [3, стр. 441].

Потоки могут существовать *только внутри процессов*; не бывает такого потока, у которого не было бы процесса-владельца. Каждый процесс содержит по меньшей мере один поток, обычно называемый *главным* или *основным*.

17. Многопоточное выполнение

17.1. Потоки

Инициатор и постоянный владелец любого потока – процесс. Невозможно создать разделяемый поток или передать владение им другому процессу. У каждого процесса есть по меньшей мере один поток, который называют главным или основным. В программе на языке Си в рамках *главного потока* выполняется функция `main`.

Все потоки имеют общий идентификатор процесса (PID). У каждого потока есть уникальный идентификатор (TID). У каждого потока есть собственная выделенная ему маска сигналов, позволяющая фильтровать сигналы, которые он будет получать.

Ниже перечислены методы, с помощью которых потоки могут разделять или передавать состояние в POSIX-совместимой системе [3, стр. 448]:

- память процесса-владельца (сегменты данных, стека и кучи). Этот метод применим *только* к потокам, но не к процессам,
- файловая система,
- отображение файлов в память,
- сеть (с использованием сетевых сокетов),
- передача сигналов между потоками,
- разделяемая память,
- POSIX-каналы,
- сокеты домена Unix,
- очереди сообщений POSIX,
- переменные среды.

Время жизни потока зависит от времени жизни владеющего им процесса. Когда процесс принудительно завершается, вместе с ним удаляются все его потоки. Когда заканчивает работу главный поток, процесс немедленно завершается.

17.2. POSIX-потоки

Библиотека `pthread` – всего лишь набор заголовков и функций, пригодных для написания многопоточных программ в POSIX-совместимых операционных системах. У каждой ОС есть своя реализация этой библиотеки, которая может отличаться от любых других, но в целом все реализации предоставляют доступ к одному и тому же интерфейсу.

В качестве общеизвестного примера можно привести библиотеку потоков POSIX (Native POSIX Threading Library, NPTL) – главную реализацию `pthread` для ОС Linux.

Согласно API `pthread`, чтобы получить доступ ко всем функциям для работы с потоками, достаточно подключить заголовок `pthread.h`. У этой библиотеки также есть некоторые расширения, доступные только при подключении `semaphore.h`. Например, одно из расширений предназначено сугубо для операций с семафорами, такими как создание, инициализация, удаление и т.д.

17.3. Порождение POSIX-потоков

Первым делом нужно создать POSIX-поток.

```

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

// Стандартный заголовок POSIX-потока для использования библиотеки pthread
#include <pthread.h>

// Эта функция содержит логику, которая должна выполняться
// как тело отдельного потока
void* thread_body(void* arg) {
    printf("Hello from first thread!\n");
    return NULL;
}

int main(int argc, char** argv) {
    // Обработчик потоков
    pthread_t thread; // ссылка на поток

    // Создаем новый поток
    int result = pthread_create(&thread, NULL, thread_body, NULL);
    // Если создание потока завершилось неудачно
    if (result) {
        printf("Thread could not be created. Error number: %d\n", result);
        exit(1);
    }

    // Ждем, пока созданный поток не завершит работу
    result = pthread_join(thread, NULL);
    // Если присоединение потока оказалось неудачным
    if (result) {
        printf("The thread could not be joined. Error number: %d\n", result);
        exit(2);
    }
    return 0;
}

```

Вверху мы подключили новый заголовочный файл `pthread.h`. Это стандартный заголовок, который дает доступ ко всем возможностям библиотеки `pthread`. Он нужен, чтобы мы могли воспользоваться объявлениями функций `pthread_create` и `pthread_join`.

Функция `thread_body` принимает один указатель `void*` и возвращает другой `void*`. `void*` – обобщенный тип указателя, который может представлять указатели любого типа, такого как `int*` или `double*`.

Это самая общая сигнатура, которую может иметь функция в Си. Стандарт POSIX требует, чтобы ее соблюдали все функции, желающие быть компаньонами потока (то есть использоваться в качестве его логики). Вот почему мы объявили функцию `thread_body` таким образом.

Функция `main` – часть логики *главного потока*; она выполняется во время его создания.

Первой инструкцией в функции `main` служит объявление переменной типа `pthread_t`. Это *ссылка на поток*, и на момент объявления она не указывает ни на что конкретное. Иными словами, данная переменная еще не содержит ID никакого действующего потока. Только после успешного создания нового потока корректная ссылка на него будет присвоена этой переменной.

Если в качестве второго аргумента передать `NULL`, то новый поток будет использовать для своих атрибутов значения по умолчанию. Третьим аргументом, переданным в `pthread_create`, был указатель на *функцию-компаньон*, которая содержит его логику. В нашем коде логика по-

тока определена в функции `thread_body`, поэтому мы передали ее адрес, чтобы привязаться к переменной-ссылке `thread`.

Все функции в библиотеке `pthread`, включая `pthread_create`, в случае успешного выполнения должны возвращать ноль! Следовательно, любое ненулевое значение говорит о том, что функция отработала неудачно, и нам возвращается номер ошибки.

Создав новый поток, мы его *присоединяем* (`join`). Каждый процесс изначально имеет всего один *главный поток*, родителем которого выступает процесс-владелец. Сам главный поток играет роль родителя для всех остальных потоков. Обычно процесс завершается вместе с главным потоком, а вслед за ним немедленно прекращают работу все другие активные и спящие потоки.

Таким образом, если новый поток еще не начал выполняться (поскольку не получил доступа к процессору), а родительский процесс (вместе с главным потоком) тем временем завершился (неважно, по какой причине), он будет удален еще до выполнения своей первой инструкции. Поэтому *главный поток* должен *присоединить* второй поток, чтобы *подождать*, пока тот не завершит работу.

Поток становится завершенным только после возвращения функции-компаньона. В предыдущем примере порожденный поток завершается, когда функция-компаньон `thread_body` возвращает `NULL`. Затем *главный поток*, *заблокированный* вызовом `pthread_join`, освобождается и получает возможность продолжить работу, что в конечном счете приводит к успешному завершению программы.

NB: следует помнить, что для выполнения потока его недостаточно просто создать. До того как он получит доступ к ядру процессора и начнет работу, может пройти какое-то время. И если процесс успеет завершиться к этому моменту, то новый поток не получит возможности успешно выполниться [3, стр. 455].

Для компиляции многопоточного приложения необходимо добавить параметр `-lpthread`

```
$ gcc program.c -o ex.out -lpthread
$ ./ex.out
```

На некоторых платформах, таких как macOS, при компоновке программе можно обойтись и без параметра `-lpthread`. Но настоятельно рекомендуется указывать его при компоновке любого кода, использующего библиотеку `pthread`. Это необходимо для того, чтобы ваши сборочные скрипты работали на любой платформе и при сборке проектов на языке Си.

Поток, который можно присоединить, называют *соединяемым* (`joinable`). Все потоки таковы по умолчанию. Их противоположенность, *отсоединенные потоки*, невозможно присоединить.

Можно не присоединять порожденный поток к главному, а сделать его *отсоединенным*. Этим мы бы дали процессу понять, что, прежде чем завершаться, он должен подождать, пока не отработает отсоединенный поток. Обратите внимание: в таком случае процесс может продолжать работу после завершения *главного потока* [3, стр. 455].

Пример с отсоединенным потоком

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

void* thread_body(void* arg) {
    printf("Hello from fist thread\n");
    return NULL;
```

```

}

int main(int argc, char** argv) {
    pthread_t thread;

    // Создаем новый поток
    int result = pthread_create(&thread, NULL, thread_body, NULL);
    // Если создание потока завершилось неудачно
    if (result) {
        printf("Thread could not be created. Error number: %d\n", result);
        exit(1);
    }

    // Отсоединение потока
    result = pthread_detach(thread);
    // Если отсоединение потока оказалось неудачным
    if (result) {
        printf("Thread could not be detached. Error number: %d\n", result);
        exit(2);
    }

    // Выходим из главного потока
    pthread_exit(NULL);
    return 0;
}

```

Сразу после создания новый поток отсоединяется от главного потока. Вслед за этим главный поток завершает работу. Инструкция `pthread_exit(NULL)` нужна для того, чтобы процесс дождался завершения другого, отсоединенного потока. Не выполни мы разъединение, процесс завершился бы вместе с главным потоком.

Если у вас есть несколько значений, которые следует передать потоку в момент его создания, то вы можете разместить их в структуре и передать указатель на структурную переменную с нужными полями.

Все потоки процесса могут свободно работать с его сегментами стека, кучи, исполняемого кода и данных [3, стр. 458].

Многопоточную программу можно считать *потокобезопасной*, исключительно когда согласно имеющимся инвариантным ограничениям у нее *нет состояния гонки* [3, стр. 459].

Функция `printf` является потокобезопасной. То есть независимо от варианта чередований, когда один поток занимается выводом строки, экземпляры `printf` в других потоках не могут ничего вывести.

Два потока не могут иницировать один и тот же вызов функции. Причина этого очевидна: каждому вызову необходимо создать *стековый фрейм*, который должен быть помещен на вершину стека какого-то определенного потока, а два разных потока имеют два разных сегмента стека. Таким образом, отдельно взятый вызов функции может быть иницирован *только одним потоком*. Иными словами, *два потока* могут вызвать одну и ту же функцию по отдельности, и в результате получится два отдельных вызова, но они **не могут разделять один и тот же вызов** [3, стр. 460].

Следует отметить, что указатель, который передается потоку, не должен быть *висячим*, иначе могут возникнуть серьезные проблемы с памятью, которые будет сложно отследить.

Висячий указатель ссылается на адрес в памяти, по которому не выделена переменная. Если точнее, то в прошлом там могли находиться переменная или массив, но в момент использования указателя этот участок памяти уже освобожден [3, стр. 460].

18. Синхронизация потоков

Мьютексы, доступные в библиотеке pthread, можно применять для синхронизации как процессов, так и потоков. Мьютекс (двоичный семафор) – семафор, который позволяет заходить на критический участок только одному потоку за раз.

Мьютекс позволяет заходить на критический участок и выполнять там операции чтения и записи разделяемой переменной только одному потоку за раз. Таким образом, он гарантирует целостность разделяемой переменной.

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

// Стандартный заголовок POSIX для использования библиотеки pthread
#include <pthread.h>

// Объект мьютекса для синхронизации доступа к разделяемому состоянию
pthread_mutex_t mtx;

void* thread_body_1(void* arg) {
    // Получаем указатель на разделяемую переменную
    int* shared_var_ptr = (int*)arg;

    // Критический участок
    pthread_mutex_lock(&mtx);
    (*shared_var_ptr)++;
    printf("%d\n", *shared_var_ptr);
    pthread_mutex_unlock(&mtx);

    return NULL;
}

void* thread_body_2(void* arg) {
    int* shared_var_ptr = (int*)arg;

    // Критический участок
    pthread_mutex_lock(&mtx);
    *shared_var_ptr += 2;
    printf("%d\n", *shared_var_ptr);
    pthread_mutex_unlock(&mtx);

    return NULL;
}

int main(int argc, char** argv) {
    // Разделяемая переменная
    int shared_var = 0;
    pthread_t thread1;
    pthread_t thread2;

    // Инициализация мьютекса и его внутренние ресурсы
    pthread_mutex_init(&mtx, NULL);

    // Создаем новые потоки
```

```

int result1 = pthread_create(&thread1, NULL, thread_body_1, &shared_var);
int result2 = pthread_create(&thread2, NULL, thread_body_2, &shared_var);

if (result1 || result2) {
    printf("The threads could not be created.\n");
    exit(1);
}

// Ждем пока потоки не завершат работу
result1 = pthread_join(thread1, NULL);
result2 = pthread_join(thread2, NULL);

if (result1 || result2) {
    printf("The threads could not be joined.\n");
    exit(2);
}

pthread_mutex_destroy(&mtx);

return 0;
}

```

В обоих потоках мьютекс используется для защиты критических участков, размещенных между выражениями `pthread_mutex_lock(&mtx)` и `pthread_mutex_unlock(&mtx)`. Наконец, прежде чем покинуть функцию `main`, мы уничтожаем объект мьютекса.

Оба участка защищены мьютексом, и в любой момент времени на каждом из них может находиться только один поток, а остальные должны ждать снаружи, пока участок не будет освобожден [3, стр. 472].

Заходя на критический участок, поток блокирует мьютекс, и остальные потоки должны остановиться перед выражением `pthread_mutex_lock(&mtx)` и подождать пока мьютекс не будет разблокирован снова. Поток, ожидающий разблокирования мьютекса, по умолчанию входит в спящий режим.

Поток, который ждет условную переменную (пребывая в состоянии «сна»), рано или поздно получает уведомление и «просыпается». Более того, поток может уведомлять любые другие потоки, ожидающие (в состоянии сна) условную переменную. Все эти операции должны быть защищены мьютексом, и именно поэтому *мьютексы всегда необходимо применять в сочетании с условными переменными*.

Следует еще раз подчеркнуть, что *условную переменную* нужно использовать только на критических участках, защищенных сопутствующим *мьютексом* [3, стр. 476].

В большинстве случаев мьютексов (двоичных семафоров) достаточно для синхронизации разных потоков, обращающихся к разделяемому ресурсу. Дело в том, что для последовательного выполнения операций чтения и записи на критическом участке должен находиться только один поток.

Но иногда у вас может возникнуть необходимость в том, чтобы сразу несколько потоков могло работать с разделяемым ресурсом на критическом участке. В таких случаях следует использовать *семафоры общего вида*.

18.1. POSIX-потоки и память

У каждого потока есть *собственный сегмент стека*, который должен быть доступен только ему. Стек потока – часть общего стека процесса-владельца, в котором по умолчанию должны выделяться сегменты стека для всех потоков. Но стек может быть выделен и в куче.

Поскольку *все потоки внутри одного процесса могут читать и изменять* его сегмент стека, они фактически имеют доступ к стекам друг друга. Но не должны пользоваться этим доступом. Стоит отметить: работа со стеками других потоков считается опасным поведением, так как переменные, определенные на вершине стека, могут быть уничтожены в любой момент времени, особенно при завершении потока или возвращении функции [3, стр. 488].

Каждый сегмент стека доступен только его потоку-владельцу, но не другим потокам. Таким образом, *локальные переменные* (объявленные на вершине стека) считаются приватными ресурсами потока и другие потоки не должны к ним обращаться.

В однопоточных приложениях есть всего один, главный поток. Поэтому мы используем его сегмент стека так, будто это стек процесса. Причина в том, что в однопоточной программе между главным потоком и самим процессом нет никакого разделения. Но в многопоточных программах все иначе. У каждого потока есть свой стек, который отличается от стеков других потоков.

Сегменты кучи и данных доступны всем потокам, однако первый является динамическим и разделяется во время выполнения, тогда как второй генерируется на этапе компиляции. *Потоки* могут *читать и изменять* содержимое кучи. Куча – отличное место для хранения состояний, которые должны разделяться между разными потоками.

Самое простое и зачастую оптимальное место *для хранения разделяемых состояний* – *сегмент данных*, в котором операции выделения и освобождения выполняются автоматически. Переменные, находящиеся в этом сегменте, считаются *глобальными* и имеют максимально продолжительное время жизни – с момента создания процесса до самого его завершения.

Переменные, которые компилятор не должен оптимизировать с помощью кэширования, можно объявлять *изменчивыми*. Стоит отметить, что изменчивые переменные могут кэшироваться на уровне процессора, но компилятор не станет размещать их в своих кэшах

```
volatile int number;
```

Создание новых процессов требует *родительского процесса*. Вот почему у любого процесса есть родитель. На самом деле он может быть только один. Цепочка родителей и прародителей растягивается до самого первого пользовательского процесса, который обычно называется *init*. Родителем служит *процесс ядра* [3, стр. 504].

Когда процесс завершает работу, но у него остаются дочерние, *осиротевшие процессы*, его место в качестве родителя занимает *init*.

Новый процесс всегда является потомком какого-то другого процесса. При вызове функции `fork` создается точная копия вызываемого (родительского) процесса. В результате оба процесса продолжают работать конкурентно, начиная с инструкции, которая идет за вызовом `fork`. Дочерний процесс наследует много атрибутов от своего родителя, включая все сегменты памяти и их содержимое. Следовательно, он имеет доступ ко всем тем же переменным в сегментах данных и кучи, а также содержит те же программные инструкции в сегменте кода.

В момент вызова функции `fork()` из *вызывающего процесса*, который впоследствии становится родителем, создается (или клонируется) новый. После этого они продолжают работать конкурентно, как два разных процесса.

Родитель у любого отдельно взятого процесса может быть только один [3, стр. 506].

Процесс, потерявший своего родителя, становится *сиротой*, и `init` делает его своим прямым потомком.

В более новых версиях почти всех известных дистрибутивов Linux процесс `init` был заменен демоном `systemd`. При использовании функции `fork` родительский и дочерний процессы выполняются конкурентно.

Философия, стоящую за функциями `exec*`, можно описать так: вначале создается простой базовый процесс, и затем в какой-то момент в него загружается нужный нам исполняемый файл, который становится новым *образом процесса*. Таким образом, использование функции `exec*` вместо создания нового процесса вызывает подмену уже существующего. Это самое важное отличие от функции `fork`. Базовый процесс не копируется, а полностью заменяется новым набором сегментов памяти и программных инструкций [3, стр. 508].

В случае успешного выполнения функций `exec*` *предыдущий процесс исчезает*, а его место занимает новый. Таким образом, создание второго процесса не происходит.

Функции `exec*` можно применять для выполнения скриптов и внешних исполняемых файлов, а `fork` подходит только создания новых процессов, которые являются копией той же программы.

18.2. Методы разделения ресурсов

Состояние либо размещается в каком-то «месте», доступном ряду процессов, либо отправляется другим процессам в виде сообщения, сигнала или события. Для первого подхода требуется хранилище (или носитель), такое как буфер памяти или файловая система, а для второго – механизм обмена сообщениями, или канал между процессами.

В качестве примера первого подхода вполне уместе массив, который находится в разделяемой области памяти и может быть прочитан и изменен разными процессами. Что касается второго подхода, то это может быть компьютерная сеть, служащая каналом для передачи сообщений между процессами, размещенными на разных компьютерах в данной сети.

19. Синхронизация процессов

Синхронизировать процессы в распределенной системе нелегко.

19.1. Локальное управление конкурентностью

Довольно часто возникает ситуация, когда несколько процессов выполняются на одном компьютере и пытаются одновременно обратиться к разделяемому ресурсу. Поскольку все процессы работают в рамках одной операционной системы, они имеют доступ ко всем механизмам, которые она предоставляет.

Ниже перечислены *управляющие механизмы*, предусмотренные в стандарте POSIX, которые можно применять, когда все процессы выполняются на одном POSIX-совместимом компьютере [3, стр. 533]:

- Именованные POSIX-семафоры. У именованных POSIX-семафоров есть имя, благодаря чему их можно использовать глобально по всей системе. То есть это уже не *анонимные* или *приватные* семафоры.
- Именованные мьютексы. У них есть имена и потому их можно использовать по всей системе.

- Именованные условные переменные. Их необходимо размещать в объекте разделяемой памяти, чтобы они были доступны разным процессам.

Для работы с именованными семафорами не обязательно использовать функцию `fork`. Один и тот же семафор могут открыть совершенно разные процессы, которые не являются родителями и потомками, – они всего лишь должны выполняться на одном компьютере и в одной и той же операционной системе.

В Unix-подобных операционных системах существует два вида именованных семафоров:

- семафоры System V,
- POSIX-семафоры (они гораздо удобнее).

Именованные мьютексы – это обычные мьютексы, размещенные в *области разделяемой памяти*.

В многопоточных программах POSIX-мьютексы работают довольно просто. Однако в *многопроцессных средах* все немного иначе. Чтобы *мьютекс* можно было использовать в *разных процессах*, его необходимо определить там, где он будет им доступен.

Лучшее место подобного рода – *область разделяемой памяти*. Следовательно, чтобы получить мьютекс, который работает в многопроцессной среде, его нужно распределить именно в этой области.

Поскольку *каждый объект разделяемой памяти* имеет глобальное имя, *мьютекс*, размещаемый в этой области, можно считать *именованным*, и обращаться к нему могут другие процессы в системе.

20. Локальные сокеты и IPC

К методам IPC обычно относят любые средства взаимодействия и передачи данных между процессами.

Методы межпроцессного взаимодействия:

- разделяемая память,
- файловая система,
- POSIX-сигнал,
- POSIX-каналы,
- очереди сообщений POSIX,
- сокеты домена Unix,
- интернет-сокеты (или сетевые сокеты).

Каждое сообщение содержит последовательность байтов, которые собираются вместе в соответствии с четко определенным интерфейсом, протоколом или стандартом. Структура сообщения должна быть известна обоим процессам, работающим с ним, и обычно описывается в рамках *коммуникационного протокола*.

20.1. Коммуникационные протоколы

Процессы могут передавать только байты. То есть речь фактически о том, что *каждый фрагмент информации*, передаваемый с помощью любого метода IPC, должен быть предварительно переведен в *байтовую последовательность*. Это называется *сериализацией*.

С другой стороны, когда процесс получает последовательность байтов по каналу IPC, он должен уметь воссоздать из них исходный объект. Это называется *десериализацией*. Сообщения, отправляемые по IPC-каналам, могут иметь *текстовое*, *двоичное* или *гибридное* содержимое. Двоичные данные состоят из байтов со значениями в диапазоне от 0 до 255. Текстовые представляют собой символы, используемые в тексте. Иными словами, в текстовом содержимом допускается только алфавитно-цифровые и некоторые дополнительные символы.

Текстовые сообщения имеет смысл сжимать перед отправкой, а вот у двоичных сообщений плохой коэффициент сжатия. Одни протоколы – сугубо текстовые (например, JSON), а другие – полностью двоичные (такие как DNS). Но протоколы на подобие BSON и HTTP позволяют хранить в сообщениях сочетание из текстовых и двоичных данных.

Файловый дескриптор – абстрактная ссылка на локальный объект, который можно использовать для *чтения* и *записи данных*. Несмотря на свое название, файловые дескрипторы могут обозначать широкий спектр различных механизмов, предназначенных для чтения и изменения байтовых потоков.

Естественно, в число объектов, на которые могут ссылаться файловые дескрипторы, входят обычные файлы, размещенные в файловой системе (либо на жестком диске, либо в памяти).

Файловый дескриптор может представлять IPC-канал с возможностью чтения и записи. При создании POSIX-канала получается два файловых дескриптора: один для записи в канал, а второй для чтения из него.

Разные процессы могут взаимодействовать на одном компьютере – *Unix-сокеты*. *Сетевые сокеты* позволяют двум процессам, находящимся на разных компьютерах, общаться друг с другом по сети. Unix-сокеты *двунаправленные*.

20.2. Введение в программирование сокетов

20.2.1. Комьютерные сети

Компьютер в сети может называться по-разному: устройство, хост, узел или даже система. Первый шаг к созданию распределенной системы – наличие нескольких компьютеров, соединенных по сети – компьютерной сети.

Любое аппаратное оборудование, необходимое для физического подключения двух устройств, относится к физическому уровню. Это первый и самый низкий уровень. Без него невозможно было бы передавать данные между двумя компьютерами и считать их соединенными.

В рамках правил, соблюдение которых обеспечивается канальными протоколами, *сообщения* разбиваются на фрагменты, называемыми *кадрами*.

Один из самых известных канальных протоколов, предназначенный для проводного соединения компьютеров, – Ethernet. Он описывает все правила и нормы относительно передачи данных по компьютерным сетям. Еще один широко распространенный протокол, определяющий работу беспроводных сетей, называется IEEE 802.11.

Сеть, состоящая из компьютеров (или любых других вычислительных машин/устройств одного типа), соединенных физически с помощью определенного канального протокола, называется *локальной вычислительной сетью* (local area network, LAN).

У компьютера может быть несколько сетевых адаптеров, каждый из которых подключен к отдельной локальной сети. Таким образом, компьютер с тремя NIC способен работать с тремя локальными сетями одновременно.

Протоколы Ethernet и IEEE 802.11 назначают каждому совместимому сетевому адаптеру MAC-адрес (media access control – управление доступом к сети). Таким образом, адаптеру Ethernet или IEEE 802.11 Wi-Fi следует иметь уникальный MAC-адрес, иначе он не сможет подключиться к LAN. MAC-адреса не должны повторяться внутри одной локальной сети.

Для соединения разных узлов в локальных сетях Ethernet используются MAC-адреса. Но что, если соединить нужно компьютеры из двух разных LAN, которые, к слову, могут быть несовместимы между собой?

Для соединения узлов из разных LAN нужен *сетевой уровень*. Сетевой уровень работает с *пакетами* точно так же, как канальный с *кадрами*. Длинные сообщения разбиваются на более мелкие части, пакеты. Важно помнить, что пакеты инкапсулируются в кадрах. *Сетевой протокол* заполняет пробел между разными локальными сетями, соединяя их между собой.

IP (Internet Protocol) – самый известный сетевой протокол. У IP есть две версии с разной длиной адресов: IPv4 и IPv6.

Но как соединить два компьютера из двух разных LAN? Ответ кроется в механизме *маршрутизации*. Получение данных из внешней локальной сети требует наличия узла-маршрутизатора. Представьте, что мы хотим соединить две разные сети: LAN1 и LAN2. Маршрутизатор – обычный узел, который находится в обеих сетях благодаря наличию двух сетевых адаптеров. Один принадлежит LAN1, а другой – LAN2.

Затем специальный алгоритм маршрутизации определяет, какие пакеты следует передавать между сетями и как это делать. Благодаря механизмам маршрутизации через узел-маршрутизатор могут проходить данные из разных сетей и в разных направлениях.

В основе любого взаимодействия двух программ, находящихся на разных узлах, лежит трехуровневый стек из трех протоколов [3, стр. 596]:

- физического,
- канального,
- сетевого.

У каждого узла в IP-сети есть IP-адрес. Есть два вида IP-адресов: IPv4 и IPv6. В отдельно взятой локальной сети каждый узел имеет как *адрес канального уровня* (MAC-адрес), так и *IP-адрес*. Например, в сети Ethernet у узла есть MAC-адрес, который прокоды канального уровня задействуют для передачи данных внутри LAN, и IP-адрес, с помощью которого программы, размещенные на разных узлах, устанавливают сетевые соединения как в рамках одной локальной сети, так и с рядом других сетей [3, стр. 597].

Два хоста (узла) в одной или разных локальных сетях могут обмениваться данными или «видеть» друг друга, – утилита ping. Она отправляет ICMP-пакеты (Internet Control Message Protocol – протокол межсетевых управляющих сообщений); если они возвращаются обратно, то это значит, другой компьютер работает, подключен к сети и отвечает на запросы.

Два компьютера можно соединить с помощью стека из трех уровней: физического, канального и сетевого. Мы должны иметь возможность установить соединение между любыми двумя процессами, размещенными на разных компьютерах. Поэтому соединение, действующее лишь на сетевом уровне, будет слишком общим для поддержки взаимодействия нескольких отдельных процессов.

Вот почему поверх сетевого уровня необходим еще один – *транспортный уровень*. Если компьютеры общаются на сетевом уровне, то запущенные на них *процессы* могут соединяться на *транспортном уровне*, который функционирует поверх сетевого.

Любое сообщение при передаче по компьютерной сети всегда разбивается на мелкие фрагменты, которые называют пакетами. Если один из пакетов потеряется во время передачи, то операционная система получателя запросит его снова, чтобы восстановить все сообщение целиком. TCP (Transport Control Protocol) – это протокол транспортного уровня, который так себя ведет.

Взаимодействие можно проводить и без соединения. Наличие соединения гарантирует два фактора: *доставку* отдельных пакетов и их *упорядоченность*. Такие протоколы, как TCP, обеспечивают одновременное выполнение этих двух условий. А вот транспортные протоколы, которые не устанавливают соединение, их не гарантируют.

Например, UDP (User Datagram Protocol – протокол пользовательских датаграм) не гарантирует доставку пакетов и их упорядоченность.

Сетевая модель IPS (Internet Protocol Suite – набор интернет-протоколов) [3, стр. 604]:

- физический уровень;
- канальный уровень – Ethernet, IEEE 802.11 Wi-Fi;
- сетевой уровень – IPv4, IPv6 и ICMP;
- транспортный уровень – TCP, UDP;
- прикладной уровень – многочисленные протоколы наподобие HTTP, FTP, DNS, DHCP и многие другие.

Программирование сокетов – метод межпроцессного взаимодействия, который позволяет соединить два процесса, размещенных на одном или разных узлах с установленным между ними сетевым соединением. Транспортный уровень отвечает за соединение двух процессов поверх имеющегося сетевого уровня.

Интернет-соединение (сетевое соединение), в рамках которого создается транспортный канал, на самом деле соединяет операционные системы или, точнее, их ядра. Следовательно, в ядре должна существовать абстракция, напоминающая соединение. Роль этой абстракции играет *сокет*. Для любого соединения в системе, уже существующего или устанавливаемого, выделяется сокет, который его идентифицирует. Для отдельно взятого соединения между двумя процессами нужно по одному сокету на каждом конце. Один из этих сокетов принадлежит стороне соединителя, а другой – стороне слушателя.

У каждого объекта сокета есть три атрибута:

- домен,
- тип,
- протокол.

Соединения на основе сокетов являются двунаправленными и полнодуплексными. Это значит, обе стороны могут читать из канала и записывать в него, не мешая друг другу.

21. Программирование сокетов

Есть две категории межпроцессного взаимодействия, которые позволяют двум и более процессам общаться и обмениваться данными:

- активные (pull-based) методики требуют наличия доступного *носителя* (такого как разделяемая память или обычный файл) для хранения и извлечения данных,
- пассивные (push-based) методики, которые подразумевают создание *канала*, доступного всем процессам, участвующим во взаимодействии.

Говоря простым языком, при использовании *активных* методик данные должны извлекаться или считываться с *носителя*, а в *пассивном* подходе данные *доставляются* читающему процессу *автоматически*. В первом случае процессы извлекают данные из разделяемого носителя, и если несколько из них могут туда записывать, то это чревато *состоянием гонки*.

Программирование сокетов – особая разновидность межпроцессного взаимодействия. Сокеты – это специальные объекты в Unix-подобных и других ОС, включая даже Windows, представляющие *двухнаправленные каналы*.

Иными словами, один объект сокета можно использовать для чтения и записи в один и тот же канал. Таким образом, два процесса, находящиеся на разных концах одного канала, могут участвовать в *двухнаправленном взаимодействии*.

Дескриптор сокета всегда представляет канал, а вот *файловый дескриптор* может представлять такие носители, как обычный файл или POSIX-канал.

Взаимодействие, основанное на сокетах, может работать как с соединениями, так и без. В первом случае канал представляет собой *поток* байтов, передающийся между двумя определенными процессами, а во втором по каналу могут передаваться *датаграммы*, и при это никакого соединения между процессами нет.

Таким образом, мы имеем два типа каналов:

- потоковые,
- датаграммные

В целом *сокеты* можно разделить на две категории:

- UDS (Unix domain sockets, Unix-сокеты). Можно использовать в случаях, когда все процессы, желающие участвовать в межпроцессном взаимодействии, находятся *на одном компьютере*. Иными словами, UDS подходит только для проектов, развернутых в рамках одной системы.
- сетевые сокеты. Можно применять в почти любой конфигурации, независимо от того, как именно развернуты процессы и где они находятся. Они могут размещаться на одном компьютере или быть развернуты по сети.

В случае с локальным развертыванием более предпочтительны сокеты UDS, поскольку они более быстрые и имеют меньше накладных расходов по сравнению с сетевыми сокетами.

Мы имеем 4 разные комбинации:

1. Unix-сокеты поверх потокового канала,
2. Unix-сокеты поверх датаграммного канала,
3. сетевой сокет поверх потокового канала,
4. сетевой сокет поверх датаграммного канала.

Сетевой сокет, представляющий потоковый канал, обычно работает по TCP. Дело в том, что TCP – самый распространенный транспортный протокол для этого вида сокетов. С другой стороны, сетевой сокет, предоставляющий датаграммный канал, обычно работает по UDP.

Обратите внимание: сокеты UDS, предоставляющие потоковые или датаграммные каналы, не имеют каких-то специальных названий, поскольку не применяют никаких транспортных протоколов.

При установлении соединения между двумя процессами, размещенными на одном компьютере, одним из лучших решений являются сокеты домена Unix (Unix domain sockets, UDS).

Датаграммные каналы не поддерживают соединения и работают не так, как потоковые каналы. Иными словами, мы не можем установить выделенное соединение между двумя процессами. Поэтому процессы передают по каналу только отдельные фрагменты данных. Клиент отправляет

отдельные и независимые друг от друга датаграммы, а сервер их принимает и в свою очередь возвращает другие датаграммы в качестве ответа.

Таким образом, важнейшим аспектом датаграммного канала является то, что сообщение с запросом или ответом должно влезать в одну датаграмму. В противном случае его нельзя разделить между двумя датаграммами, и ни сервер, ни клиент не смогут его обработать.

22. Интеграция с другими языками

В качестве библиотек, которые загружаются в другие языки, могут выступать только разделяемые объектные файлы. Статическую библиотеку загрузить не получится, так как ее можно скомпоновать только с исполняемым или разделяемым объектным файлом. В Unix-подобных операционных системах разделяемые объектные файлы имеют расширение *.so, а в macOS – *.dylib.

23. Модульное тестирование и отладка

В целом мы тестируем отдельные аспекты системы. Они могут быть *функциональными* и *нефункциональными*.

Функциональное тестирование относится к определенным возможностям, входящим в список функциональных требований. Эти тесты предоставляют определенный ввод какому-то программному элементу, такому как функция, модуль, компонент или программная система, и ожидают получить от них определенный вывод.

Нефункциональное тестирование относится к *уровню качества*, на котором программный элемент, такой как функция, модуль, компонент или система в целом, выполняет определенные функции.

Эти тесты обычно измеряют различные показатели, например:

- потребление памяти,
- время выполнения,
- конфликты при блокировках,
- уровень безопасности и пр.

Тест считается пройденным, только если измеренный показатель находится в допустимом диапазоне. Ожидания относительно этих показателей выводятся из нефункциональных требований, предъявляемых к системе.

23.1. Уровни тестирования

В любой программной системе можно предусмотреть следующие уровни тестирования (это неполный список):

- модульное тестирование,
- интеграционное тестирование,
- системное тестирование,
- приемочное тестирование,
- регрессионное тестирование.

В модульном тестировании проверяется единица функциональности. Это может быть функция, выполняющая определенную работу, или набор функций, совместно удовлетворяющих некие потребности. Это может быть и класс с определенной конечной целью, и даже компонент, на который возложена конкретная задача.

Компонент – часть программной системы с четко определенным набором функций; компоненты, собранные воедино, составляют всю систему в целом.

Когда единицей функциональности выступает компонент, процесс тестирования называется *компонентным*.

Модули в совокупности формируют компонент. В компонентном тестировании проверяются отдельные изолированные компоненты. Но если их сгруппировать, то возникнет необходимость в другом уровне тестирования, который относится к их функциональности или характеристикам, – в *интеграционном тестировании*.

Тестирование всей системы в целом происходит на другом уровне. Здесь у нас есть набор всех компонентов, которые полностью интегрированы. Таким образом, мы проверяем, соответствуют ли возможности, предоставляемые системой, и ее характеристики заявленным требованиям.

Еще один уровень предусмотрен для проверки системы на соответствие *бизнес-требованиям* с точки зрения *заинтересованной стороны* или конечного пользователя. Это так называемое *приемочное тестирование*.

Как и *системное*, оно относится ко всей системе в целом, однако на самом деле оба уровня существенно отличаются:

- за системное тестирование отвечают разработчики и тестировщики, а приемочное обычно проводят конечные пользователи или заинтересованная сторона;
- приемочное тестирование охватывает только функциональные требования, а системное – еще и нефункциональные;
- в системном тестировании в качестве ввода обычно используется небольшой, заранее подготовленный набор данных, тогда как приемочное имеет дело с настоящими данными, которые поступают в систему в режиме реального времени.

23.2. Модульное тестирование

В рамках модульного тестирования проверяются изолированные модули любых размеров – от *функции* до *компонента*.

Самый важный аспект модульного тестирования состоит в том, что проверяемые модули должны быть *изолированы* друг от друга [3, стр. 696]. Например, если одна функция зависит от другой, то мы можем как-то протестировать их по отдельности.

Пример тестов

ExtremeC_examples_chapter22_1__next_even_number__tests.c

```
#include <assert.h>
#include "ExtremeC_examples_chapter22_1.h"

void TESTCASE_next_even_number__even_numbers_should_be_returned() {
    assert(next_even_number() == 0);
    assert(next_even_number() == 2);
    ...
}

void TESTCASE_next_even_number__numbers_should_rotate() {
```



```

    int64_t number = next_even_number();
    next_even_number();
    next_eve_number();
    ...
    int64_t number2 = next_even_number();
    assert(number == number2);
}

```

Следует отметить, что принцип, по которому выбираются имена для тестовых случаев, вырабатываются самостоятельно; никакого стандарта на сей счет не существует. Имя тестового случая должно подсказывать, что делает тот или иной тест.

Чтобы тестовые случаи можно было легко отличить от обычных функций, можно добавить к их именам префикс `TESTCASE` в верхнем регистре.

При успешном прохождении всех тестов возвращаемое значение должно быть равно 0.

Тесты чаще всего собираются в отладочном режиме, поскольку в случае провала одного из них нам сразу же нужна точная трассировка стека и дополнительная отладочная информация. Более того, инструкции `assert` обычно удаляются из сборки выпуска, но в исполняемом файле средства выполнения тестов должны присутствовать. Параметр `-g` добавляет отладочные символы в итоговый исполняемый файл.

```

$ gcc -g -c ExtremeC_examples_chapter22_1.c -o impl.o
$ gcc -g -c ExtremeC_examples_chapter22_1__next_even_number__tests.c -o tests1.o
$ gcc -g -c ExtremeC_examples_chapter22_1__calc_factorial__tests.c -o tests2.o
$ gcc -g -c ExtremeC_examples_chapter22_1_tests.c -o main.o
$ gcc impl.o tests1.o tests2.o main.o -o ex22_1_tests.out
$ ./ex22_1_tests.out

```

Функция-заглушка – очень простая функция, которая обычно возвращает фиксированное значение [3, стр. 705].

Фреймворки для модульного тестирования:

- Check,
- CMocka,
- Google Test etc.

Первое преимущество CMocka: этот фреймворк написан исключительно на Си и зависит только от стандартной библиотеки данного языка – никаких других библиотек он не использует. CMocka де-факто стандартный фреймворк для модульного тестирования в Си. Она поддерживает *средства тестирования* (test fixtures), которые позволяют инициализировать и очищать тестовую среду перед выполнением тестового случая и после него. Вдобавок CMocka поддерживает *макетирование функций*, что очень полезно при написании макета какой-либо функции на Си. Макетную функцию можно сконфигурировать так, чтобы она возвращала соответствующее значение для определенного ввода.

Пример на основе CMocka

```

// Нужно для библиотеки CMocka
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

#include "ExtremeC_examples_chapter22_1.h"

int64_t input_value = 1;

```

```

int64_t feed_stub() {
    return input_value;
}

void calc_factorial__fact_of_zero_is_one(void** state) {
    input_value = 0; // переопределяем
    int64_t fact = calc_factorial(feed_stub);
    assert_int_equal(fact, 1);
}

void calc_factorial__fact_of_negative_is_one(void** state) {
    input_value = -10; // переопределяем
    int64_t fact = calc_factorial(feed_stub);
    assert_int_equal(fact, 1);
}

void calc_factorial__fact_of_5_is_120(void** state) {
    input_value = 5;
    int64_t fact = calc_factorial(feed_stub);
    assert_int_equal(fact, 120);
}

...

// фикстура
int setup(void** state) {
    return 0;
}

// фикстура
int tear_down(void** state) {
    return 0;
}

int main(int argc, char* argv) {
    const struct CMUnitTest tests[] = {
        cmocka_unit_test(calc_factorial__fact_of_zero_is_one),
        cmocka_unit_test(calc_factorial__fact_of_negative_is_one),
        ...
    };
    return cmocka_run_group_tests(tests, setup, tear_down);
}

```

Здесь в локальной области видимости функций `calc*` переопределяется глобальная переменная `input_value`. В Python этот подход работать не будет, так как *функция привязывается к своей области определения*

```

import typing as t

a = -1

def f() -> int:
    return a

def g(ff: t.Callable[[], int]) -> int:
    a = 100 # не оказывает никакого влияния!!!
    return ff() # функция f() будет искать значение 'a' в своей области определения (в глобальной области видимости)

```

```
if __name__ == "__main__":
    print(f()) # -1
    print(g(f)) # -1
```

При вызове функции `g(f)`, вызывается функция `f()`, но искать значение переменной `a` эта функция будет в области видимости своего определения, то есть в данном случае в глобальной области видимости (на уровне модуля) и потому вернет `-1`.

В Си аналогичный код будет работать так

```
#include <stdio.h>

typedef int (*f_t f)(); // указатель на функцию

int a = -1;

int f() {
    return a;
}

int g(f_t ff) {
    a = 100; // переопределяет глобальную переменную
    return ff();
}

int main(int argc, char** argv) {
    printf("%d\n", f()); // -1
    printf("%d\n", g(f)); // 100
}
```

В CMocka каждый тестовый случай должен возвращать `void` и принимать аргумент `void**`, Аргумент-указатель будет использоваться для получения информации, `state`, которая относится к отдельно взятому тестовому случаю. В функции `main` мы создаем список тестовых случаев и в конце вызываем `smocka_run_group_tests`, чтобы выполнить все модульные тесты.

Функции `setup` и `tear_down` это средства тестирования. Они вызываются перед выполнением каждого тестового случая и после него и предназначены для его подготовки и уничтожения. Первое вызывается перед тестовым случаем, а второе – после него. Эти функции можно было назвать как угодно, но для ясности были выбраны имена `setup` и `tear_down`.

Чтобы скомпилировать программу, нужно сначала установить CMocka. В системах Linux на основе Debian для этого достаточно выполнить `sudo apt-get install libsmocka-dev`, а в macOS можно воспользоваться командой `brew install smocka`.

Чтобы собрать код

```
$ gcc -g -c ExtremeC_examples_chapter22_1.c -o impl.o
$ gcc -g -c ExtremeC_examples_chapter22_1_smocka_tests.c -o smocka_tests.o
$ gcc impl.o smocka_tests.o -lsmocka -o ex22_tests.out
$ ./ex22_tests.out
```

Используется флаг `-lsmocka`, чтобы скомпоновать программу с установленной библиотекой CMocka.

24. Системы сборки

Сборка кодовой базы – получение конечных продуктов компиляции из исходных файлов. Например, если говорить о Си, то конечными продуктами могут быть:

- исполняемые файлы,
- разделяемые объектные файлы,
- статические библиотеки.

Задача системы сборки состоит в том, чтобы сгенерировать их из исходных файлов, из которых состоит кодовая база. Система сборки находит все зависимости в кодовой базе, загружает их и использует загруженные артефакты в процессе сборки.

24.1. Make

Система сборки Make использует файлы Makefile. Makefile – текстовый файл (без какого-либо расширения), который находится в каталоге с исходниками и содержит цели сборки и команды, позволяющие Make знать, как собирать текущую кодовую базу.

Иерархия проекта

```
ex23_1/
- Makefile
- calc/
  - add.c
  - calc.h
  - multiply.c
  - subtract.c
- exec/
  - main.c
```

Чтобы собрать этот проект с помощью системы сборки, нужно выполнить следующие команды

```
# директория out для переносимых объектных файлов
# и продуктов компиляции
$ mkdir -p out
$ gcc -c calc/add.c -o out/add.o
$ gcc -c calc/multiply.c -o out/multiply.o
$ gcc -c calc/subtract.c -o out/subtract.o
$ ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o
$ gcc -c -Icalc exec/main.c -o out/main.o
$ gcc -Lout -lcalc out/main.o -o out/ex23_1.out
```

Очень простой файл Makefile

```
build:
  mkdir -p out
  gcc -c calc/add.c -o out/add.o
  gcc -c calc/multiply.c -o out/multiply.o
  gcc -c calc/subtract.c -o subtract.o
  ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o
  gcc -c -Icalc exec/main.c -o out/main.o
  gcc -Lout -lcalc out/main.o -o out/ex23_1.out
clean:
  rm -rfv out
```

Этот Makefile содержит две цели: `build` и `clean`. Каждая из них имеет набор команд, которые должны быть выполнены при ее вызове. Этот набор называют рецептом цели.

Чтобы выполнить команды, указанные в Makefile, необходимо использовать утилиту `make`. Требуется сообщить какую цель нужно выполнить; если этого не сделать, то по умолчанию всегда выполняется первая.

Утилита **make** автоматически ищет файл **Makefile** в корневом каталоге и выполняет его первую цель. Если нам нужно выполнить цель **clean**, то мы должны указать команду **make clean**. Цель **clean** позволяет удалить файлы, сгенерированные в процессе сборки, чтобы мы могли начать с чистого листа.

Тоже простой пример Makefile

```
CC = gcc

build: prereq out/main.o out/libcalc.a
    ${CC} -lout -lcalc out/main.o -o out/ex23_1.out

prereq:
    mkdir -p out

out/libcalc.a: out/add.o out/multiply.o out/subtract.o
    ar rcs out/libcalc.a out/add.o out/multiply.o out/subtract.o

out/main.o: exec/main.c calc/calc.h
    ${CC} -c -lcalc exec/main.c -o out/main.o

out/add.o: calc/add.c calc/calc.h
    ${CC} -c calc/add.c -o out/add.o

out/subtract.o: calc/subtract.c calc/calc.h
    ${CC} -c calc/subtract.c -o out/subtract.o

out/multiply.o: calc/multiply.c calc/calc.h
    ${CC} -c calc/multiply.c -o out/multiply.o

clean: out
    rm -rf out
```

Мы объявили внутри Makefile переменную и использовали ее в разных местах по аналогии с **CC**. Особенность Make – возможность подключать другие файлы Makefile.

Каждый файл Makefile может несколько целей. Цель начинается с новой строки и содержит двоеточие в конце. Все инструкции цели (рецепта) должны иметь отступы в виде символов табуляции, чтобы программа **make** могла их распознать. Цели обладают одним интересным свойством: могут зависеть от других целей.

Например, в приведенном выше Makefile цель **build** зависит от целей **prereq**, **out/main.o** и **out/libcalc.a**. Таким образом, при вызове цели **build** система Make сначала проверяет цели, от которых та зависит, и вызывает их, если они еще не были выполнены.

Еще одно свойство целей в Makefile состоит в том, что они могут ссылаться на файлы и каталоги, размещенные на диске, такие как **out/multiply.o**, и если программа **make** не обнаружит в них свежих изменений с момента последней сборки, то соответствующая цель будет пропущена. Если файл **calc/multiply.c** в последнее время не менялся и если его уже скомпилировали, то повторная его компиляция будет бессмысленной.

Таким образом, можно компилировать только те исходный файлы, которые изменились с момента последней сборки, что позволяет избежать компиляции огромного количества кода. Конечно, чтобы данная возможность заработала, проект нужно полностью скомпилировать хотя бы раз. Но после этого компиляции и компоновке будут подлежать лишь измененные исходники.

Простое изменение заголовочного файла может инициировать компиляцию нескольких исходных файлов, которые от него зависят.

Наличие множества зависимостей между заголовчными файлами обычно имеет катастрофические последствия. Даже небольшое изменение заголовка, подключенного к большому количеству других заголовков, которые, в свою очередь, подключаются ко многим исходным файлам, может привести к сборке всего проекта или кода аналогичного масштаба. Это фактически ухудшает качество разработки и затягивает ожидание между сборками до нескольких минут. Make поддерживает регулярные выражения.

```
BUILD_DIR = out
OBJ = ${BUILD_DIR}/calc/add.o ${BUILD_DIR}/calc/subtract.o ${BUILD_DIR}/calc/multiply.o ${BUILD_DIR}/exec/main.o
CC = gcc
HEADER_DIRS = -Icalc
LIBCALCNAME = calc
LIBCALC = ${BUILD_DIR}/lib${LIBCALCNAME}.a
EXEC = ${BUILD_DIR}/ex23_1.out

build: prereq ${BUILD_DIR}/exec/main.o ${LIBCALC}
    ${CC} -L${BUILD_DIR} -l${LIBCALCNAME} ${BUILD_DIR}/exec/main.o -o ${EXEC}

prereq:
    mkdir -p ${BUILD_DIR}
    mkdir -p ${BUILD_DIR}/calc
    mkdir -p ${BUILD_DIR}/exec

${LIBCALC}: ${OBJ}
    ar rcs ${LIBCALC} ${OBJ}

${BUILD_DIR}/calc/%.o: calc/%.c
    ${CC} -c ${HEADER_DIRS} $< -o $@

${BUILD_DIR}/exec/%.o: exec/%.c
    ${CC} -c ${HEADER_DIRS} $< -o $@

clean: ${BUILD_DIR}
    rm -rf ${BUILD_DIR}
```

Вместо символа \$< подставляется развернутая правая часть цели, а вместо \$@ – сама развернутая цель. Вместо символа % подставляется элемент имени цели

```
${BUILD_DIR}/exec/%.o -> out/exec/main.o
```

Вывод

```
$ make
mkdir -p out
mkdir -p out/calc
mkdir -p out/exec
gcc -c -Icalc exec/main.c -o out/exec/main.o
gcc -c -Icalc calc/add.c -o out/calc/add.o
gcc -c -Icalc calc/subtract.c -o out/calc/subtract.o
gcc -c -Icalc calc/multiply.c -o out/calc/multiply.o
ar rcs out/libcalc.a out/calc/add.o out/calc/subtract.o out/calc/multiply.o out/exec/main.o
gcc -Lout -lcalc out/exec/main.o -o out/ex23_1.out
$
```

24.2. CMake – не система сборки

<https://cmake.org/cmake/help/latest/index.html>

CMake – *генератор скриптов сборки* для других систем, таких как Make и Ninja. CMake умеет проверять установленные зависимости и не генерировать скрипты сборки, если какая-то из них отсутствует в системе. Проверка компиляторов и их версий, определение их местоположения и возможностей – все это выполняется перед *генерацией сценария сборки*.

Если системе Make нужен файл Makefile, то CMake ищет файл CMakeLists.txt. Наличие этого файла в проекте означает, что для генерации Makefile используется CMake. К счастью, CMake, в отличие от Make, поддерживает вложенные модули. То есть у вас может быть несколько файлов CMakeList.txt, размещенных в разных каталогах проекта, и все они будут найдены; если запустить утилиту CMake в корне проекта, она сгенерирует для каждого из них подходящий файл Makefile.

```
ex23_1/  
- CMakeLists.txt  
- calc/  
  - CMakeLists.txt  
  - add.c  
  - calc.h  
  - multiply.c  
  - subtract.c  
- exec/  
  - CMakeLists.txt  
  - main.c
```

Содержимое файла CMakeLists.txt, размещенного в корне проекта

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)  
  
include_directories(calc)  
  
add_subdirectory(calc)  
add_subdirectory(exec)
```

Этот файл добавляет `calc` в число подключенных каталогов, которые будут использоваться компилятором Си во время сборки исходных файлов. Он также добавляет два подкаталога, `calc` и `exec`, каждый из которых содержит собственный файл CMakeLists.txt, описывающий процедуру компиляции содержимого.

calc/CMakeLists.txt

```
add_library(calc STATIC  
  add.c  
  subtract.c  
  multiply.c  
)
```

Это простое *объявление цели calc*. Оно означает, что у нас должна быть статическая библиотека `calc` (на самом деле после сборки она будет называться `libcalc.a`) с переносимыми объектными файлами для исходных файлов `add.c`, `subtract.c` и `multiply.c`. В CMake цели обычно предоставляют конечные продукты компиляции кодовой базы. Следовательно, в случае с модулем `calc` у нас получится ровно один продукт – статическая библиотека.

Следующий файл CMakeLists.txt

exec/CMakeLists.txt

```
add_executable(ex23_1.out
    main.c
)

target_link_libraries(ex23_1.out
    calc
)
```

Цель, объявленная в этом листинге, представляет собой исполняемый файл, который должен быть скомпонован с целью `calc`, уже объявленной в другом файле `CMakeLists.txt`.

Чтобы сгенерировать скрипты сборки (в данном случае `Makefile`) из файла `CMakeLists.txt`, размещенного в корневом каталоге

```
$ cd ex23_1
$ mkdir -p build
$ cd build
$ rm -rfv *
...
$ cmake ..
-- The C compiler ...
```

В результате в каталоге `build` был сгенерирован файл `Makefile`. Итак, в каталоге `build` есть `Makefile`. Теперь можно воспользоваться командой `make`. Она займется компиляцией.

25. 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

```
#pragma once

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

26. Ввод / вывод

26.1. 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` – шестнадцатеричное 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 восьмиричных значений добавляет в начало символ 0, для шестнадцатиричных 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.

26.2. 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.

26.2.1. 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;  
}
```

26.2.2. 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);
```

27. 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.

27.1. 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].

27.2. 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».

ВН: 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.

27.2.1. 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]);  
}
```

27.2.2. 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 автоматически 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 текущий 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.

27.2.3. 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) exit(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]);
    freep(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;
}
```

27.2.4. 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);
}
```

27.2.5. 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));
```

27.2.6. 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.

27.2.7. 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
```

Присвоить всем элементам массива значение 0 можно так

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

Количество элементов массива задается при объявлении и не может быть изменено позже. Поэтому лучше количество элементов сохранить в каком-нибудь макросе и в дальнейшем указывать этот макрос, например, при переборе элементов массива.

Замечание

Следует учитывать, что проверка выхода указанного индекса массива 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. Контроль корректности индекса входит в обязанности программиста!

После определения массива выделяется необходимый размер памяти, а в переменной сохраняется *адрес первого элемента массива*. Цикл `for` всегда можно заменить циклом `while`.

Вместо доступа к элементу массива по индексу, указанному в квадратных скобках (`p[i]`¹, где `p` – указатель) лучше использовать адресную арифметику, просто перемещая указатель. Тогда никаких дополнительных вычислений положения элемента внутри массива производится не будет.

Сортировка пузырьком

```
#include <stdio.h>

#define ARR_SIZE 5

int main(void) {
    int arr[ARR_SIZE] = {10, 5, 6, 1, 3};
    int tmp = 0, k = ARR_SIZE - 2;
    _Bool is_swap = 0;

    for (int i = 0; i <= k; ++i) {
        for (int j = k; j >= i; j--) {
            if (arr[j] > arr[j + 1]) {
                tmp = arr[j + 1];
                arr[j + 1] = arr[j];
                arr[j] = tmp;
                is_swap = 1;
            }
        }
    }
}
```

¹В этом случае положение элемента *каждый раз* вычисляется относительно начала массива, то есть `*(p + i)`

```

    }
    if (!is_swap) break;
}
for (int i = 0; i < ARR_SIZE; ++i) {
    printf("%d\n", arr[i]);
}
}

```

Этот же алгоритм на Python выглядел бы так

```

def bubble_sort(arr: t.List[int]) -> t.List[int]:
    """Bubble sorting algorithm"""
    for i in range(k + 1): # i in [0, k]
        is_swap = False
        for j in range(k, i - 1, -1): # j in [k, i]
            if (arr[j] > arr[j + 1]):
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                is_swap = True

        if not is_swap:
            break

    return arr

```

28. Символы и С-строки

В языке Си доступны два типа строк: *С-строка* и *Л-строка*. С-строка является *массивом однобайтовых символов* (тип `char`), последний элемент которого содержит *нулевой символ* (`'\0'`). Обратите внимание, что нулевой символ (нулевой байт) не имеет никакого отношения к символу `'0'`. Коды этих символов разные. Л-строка является *массивом широких символов* (тип `wchar_t`), последний элемент которого содержит *нулевой символ*.

Для хранения символа используется тип `char`. Переменной, имеющей тип `char`, можно присвоить числовое значение (код символа) или указать символ внутри апострофов. Внутри апострофов можно указать *специальные символы* – комбинации знаков, соответствующих служебным или непечатаемым символам (`\0`, `\n`, `\r`, ...).

Когда переменной присваивается символ внутри апострофов, он автоматически преобразуется в соответствующий целочисленный код.

Присвоить строку в двойных кавычках можно только при инициализации

```
char str[7] = "String"; // емкость 7, потому что 6 однобайтовых символов + нулевой символ '\0'
```

Но строку можно скопировать

```
char str[7];
strcpy(str, 7, "String");
```

Разместить С-строку при инициализации на нескольких строках нельзя. Чтобы разместить С-строку при нескольких строках, следует перед символом перевода строки указать символ `\`

```
char str[] = "string1 \
string2 \
string3";
```

Можно воспользоваться неявной конкатенацией. Если строки расположены подряд внутри одной инструкции, то они объединяются в одну большую строку

```
char str[] = "string1"
"string2" "string3";
```

Строку можно присвоить указателю, но в этом случае строка будет доступна только для чтения [2, стр. 224]

```
const char *p = "string";
printf("%s\n", p); // string
```

Мы можем объявить несколько указателей и присвоить им одинаковую строку, но все они будут ссылаться на один и тот же адрес. Иными словами, строка, указанная в программе внутри двойных кавычек, в памяти существует в единственном экземпляре. Если бы была возможность изменить такую строку, то она изменилась бы во всех указателях. Поэтому строка, присвоенная указателю, доступна только для чтения.

Объявление массива строка выглядит так

```
char str[][20] = {"String1", "String2", "String3"};
```

Или так

```
char *str[] = {"String1", "String2", "String3"};
```

При использовании второго способа попытка изменения символа приведет к ошибке при выполнении программы. Так что эти способы не эквивалентны.

После определения C-строки в переменной сохраняется адрес первого символа. Иными словами, название переменной является *указателем*, который *ссылается* на первый символ строки. Поэтому доступ к символу в строке может осуществляться как по индексу, так и с использованием адресной арифметики.

Символ можно не только получить таким образом, но и изменить (например, в Python таким способом изменить строку нельзя)

```
char str[] = "string";

str[0] = 'C'; // с помощью индекса
*(str + 1) = 'T'; // с помощью указателя
```

Объявить указатель и присвоить ему адрес строки можно следующим образом:

```
char str[] = "string";
char *p = NULL;

p = str;
*p = 'S';
++p;
*p = 'T';
```

Получить длину строки можно с помощью функции `strlen()` – возвращает количество символов без учета нулевого символа.

```
char solver_name[] = "gurobi";
printf("%d\n", strlen(solver_name));
```

При использовании L-строк все символы кодируются 2 байтами. Не следует рассматривать L-строку, как строку в какой-то кодировке, например в UTF-16. Нужно думать об L-строке как о строке в *абстрактной кодировке*, позволяющей закодировать более 65 000 символов. Когда мы говорим о строке в какой-либо кодировке, мы всегда подразумеваем C-строку.

29. Пользовательские функции

Функция – это фрагмент кода, который можно неоднократно вызывать из любого места программы. Описание функции состоит из двух частей: *объявления* и *определения*. Объявление функции (называемое также *прототипом функции*) содержит информацию о типе. Используя эту информацию, компилятор может найти несоответствие типа и количества параметров.

<Тип результата> задает *тип значения*, которое возвращает функция с помощью оператора **return**. Если функция не возвращает никакого значения, то *вместо типа* указывается *ключевое слово* **void**. Если функция не принимает параметров, то указываются круглые скобки, внутри которых задается ключевое слово **void**. После объявления функции должна стоять точка с запятой.

Определение функции содержит описание типа и названия параметров, а также реализацию. В отличие от прототипа в определении функции после типа обязательно должно быть указано *название параметра*, которое является *локальной переменной*. Эта переменная *создается при вызове функции*, а *после выхода из функции она удаляется*. Таким образом, локальная переменная видна только внутри функции.

Вернуть значение из функции позволяет оператор **return**. После исполнения этого оператора выполнение функции останавливается, и управление передается обратно в точку вызова функции. Это означает, что инструкции после оператора **return** никогда не будут выполнены.

Если перед названием функции указано ключевое слово **void**, то оператора **return** может не быть. Однако если необходимо досрочно прервать выполнение функции, то оператор **return** указывается без возвращаемого значения.

Все инструкции в программе выполняются последовательно сверху вниз. Это означает, что прежде чем использовать функцию в программе, ее необходимо предварительно *объявить*. Поэтому *объявление функции* должно быть расположено *перед вызовом функции*. Название функции всегда является *глобальным идентификатором*.

Объявления функций следует размещать в начале программы *перед* функцией **main()**, а *определения* – *после* функции **main()**. В этом случае порядок следования определений функций не имеет значения [2, стр. 315]. Пример

```
#include <stdio.h>

int sum_i(int, int); // объявление

int main(void) {
    int z = sum_i(10, 20);
    printf("z = %d\n", z); // z = 30
    return 0;
}

int sum_i(int x, int y) { // определение
    return x + y;
}
```

При увеличении размера программы объявлений и определений функций становится все больше и больше. В этом случае программу разделяют на несколько отдельных файлов. Объявления функций выносят в заголовочный файл с расширением **h**, а определения функций размещают в *одноименном* файле с расширением **c**.

Все файлы располагают в одном каталоге с основным файлом, содержащим функцию `main()`. В дальнейшем с помощью директивы `#include` подключают заголовочный файл во всех остальных файлах. Если в директиве `#include` название заголовочного файла указывается в угловых скобках, то поиск файла осуществляется в *путях поиска заголовочных файлов*. Если название указано внутри кавычек, то поиск *вначале* производится в каталоге с основным файлом, а затем в путях поиска заголовочных файлов.

Пример

my_module.c

```
#include "my_module.h" // подключение заголовочного файла

int sum_i(int x, int y) { // определение
    return x + y;
}
```

my_module.h

```
#ifndef MY_MODULE_H_
#define MY_MODULE_H_

#include <stdio.h>
int sum_i(int, int); // объявление

#endif /* MY_MODULE_H_ */
```

Все содержимое файла `my_module.h` расположено внутри условия, которое проверяется перед компиляцией. Условие выглядит так

```
#ifndef MY_MODULE_H_
// Инструкции
#endif /* MY_MODULE_H_ */
```

Это условие следует читать так: если не существует макроопределение `MY_MODULE_H_`, то вставить инструкции в то место, где подключается файл. Название макроопределения обычно совпадает с названием заголовочного файла. Только все буквы указываются в верхнем регистре и точка заменяется символом подчеркивания. Все это необходимо, чтобы объявления идентификаторов не вставлялись дважды.

Вместо этих директив можно указать в самом начале *заголовочного файла* директиву препроцессора `#pragma` со значением `once`, которая также препятствует повторному включению файла

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

Основная программа

```
#include "my_module.h"

int main(void) {
    int z = sum_i(10, 20);
    printf("z = %d\n", z);
    return 0;
}
```

При использовании больших программ создают *статическую* или *динамическую* библиотеку. Статические библиотеки становятся частью программы при компиляции, а динамические библиотеки подгружаются при запуске программы.

29.1. Способы передачи параметров в функцию

В определении функции после типа обязательно должно быть указано *название параметра*, которое является *локальной переменной*. Эта переменная создается при вызове функции, а после выхода из функции она удаляется. Таким образом, локальная переменная видна только внутри функции, и ее значение между вызовами не сохраняется (исключением являются *статические переменные*).

По умолчанию в функцию передается *копия* значения переменной. Таким образом, изменение значения внутри функции не затронет значение исходной переменной.

Передача копии значения для чисел является хорошим решением, но при использовании массивов, а также при необходимости изменить значение исходной переменной, лучше применить *передачу указателя*. Для этого при вызове функции перед названием переменной указывается оператор `&` (взятие адреса), а в прототипе функции объявляется указатель. В этом случае в функцию передается *не значение переменной, а ее адрес*. Внутри функции вместо переменной используется *указатель*

```
#include <stdio.h>

void func(int *px);

int main(void) {
    int x = 10;
    func(&x);
    printf("%d\n", x); // 30, а не 10
    return 0;
}

void fund(int *px) {
    *px += 20;
}
```

29.2. Передача массивов и строк в функцию

Многомерный массив в функцию лучше передавать как одномерный [2, стр. 322]. В этом случае в функцию передается адрес первого элемента массива, а в параметре объявляется указатель. Так как все элементы массива располагаются в памяти последовательно, зная количество элементов или размеры, можно вычислить положение текущего элемента, используя адресную арифметику.

```
#include <stdio.h>

#define ARR_ROWS 2
#define ARR_COLS 4

void func(int *pa, int rows, int cols);

int main(void) {
    int arr[ARR_ROWS][ARR_COLS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8}
    };

    func(arr[0], ARR_ROWS, ARR_COLS); // arr[0] !!!
    return 0;
}
```

```

}

void func(int *pa, int rows, int cols) {
    for (int i = 0, j; i < rows; ++i) {
        for (j = 0; j < cols; ++j) {
            printf("%2d ", *(pa + i * cols + j));
            // printf("%2d ", pa[i * cols + j]);
        }
        printf("\n");
    }
}

```

Передача в функцию массива *С-строк* осуществляется так же, как передача массива указателей. Для того чтобы в функцию не передавать количество элементов, можно при инициализации массива *С-строк* последнему элементу присвоить нулевой указатель. Этот элемент будет служить ориентиром конца массива.

```

#include <stdio.h>

void func(char *s[]);

int main(void) {
    // массив С-строк как массив указателей на char
    char *str[] = {"String1", "String2", "String3", NULL};

    func(str);
    return 0;
}

void func(char *s[]) { // или void func(char **s)
    // s -- это указатель, а не массив!
    while (*s) {
        printf("%s\n", *s);
        ++s;
    }
}

```

Передача одномерных массивов и строк осуществляется с помощью *указателей*. Объявление `char *s` эквивалентно объявлению `char s[]`. И в том и в другом случае *объявляется указатель на тип char*. Чаще используется первый способ.

29.3. Переменное количество параметров

Количество параметров в функции может быть произвольным при условии, что существует один обязательный параметр. В объявлении и определении функции *переменное число параметров* обозначается тремя точками. Например

```
int printf(const char *format, ...);
```

Вначале объявляется указатель типа `va_list`. Далее должна производиться инициализация указателя с помощью макроса `va_start()`. В первом параметре передается указатель, а во втором – название последнего обязательного параметра. Доступ к параметрам осуществляется с помощью макроса `va_arg()`, который возвращает значение текущего параметра и перемещает указатель на следующий параметр. В первом параметре макроса `va_arg()` передается указатель, а во втором – название типа данных. Макрос `va_end()` сообщает об окончании перебора параметров. Пример

```

#include <stdio.h>
#include <stdarg.h>

int sum(int n, ...);

int main(void) {
    printf("%d\n", sum(2, 20, 30)); // 50
    printf("%d\n", sum(3, 1, 2, 3)); // 6
    return 0;
}

int sum(int n, ...) {
    int result = 0;
    va_list p; // указатель типа va_list
    va_start(p, n); // инициализация указателя
    for (int i = 0; i < n; ++i) {
        result += va_arg(p, int);
    }
    va_end(p); // окончание перебора параметров
    return result;
}

```

При передаче в функцию значения типа `char` и `short` автоматически расширяются до типа `int`, а значение типа `float` – до типа `double`.

29.4. Константные параметры

Если внутри функции значение параметра не изменяется, то такой параметр следует объявить константным. Для этого при объявлении перед параметром указывается ключевое слово `const`.

Пример

```

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

```

При использовании указателей важно учитывать местоположение ключевого слова `const`.

```

void print(const char *s) { // или char const *s
    // s -- указатель!
    char s2[] = "New"; // С-строка
    s = s2; // указателю присваивается адрес первого символа строки; там можно
    s[0] = 's'; // Ошибка!
}

```

```

void print(char * const s) {
    // s - указатель!
    char s2[] = "New"; // С-строка
    s[0] = 's'; // Можно
    s = s2; // Ошибка
}

```

```

void print(const char * const s) {
    // s - указатель!
    char s2[] = "New";
    s = s2; // Ошибка
    s[0] = 's'; // Ошибка
}

```

29.5. Статические переменные и функции

Переменные, указанные в *параметрах*, а также переменные, объявленные внутри функции, являются *локальными переменными*. Эти переменные *создаются при вызове функции*, а *после выхода из функции они удаляются*.

Статические переменные позволяют отказаться от использования глобальных переменных, для *сохранения промежуточных значений* между вызовами функции. Инициализация статической переменной производится только при первом вызове функции. Если пре объявлении статической переменной не присвоено начальное значение, то переменная автоматически инициализируется нулевым значением. После завершения работы функции статическая переменная сохраняет свое значение, которое доступно при следующем вызове функции.

Ключевое слово **static** можно также указать для глобальной переменной, или функции. В этом случае *область видимости* будет ограничена *текущим файлом*.

Вернуть значение из функции позволяет оператор **return**. После исполнения этого оператора выполнение функции останавливается, и управление передается обратно в точку вызова функции. Это означает, что инструкции после оператора **return** никогда не будут выполнены.

Если внутри функции нет оператора **return**, то по достижении закрывающей фигурной скобки управление будет передано в точку вызова функции. В этом случае возвращаемое значение не определено.

Если функция не возвращает никакого значения, то вместо типа данных в объявлении и определении функции указывается ключевое слово **void**.

Функция может вернуть значение любого типа, кроме массива [2, стр. 328]. Работать с массивом необходимо через параметры функции, передавая указатель на него или возвращая указатель на конкретный элемент. Вызов функции, возвращающий какое-либо значение, можно разместить внутри выражения с правой стороны от оператора **=**.

Функция может возвращать указатель

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

char *func(char *s);

int main(void) {
    char *p = NULL, str[] = "String";

    printf("Size of 'str': %d bytes\n", (int)sizeof(str)); // 7 байт, т.к. str -- это массив

    p = func(str);
    if (p) {
        printf("%c\n", *p);
    }
    else {
        puts("NULL");
    }
    return 0;
}

char *func(char *s) {
    if (!s) return NULL;
    size_t len = strlen(s); // получаем длину строки
    printf("Size of 's': %d bytes\n", (int)sizeof(s)); // 8 байт, т.к. s -- указатель
    if (!len) return NULL; // нулевой указатель, если пусто
```

```

    else return &s[len - 1]; // указатель на последний символ
}

```

Нельзя возвращать указатель на переменные или массивы, объявленные внутри функции, так как при выходе из функции локальные переменные будут удалены [2, стр. 329].

29.6. Указатели на функции

Функции так же, как и переменные, имеют *адрес*, который можно *сохранить в указателе*. Объявление указателя на функцию

```

<Тип> (*<Название указателя>) ([<Тип 1>[, ..., <Тип N>]]);

```

Для того чтобы *присвоить указателю адрес функции*, необходимо указать название функции без параметров и круглых скобок справа от оператора =. Типы параметров указателя и функции должны совпадать

```

int (*pfunc) (int, int) = NULL;
pfunc = sum; // указателю присваивается адрес функции

```

Вызвать функцию через указатель можно так

```

int x = pfunc(30, 10); // аналогично x = sum(30, 10);
// или так
int y = (*pfunc)(30, 10);

```

Пример

```

// Объявления функций
void print_ok(void);
int add(int x, int y);
int sub(int x, int y);
int func(int x, int y, int (*pfunc)(int, int));

int main(void) {
    // Объявления указателей на функции
    void (*pf1)(void) = NULL;
    int (*pf2)(int, int) = NULL;

    // Присваивания адреса функции
    pf1 = print_ok;
    pf2 = add;

    // Вызов функции через указатели
    pf1();
    printf("%d\n", pf2(10, 20));

    printf("%d\n", func(10, 5, pf2));
    printf("%d\n", func(10, 5, add));
    printf("%d\n", func(10, 5, sub));
}

void print_ok(void) {
    puts("OK");
}

int add(int x, int y) {
    return x + y;
}

```

```
int sub(int x, int y) {
    return x - y;
}

int func(int x, int y, int (*pfunc)(int, int)) {
    return pfunc(x, y);
}
```

29.7. Передача в функцию и возврат данных произвольного типа

Если в функцию нужно передать данные *произвольного типа* или вернуть данные произвольного типа, то нужно использовать указатели с типом `void *`. Пример объявления указателя

```
void *p = NULL;
```

Присвоить адрес указателю можно, как обычно

```
int x = 10;
p = &x;
```

Для того чтобы получить значение, нужно привести указатель к определенному типу, а затем выполнить разыменование указателя

```
printf("%d\n", *((int *)p));
```

Нетипизированный указатель можно присвоить *типизированному*. В этом случае в языке Си приведение типа выполняется автоматически

```
int *px = p; // int *px = (int *)p;
```

Можно сохранить в указателе данные любого типа, но, чтобы получить данные, нужно знать, к какому типу выполнить приведение

```
double y = 20.2123;
p = &y; // нетипизированному указателю присваивается адрес переменной
printf("%.1f\n", *((double *)p));
```

30. Чтение и запись файлов

Для открытия файла предназначены функции `fopen` и `_wfopen`. В первом параметре функции принимают путь к файлу, а во втором – режим открытия файла. Функции возвращают *указатель на структуру FILE* или нулевой указатель в случае ошибки.

```
#include <stdio.h>

FILE *fp = NULL; // указатель на структуру FILE

fp = fopen("C:\\book\\file1.txt", "w");
if (fp) {
    fputs("String", fp);
    fclose(fp);
} else puts("Oops. Error");
```

Количество одновременно открытых файлов ограничено значением макроса `FOPEN_MAX`. Определение макроса выглядит так

```
#define FOPEN_MAX 20
```

В параметре `filename` в функциях `fopen()` и `_wopen()` указывается путь к файлу. Путь может быть абсолютным или относительным (в этом случае путь определяется с учетом местоположения текущего рабочего каталога).

Если файл запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается файл. Получить строковое представление текущего рабочего каталога позволяет функция `getcwd()` (заголовочный файл `unistd.h`).

Параметр `mode` в функциях `fopen()` и `_wopen()` может принимать следующие значения внутри строки:

- `r` – только чтение. Если файл не существует, то функции возвращают нулевой указатель.
- `r+` – чтение и запись. Если файл не существует, то функции возвращают нулевой указатель.
- `w` – запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан.
- `w+` – чтение и запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан.
- `a` – запись. Если файл не существует, то он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется.
- `a+` – чтение и запись. Если файл не существует, то он будет создан. Чтение производится с начала файла. Запись осуществляется в конец файла. Содержимое файла не удаляется.

Функция `fputs()` записывает С-строки в файл. Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только в момент закрытия файла. *Сбросить буфер в файл* явным образом можно с помощью функции `fflush()`.

Функция `fwrite()` – записывает объекты произвольного типа в файл. Файл должен быть открыт в бинарном режиме. Функция возвращает количество записанных объектов.

Функция `fread()` – считывает из файла объект произвольного типа. Файл должен быть открыт в бинарном режиме.

Пример

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

struct Point {
    int x;
    int y;
} point1, point2;

int main(void) {
    FILE *fp = NULL;
    fp = fopen("./file.txt", "wb");
    if (!fp) {
        perror("Error");
        exit(1);
    }

    point1.x = 10;
    point1.y = 20;

    size_t n = fwrite(&point1, sizeof(struct Point), 1, fp);
```

```

fflush(fp); // сбросить буфер в файл
fclose(fp);

fp = fopen("./file.txt", "rb");
if (!fp) {
    perror("Error");
    exit(1);
}

n = fread(&point2, sizeof(struct Point), 1, fp);
printf("%d\n", point2.x);
printf("%d\n", point2.y);
printf("%d\n", (int)sizeof(struct Point));
}

```

Для создания временного файла предназначена функция `tmpfile()`. Она возвращает файловый указатель на поток, открытый в режиме `w+b`, или нулевой указатель в случае ошибки. После закрытия временный файл автоматически удаляется.

Идентификаторы `stdin`, `stdout` и `stderr` являются файловыми указателями, связанными по умолчанию с окном консоли. Следовательно их можно использовать вместо обычных файловых указателей в функциях, предназначенных для работы с файлами.

Вывести строку в окно консоли

```

fputs("String1\nString2", stdout);
fflush(stdout);

```

Пример вывода сообщения об ошибке в поток `stderr`

```

fputs("Error message", stderr);
fflush(stderr);

```

С помощью `fflush(fp)` можно сбросить буфер в файл [2, стр. 362].

31. Потоки и процессы

Если процессор является одноядерным, то будет происходить переключение между потоками, имитируя параллельное выполнение. Потоки выполняются в рамках процесса и имеют общий доступ к его ресурсам, например глобальным переменным. Процесс содержит *минимум один поток*, который создается *при запуске* приложения. В этом *основном потоке* выполняются инструкции, расположенные *внутри тела* функции `main()`. Помимо запуска задачи в отдельном потоке можно запустить задачу в отдельном процессе, которые будет выполняться параллельно или заменит текущий процесс.

Если *несколько потоков* пытаются получить доступ к *одному ресурсу*, то результат этого действия может стать *непредсказуемым*. Для того чтобы не допустить проблем и избежать состояния гонок, необходимо выполнять *синхронизацию* критичных секций (секций, к которым имеют доступ несколько потоков). При доступе к синхронизированной секции *поток запрашивает блокировку*. Если блокировка получена, то поток изменяет что-то внутри синхронизированного блока. Если не удалось получить блокировку, то поток блокируется до момента получения разрешения. Таким образом, код внутри критичной секции в один момент времени выполняется только одним потоком как атомарная операция [2, стр. 412].

Замечание

Многие функции из стандартной библиотеки языка Си не являются потокобезопасными. Можно пользоваться потокобезопасными функциями, объявленными в заголовочном файле `strsafe.h`, а также функциями из WinAPI

Получив блокировку, можно выполнять инструкции внутри критической секции. После выхода из критической секции нужно обязательно снять блокировку, иначе секция будет заблокирована навсегда.

Вместо функции `CreateThread()` из WinAPI для создания потока управления лучше использовать функцию `_beginthreadex()`, объявленную в заголовочном файле `process.h`, так как в этом случае внутри потока можно вызывать функции из стандартной библиотеки языка Си.

Дождаться завершения работы потока позволяет функция `pthread_join()`.

32. Создание библиотек

32.1. Статические библиотеки

На первом этапе компиляции файл с исходным кодом преобразуется в объектный файл (файл с расширением `.o`). Причем каждый файл с исходным кодом преобразуется отдельно. Этим достигается ускорение компиляции, так как нужно преобразовывать лишь файлы, которые были изменены, а не все файлы с исходным кодом. Так вот на втором этапе вместо EXE-файла можно создать статическую библиотеку.

module1.h

```
#ifndef MODULE1_H_
#define MODULE1_H_
#ifdef __cplusplus
extern "C" { // чтобы библиотеку можно было использовать в программах на C++
#endif

int sum_int(int x, int y); // объявление функции

#ifdef __cplusplus
}
#endif

#endif /* MODULE1_H_ */
```

module1.c

```
#include "module1.h" // подключение заголовочного файла проекта

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

module2.h

```
#ifndef MODULE2_H_
#define MODULE2_H_

#ifdef __cplusplus
extern "C" { // чтобы библиотеку можно было использовать в программах на C++
```

```

#endif

double sum_double(double x, double y); // объявление функции

#ifdef __cplusplus
}
#endif

#endif /* MODULE2_H_ */

```

module2.c

```

#include "module2.h" // подключение заголовочного файла проекта

double sum_double(double x, double y) {
    return x + y;
}

```

Чтобы библиотеку можно было использовать в программах на языке C++, мы поместили объявления функций внутри следующего условия

```

#ifdef __cplusplus
extern "C" {
#endif

// Объявления функций и т.д.

#ifdef __cplusplus
}
#endif

```

Создать *объектный файл* можно так

```
gcc -Wall -Wconversion -c -finput-charset=cp1251 -fexec-charset=cp1251 -o moduleX.o moduleX.c
```

Здесь флаг `-c` нужен для создания объектных файлов, а флаг `-o` для указания называния объектных файлов.

В результате компиляции были созданы два файла: `module1.o` и `module2.o`. Эти два объектных файла можно объединить в статическую библиотеку с помощью следующей команды

```
ar rcs lib<Название библиотеки.a> <Объектные файлы через пробел>
```

Программа `ar.exe` создает архив, содержащий объектные файлы статической библиотеки. Пример

```
ar rcs libmodules_1_2.a module1.o module2.o
```

В результате в каталоге будет создан файл `libmodules_1_2.a`. Теперь для того чтобы использовать полученную статическую библиотеку, следует

test.c

```

#include <stdio.h>
#include <locale.h>
#include <module1.h>
#include <module2.h>

int main(void) {
    printf(sum_int(5, 10));
    printf(sum_double(3.125, 5.6));
}

```

```
}
```

Скомпилируем и запустим библиотеку

```
$ gcc -I~/C_projects/book/mylib/ -c test.c  
$ gcc test.o -L
```

При использовании статических библиотек *заголовочные файлы* обычно размещают в каталоге `include`, а сами библиотеки – в каталоге `lib`.

Статические библиотеки становятся частью программы при компиляции, а динамические библиотеки подгружаются при запуске программы. В итоге размер программы при использовании статических библиотек будет больше размера программы, использующей динамические библиотеки.

Чтобы избежать проблем, лучше размещать статические и динамические библиотеки в каталогах с разными названиями (обычно `lib` и `bin`).

Перед компиляцией запускается специальная программа – *препроцессор*, которая подготавливает код к компиляции.

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.
3. Амини К. Экстремальный Си. Параллелизм, ООП и продвинутые возможности. – СПб.: Питер, 2022. – 752 с.

Листинги