1. Правилно подреждане на логиката на програмата в различни сорс файлове/различни папки;

2. Да се използва енкапсулация, ако е необходимо. Хедър файловете да бъдат включени само където се използват;

3. Програмата да бъде разделена на функции, съдържащи малки блокове код;

4. Всяка функция трябва да прави не повече от едно нещо и дължината и не трябва да бъде повече от 20 реда;

5. Имената на функциите и променливите трябва да бъдат описателни (функции - глаголи ; променливи - съществителни);

6. Да няма повтарящи се блокове код;

7. Структурирано изпълнение на програмата в мейна;

8. Използване на спомагателни атрибути за следене на паметта;

9. Кодът да бъде преизползваем и лесен за поддръжка;

10. Кодът за бъде лесен за разширяване и скалируем(лесно да се добавят нови функционалности – да може да работи с 1 и с 1000);

11. Избягване на магически и глобални променливи;

12. Функциите да връщат стойност за по-добро проследяване при възникнали грешки;

13. Функция, която не се използва в други сорс файлове да бъде статична;

14. Добра практика е използването на пойнтери към функциите, когато искаме да работим с поведението на различни обекти в зависимост от параметрите на функцията;

15. Използваме пойнтери към функции, когато искаме да имаме няколко варианта на изпълнение;

16. Подаване на пойнтери към функции в структури с цел (полиморфизъм)подаване и използване на void пойнтери като аргумент на функциите;

typedef struct pistol\_t { int bullet; int (\*shoot)(void);} pistol\_t;

int playerShoot(void \*vp) {

pistol\_t \*shooter = (pistol\_t\*) vp;

if(shooter ->shoot()) {

perror(“Error!”);

return -1;

} else {

return 0;

}

}

17. Използване на void пойнтери в структури за запазване на информацията като колекция от данни;

18. Използване на масиви от пойнтери към функции, когато искаме да изпълняваме различни действия;

19. Използване на структури, които съдържат пойнтери към функции с цел оптимизация;

20. Не е добра практика да се пакетират структури, тъй като се забавя работата на процесора;

21. Пакетирането на структури е ВАЖНО, когато процесорът се обръщат към конкретния обект.

22. **Volatile** използваме, когато не искаме да кешираме дадена променлива;

23. Векторът представлява динамичен масив;

24. Използване на подходящи динамични структури:

- Използваме динамичен масив, когато ще модифицираме дадените елементи;

- Използваме лист, когато знаем, че ще добавяме и премахваме елементи на случайни места многократно;

- Използваме опашка, когато знаем, че елементите, които ще достъпваме са в реда, в който сме ги добавили;

- Използваме стек, когато знаем, че ще достъпваме елементите в обратния ред на тяхното добавяне;

25. Използваме static променливите, за да ограничаваме видимостта на променливите (енкапсулация);

26. Коментари за документация;

27. Разлики между stack и heap. Malloc се използва в подходящите структури

28. CMake - cross-platform tool за създаване на Make файлове. Създаване на CMakeList.txt файлове в папката и подпапките на проекта.

* Invoke options - когато билдваме с опция -g за Debug и за Release (-O3 compiler flag): ”*cmake -DCMAKE\_BUILD\_TYPE=Release*” or ”*cmake -DCMAKE\_BUILD\_TYPE=Debug*”
* ulimit -c unlimited - създава out file-ове, които можем да дебъгваме след Segmentation fault
* CMake работи с комплексни йерархични директории
* Оказвайки абсолютния път в CMakeList.txt с командата: *set(INC\_DIR ${CMAKE\_CURRENT\_SOURCE\_DIR}/include)* ни позволява да не използваме абсолютния път в самия файл, а просто името на файла, който трябва да се инклудне
* Хубаво е да не се прекаляваа с използването на функции в CMake файла
* Използване на helper функция, която да третира warning-ите като грешки

function(enable\_target\_c\_warnings target)

target\_compile\_options(

${target}

PRIVATE

-Wall

-Wextra

-Werror

-Wundef

-Wuninitialized

-Wshadow

-Wpointer-arith

-Wcast-align

-Wcast-qual

-Wunused-parameter

-Wlogical-op

-Wdouble-promotion

-Wduplicated-cond

-Wduplicated-branches

-Wnull-dereference

)

endfunction()