Ответы на вопросы к экзамену «Программирование на языке Си, 2 часть»

Обновлено 17.01.2021

Содержание

Указатель на `void`. Динамическое выделение памяти

❖ Для чего используется указатель на `void`? Приведите примеры.

Тип указателя `void` (обобщенный указатель, англ. generic pointer) используется, если тип объекта неизвестен:

- Полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов.
- Позволяет передавать в функцию указатель на объект любого типа.

Примеры: выделение (`malloc`, `calloc`), копирование областей памяти (`memmove`, `memcpy`), универсальные функции (`qsort`).

❖ Каковы особенности использования указателя на `void`? Приведите примеры.

В языке C допускается присваивание указателя типа `void` указателю любого другого типа и наоборот без явного преобразования типа указателя:

```
double d = 5.0;
double *pd = &d;
void *pv = pd;

pd = pv;
```

Указатели типа `void` нельзя разыменовывать. К указателям типа `void` не применима адресная арифметика (не знаем тип, а соответственно размер).

Без ключа `-pedantic` компилятор может рассматривать указатель на `void` как указатель на `char`.

❖ Функции для выделения и освобождения памяти `malloc`, `calloc`, `free`. Порядок работы и особенности использования этих функций.

Для выделения памяти необходимо вызвать одну из трех функций (C99 7.20.3), объявленных в заголовочном файле `stdlib.h`:

- `malloc` (выделяет блок памяти и не инициализирует его). Получает размер области памяти в байтах.
- `calloc` (выделяет блок памяти и делает каждый бит равным нулю). Получает количество элементов и размер каждого элемента в байтах.
- `realloc` (перевыделяет блок памяти). Получает указатель и новый размер в байтах.

Блок памяти после использования должен быть освобожден. Сделать это можно с помощью функции `free`.

Функция `free` (C99 7.20.3.2) освобождает (делает возможным повторное использование) ранее выделенный блок памяти, на который указывает `ptr`.

Если значением `ptr` является нулевой указатель, ничего не происходит.

Если указатель `ptr` указывает на блок памяти, который не был получен с помощью одной из функций `malloc`, `calloc` или `realloc`, поведение функции `free` не определено.

Если функции `free` передан указатель на область памяти, которая уже была освобождена (при помощи `free` или `realloc`), поведение также не определено.

❖ Функция `realloc`. Особенности использования.

`ptr == NULL && size != 0`
Выделение памяти (как `malloc`).

`ptr ≠ NULL && size == 0`
Освобождение памяти (как `free`).

`ptr ≠ NULL && size ≠ 0`
Перевыделение памяти. В худшем случае:

- Выделить новую область.
- Скопировать данные из старой области в новую.
- Освободить старую область.

❖ Общие «свойства» функций `malloc`, `calloc`, `realloc`.

Указанные функции не создают переменную, а лишь выделяют область памяти. В качестве результата они возвращают адрес расположения этой области в памяти компьютера, т.е. указатель.

Поскольку ни одна из этих функций не знает, данные какого типа будут располагаться в выделенном блоке, все они возвращают указатель на `void`.

В случае, если запрашиваемый блок памяти выделить не удалось, любая из этих функций вернет значение `NULL`.

❖ Функция выделения памяти и явное приведение типа: за и против.

За:

- Компиляция с помощью C++ компилятора. Так как язык C++ обладает более строгой типизацией, возникает необходимость в явном приведении типа.
- У функции `malloc` до стандарта ANSI C был другой прототип, где возвращаемым значением был указатель на `char`. В случае типизированных указателей неявное преобразование типа запрещено.
- Дополнительная «проверка» аргументов разработчиком.

Против:

- Начиная с ANSI C, приведение не нужно.
- Может скрыть ошибку, если забыли подключить `stdlib.h`. Компилятор, встретив функцию, прототип которой неизвестен, предполагает, что она возвращает `int`. Таким образом, на 64-битной системе функция `malloc` вернет 64-битный указатель, который будет обрезан преобразованием в `int` до 32 бит.
- При изменении типа указателя придется менять и тип в приведении.

❖ Особенности выделения 0 байт памяти.

Результат вызова функций `malloc`, `calloc` или `realloc`, когда размер запрашиваемого блока равен 0, зависит от реализации (C99 7.20.3):

- Вернется нулевой указатель.
- Вернется «нормальный» указатель, но его нельзя использовать для разыменования.

Поэтому перед вызовом этих функций нужно убедиться, что запрашиваемый размер блока не равен нулю.

❖ Способы возвращения динамического массива из функции.

Как возвращаемое значение:

```
// int *create_array(FILE *f, int *n);
int n;
int *arr = create_array(f, &n);
```

Как параметр функции:

```
// int create_array(FILE *f, int **arr, int *n);
int *arr, n;
int rc = create_array(f, &arr, &n);
```

❖ Типичные ошибки при работе с динамической памятью (классификация, примеры).

Неверный расчет количества выделяемой памяти.

Отсутствие проверки успешности выделения памяти.

Утечки памяти.

Логические ошибки:

- Wild pointer (англ. дикий указатель). Использование неинициализированного указателя.
- Dangling pointer (англ. висящий указатель). Использование указателя сразу после освобождения памяти.
- Изменение указателя, который вернула функция выделения памяти.
- Двойное освобождение памяти.
- Освобождение невыделенной или нединамической памяти.
- Выход за границы динамического массива.

❖ Подходы к обработке ситуации отсутствия свободной памяти при выделении.

Ситуация отсутствия свободной памяти при выделении называется OOM (Out Of Memory). Существуют следующие подходы к ее обработке:

- Возвращение ошибки (англ. return failure). Подход, который используем мы.
- Ошибка сегментации (англ. segmentation fault). Обратная сторона — проблемы с безопасностью.
- Аварийное завершение (англ. abort). Идея принадлежит Кернигану и Ритчи (`xmalloc`).
- Восстановление (англ. recovery). Например, `xmalloc` из Git.

Указатели на функцию

❖ Для чего используется указатель на функцию? Приведите примеры.

С помощью указателей на функции в языке Си реализуются

- функции обратного вызова (англ. callback) — «действие», передаваемое в функцию в качестве аргумента, которое обычно используется для обработки данных внутри функции (map) или для того, чтобы «связаться» с тем, кто вызвал функцию, при наступлении какого-то события;
- таблицы переходов (англ. jump table)[1];
- динамическое связывание (англ. binding).

❖ Указатель на функцию: описание, инициализация, вызов функции по указателю.

Объявление указателя на функцию

```
double trapezium(double a, double b, int n, double (*func)
(double));
```

Получение адреса функции

```
result = trapezium(0, 3.14, 25, &sin /* sin */);
```

Вызов функции по указателю

```
y = (*func)(x); // y = func(x);
```

❖ Функция `qsort`, примеры использования.

```
void qsort(void *base, size_t nmemb, size_t size, int
(*compar)(const void*, const void*));
```

Пусть необходимо упорядочить массив целых чисел по возрастанию.

[1] https://stackoverflow.com/a/48031

```
int compare_int(const void *p, const void *q)
{
    const int *a = p;
    const int *b = q;
    return *a - *b; // return *(int*)p - *(int*)q;
}
…
int a[10];
…
qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]),
compare_int);
```

❖ Особенности использования указателей на функцию.

Согласно C99 6.7.5.3 #8, выражение из имени функции неявно преобразуется в указатель на функцию.

Операция «&» для функции возвращает указатель на функцию, но из-за 6.7.5.3 #8 это лишняя операция.

Операция «*» для указателя на функцию возвращает саму функцию, которая неявно преобразуется в указатель на функцию.

Указатель на функцию может быть типом возвращаемого значения функции.

❖ Указатель на функцию и адресная арифметика.

Указатели на функции можно только сравнивать.

❖ Указатели на функцию и указатель на `void`.

Согласно C99 6.3.2.3 #1 и C99 6.3.2.3 #8, указатель на функцию не может быть преобразован к указателю на `void` и наоборот. Но POSIX требует, чтобы такое преобразование было возможно при работе с динамическими библиотеками.

make

❖ Утилита make: назначение, входные данные, идея алгоритма работы.

make — это утилита, автоматизирующая процесс преобразования файлов из одной формы в другую.

Для её работы необходимо создать так называемый сценарий сборки проекта (make-файл). Этот файл

- описывает отношения между файлами программы;
- содержит команды для обновления каждого файла.

Утилита make использует информацию из make-файла и время последнего изменения каждого файла для того, чтобы решить, какие файлы нужно обновить.

❖ Разновидности утилиты make.

Существуют следующие разновидности утилиты make:

- GNU Make (рассматривается в курсе)
- BSD Make
- Microsoft Make (nmake)

Отличие утилит — в синтаксисе.

❖ Сценарий сборки проекта: название файла, структура сценария сборки.

Файл называется `makefile` или `Makefile`[2].

```
цель: зависимость_1 … зависимость_n
[tab]команда_1
…
[tab]команда_m
```

[2] https://www.gnu.org/software/make/manual/html_node/Makefile-Names.html

Правило по умолчанию — первое в make-файле.

❖ Правила: составные части, особенности использования правил в зависимости от составных частей.

```
что создать/сделать: из чего создать
    как создать/что сделать
```

Имя цели не обязано совпадать с именем какого-либо файла. В этом случае цель называется ложной и используется для выполнения определенных действий, например, очистки директории (`make clean`). Для этого рекомендуется использовать атрибут `.PHONY`.

Если у правила нет зависимостей, то оно будет выполняться всегда.

Если несколько правил имеют одинаковые имена целей, то утилита make объединяет зависимости из каждого правила в общий список.

❖ Особенности выполнения команд.

Ненулевой код возврата может прервать выполнение сценария.

Каждая команда выполняется в своем shell.

❖ Простой сценарий сборки.

```
greeting.exe : hello.o bye.o main.o
    gcc -o greeting.exe hello.o bye.o main.o

test_greeting.exe : hello.o bye.o test.o
    gcc -o test_greeting.exe hello.o bye.o test.o

hello.o : hello.c hello.h
    gcc -std=c99 -Wall -Werror -pedantic -c hello.c

bye.o : bye.c bye.h
    gcc -std=c99 -Wall -Werror -pedantic -c bye.c
```

```
main.o : main.c hello.h bye.h
    gcc -std=c99 -Wall -Werror -pedantic -c main.c

test.o : test.c hello.h bye.h
    gcc -std=c99 -Wall -Werror -pedantic -c test.c

clean :
    rm *.o *.exe
```

❖ Алгоритм работы утилиты make на примере простого сценария сборки.

Первый запуск make:

- make читает сценарий сборки и начинает выполнять первое правило
  ```
  greeting.exe : hello.o bye.o main.o
      gcc -o greeting.exe hello.o bye.o main.o
  ```
- Для выполнения этого правила необходимо сначала обработать зависимости `hello.o`, `bye.o`, `main.o`
- make ищет правило для создания файла `hello.o`
  ```
  hello.o : hello.c hello.h
      gcc -std=c99 -Wall -Werror -pedantic -c hello.c
  ```
- Файл `hello.o` отсутствует, файлы `hello.c` и `hello.h` существуют. Следовательно, правило для создания `hello.o` может быть выполнено
  ```
  gcc -std=c99 -Wall -Werror -pedantic -c hello.c
  ```
- Аналогично обрабатываются зависимости `bye.o` и `main.o`.
- Все зависимости получены, теперь правило для построения `greeting.exe` может быть выполнено
  ```
  gcc -o greeting.exe hello.o bye.o main.o
  ```

Второй запуск make (`hello.c` был изменен)

- make читает сценарий сборки и начинает выполнять первое правило
  ```
  greeting.exe : hello.o bye.o main.o
      gcc -o greeting.exe hello.o bye.o main.o
  ```
- Для выполнения этого правила необходимо сначала обработать зависимости `hello.o`, `bye.o`, `main.o`
- make ищет правило для создания файла `hello.o`
  ```
  hello.o : hello.c hello.h
      gcc -std=c99 -Wall -Werror -pedantic -c hello.c
  ```
- Файлы `hello.o`, `hello.c` и `hello.h` существуют, но время изменения `hello.o` меньше времени изменения `hello.c`. Придется пересоздать файл `hello.o`
  ```
  gcc -std=c99 -Wall -Werror -pedantic -c hello.c
  ```
- Аналогично обрабатываются зависимости `bye.o` и `main.o`, но эти файлы были изменены позже соответствующих .c-файлов, т.е. ничего делать не нужно.
- Все зависимости получены. Время изменения `greeting.exe` меньше времени изменения `hello.o`. Придется пересоздать `greeting.exe`
  ```
  gcc -o greeting.exe hello.o bye.o main.o
  ```

❖ Ключи запуска утилиты make.

Ключ «`-f`» используется для указания имени файла сценария сборки.

Ключ «`-B`» используется для безусловного выполнения правил.

Ключ «`-n`» используется для вывода команд без их выполнения.

Ключ «`-i`» используется для игнорирования ошибок при выполнении команд.

Ключ «`-p`» показывает неявные правила и переменные.

Ключ «-r» запрещает использовать неявные правила.

❖ Использование переменных. Примеры использования.

Определить переменную в make-файле можно следующим образом:

```
VAR_NAME := value
```

Чтобы получить значение переменной, необходимо заключить ее имя в круглые скобки и перед ними поставить символ «$»:

```
$(VAR_NAME)
```

Пример использования:

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
    $(CC) -o greeting.exe $(OBJS) main.o
```

❖ Неявные правила и переменные.

Так как некоторые процессы (например, получение объектных файлов из файлов исходного кода) очевидны, утилита make может выполнять их автоматически, без соответствующих правил.

Ключ «-p» показывает неявные правила и переменные. Ключ «-r» запрещает использовать неявные правила.

Неявными переменными являются, например, CC и CFLAGS. Утилита make может присваивать им значения по умолчанию для использования в неявных правилах.

Пример:

```
# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
    $(CC) -o greeting.exe $(OBJS) main.o

test_greeting.exe : $(OBJS) test.o
    $(CC) -o test_greeting.exe $(OBJS) test.o

.PHONY : clean
clean :
    $(RM) *.o *.exe
```

❖ Автоматические переменные и их использование.

Автоматические переменные — это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд.

- Переменная «$^» означает «список зависимостей».
- Переменная «$@» означает «имя цели».
  ```
  greeting.exe : $(OBJS) main.o
  $(CC) -o greeting.exe $(OBJS) main.o
  # или $(CC) -o $@ $^
  ```
- Переменная «$<» является просто первой зависимостью.
  ```
  hello.o : hello.c hello.h
  $(CC) $(CFLAGS) -c hello.c
  # или $(CC) $(CFLAGS) -c $<
  ```

❖ Шаблонные правила. Примеры использования.

Шаблонные правила применяются не к конкретному файлу, а к группе файлов, задаваемой расширениями.

```
%.расш_файлов_целей : %.расш_файлов_зависимостей
[tab]команда_1
[tab]команда_2
…
[tab]команда_m

%.o : %.c *.h # каждый .c-файл зависит от всех .h-файлов
    $(CC) $(CFLAGS) -c $<
```

❖ Условные конструкции в сценарии сборки. Примеры использования.

```
ifeq ($(mode), debug)
    # Отладочная сборка: генерация отладочной информации
    CFLAGS += -g3
endif

ifeq ($(mode), release)
    # Финальная сборка: исключим отладочную информацию и
    # утверждения (asserts)
    CFLAGS += -DNDEBUG -g0
endif
```

Значение переменной `mode` задается через командную строку:

```
make mode=debug
```

❖ Переменные, зависящие от цели. Примеры использования.

```
# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

debug : CFLAGS += -g3
debug : greeting.exe

release : CFLAGS += -DNDEBUG -g0
release : greeting.exe
```

❖ Автоматическая генерация зависимостей.

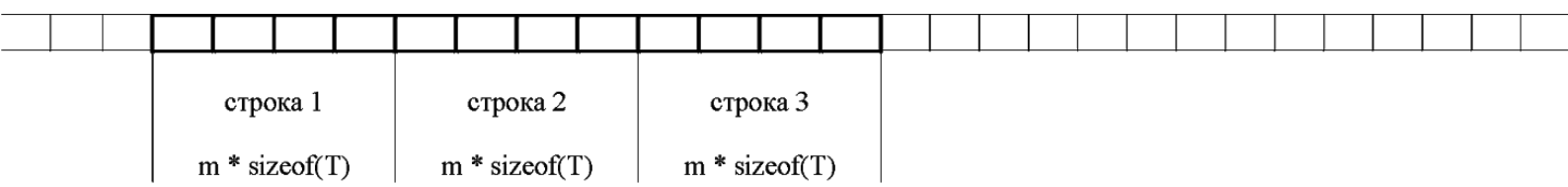
Так как компилятор знает, от каких заголовочных файлов зависит каждый файл исходного кода, можно использовать эту информацию для автоматической генерации зависимостей. Для этого у GCC существует специальный флаг `-M`. Флаг `-MM` исключит из списка зависимостей системные библиотеки.

```
# Все с-файлы (или так SRCS := $(wildcard *.c))
SRCS := hello.c bye.c test.c main.c

%.d : %.c
    $(CC) -M $< > $@

# $(SRCS:.c=.d) заменяет в переменной SRCS имена файлов
# с расширением «.c» на имена с расширением «.d»
include $(SRCS:.c=.d)
```

Директива `include` в make-файле работает по тому же принципу, что и в файлах исходного кода, с тем лишь отличием, что в случае, если искомый файл не найден, ошибка выдана не будет. Утилита make постарается найти правило, по которому сможет получить искомый файл.

Динамические матрицы

❖ Представление динамической матрицы с помощью одномерного массива. Преимущества и недостатки.



k	0	1	2	3	4	5	6	7	8	9	10	11
i;j	0;0	0;1	0;2	0;3	1;0	1;1	1;2	1;3	2;0	2;1	2;2	2;3

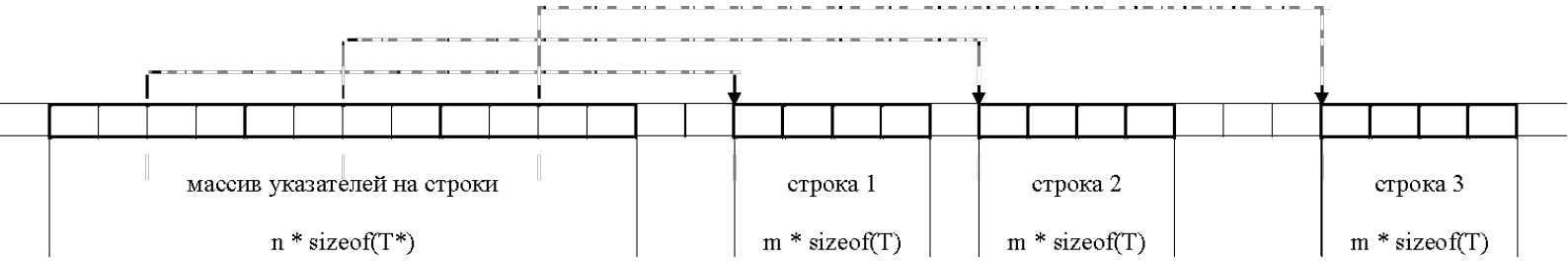
Преимущества:

- Простота выделения и освобождения памяти.
- Возможность использовать как одномерный массив.

Недостатки:

- Отладчик использования памяти (например, Dr. Memory) не может отследить выход за пределы строки.
- Нужно использовать специальную формулу для получения значения конкретной ячейки.

❖ Представление динамической матрицы с помощью массива указателей на строки/столбцы. Преимущества и недостатки.



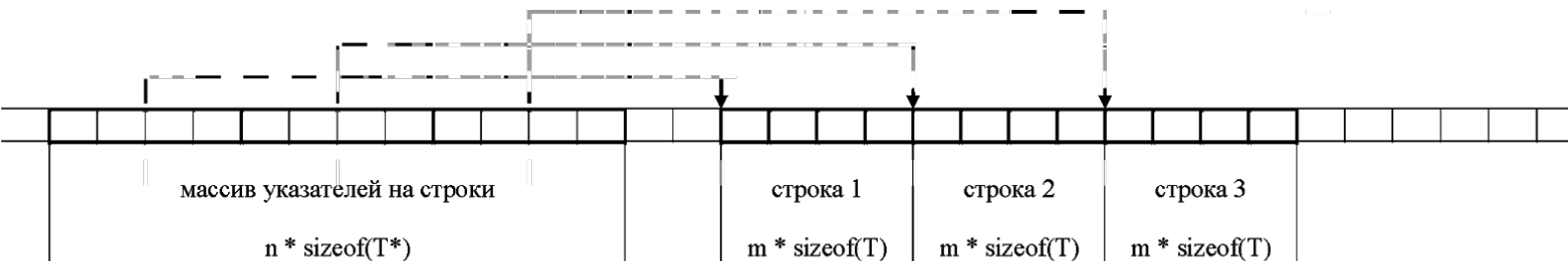
Преимущества:

- Возможность обмена строк через обмен указателей.
- Отладчик использования памяти может отследить выход за пределы строки.

Недостатки:

- Сложность выделения и освобождения памяти.
- Память под матрицу не «лежит» одной областью.

❖ Объединенный подход для представления динамической матрицы (отдельное выделение памяти под массив указателей и массив данных). Преимущества и недостатки.



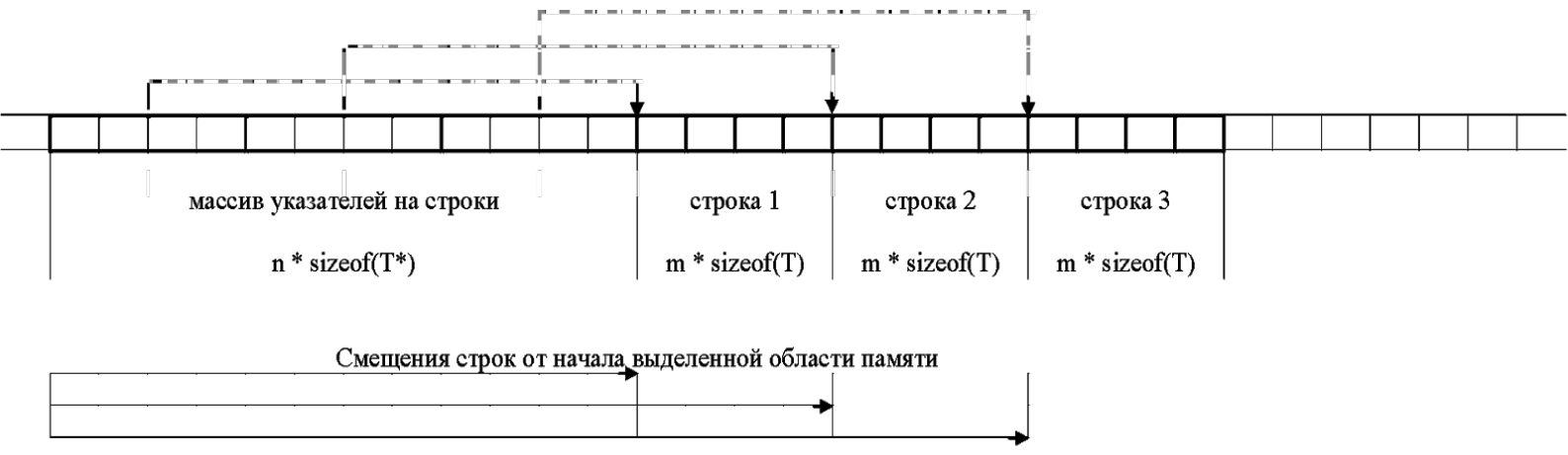
Преимущества:

- Относительная простота выделения и освобождения памяти.
- Возможность использовать как одномерный массив.
- Перестановка строк через обмен указателей.

Недостатки:

- Относительная сложность начальной инициализации.
- Отладчик использования памяти не может отследить выход за пределы строки.

❖ Объединенный подход для представления динамической матрицы (массив указателей и массив данных располагаются в одной области). Преимуществ и недостатки.



Преимущества:

- Простота выделения и освобождения памяти.
- Возможность использовать как одномерный массив.
- Перестановка строк через обмен указателей.

Недостатки:

- Сложность начальной инициализации.
- Отладчик использования памяти не может отследить выход за пределы строки.

❖ Необходимо реализовать функцию, которая может обрабатывать как статические, так и динамические матрицы. Какими способами это можно сделать?

Первый способ — хранение матрицы как одномерного массива. В таком случае в прототипе функции будет `*array`.

Другим способом является создание статического массива указателей на строки/столбцы матрицы. В таком случае в прототипе функции будет `**array`. Пример:

```
#define N 3
#define M 2

void foo_2(int **a, int n, int m);

int main(void)
{
    int a[N][M], n = N, m = M;
    int *b[N] = {a[0], a[1], a[2]};

    foo_2(b, n, m);

    return 0;
}
```

Чтение сложных объявлений

❖ Умение читать сложные объявления и использовать это на практике.

Ресурсы:

- https://medium.com/@bartobri/untangling-complex-c-declarations-9b6a0cf88c96
- https://cdecl.org

Строки/структуры и динамическое выделение памяти

❖ Функции, возвращающие динамическую строку: `strdup`/`strndup`, `getline`, `snprintf`/`asprintf`.

```
char *strdup(const char *s);
char *strndup(const char *s, size_t n);
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
int snprintf(char *restrict buf, size_t num,
             const char *restrict format, …);
int asprintf(char **strp, const char *fmt, …);
```

`strdup` — нестандартная функция языка программирования Си, создающая копию нуль-терминированной строки в куче (используя `malloc`) и возвращающая указатель на неё или `NULL` в случае ошибки. Чтобы освободить место, используемое созданной копией, необходимо вызвать `free`.

`strndup` работает таким же образом, но позволяет ограничить размер копируемой строки определенным количеством байт. Если строка длиннее ограничения `n`, то копируется только `n` байт и в конец добавляется `\0`. В случае ошибки возвращается `NULL`.

Функция `getline` считывает целую строку из потока `stream` и сохраняет адрес буфера с текстом в `lineptr`. Буфер завершается `\0` и включает символ новой строки.

Если `lineptr` равно `NULL` и `n` равно нулю перед вызовом, то `getline` выделит буфер для хранения строки. Этот буфер должен быть высвобожден программой пользователя, даже если `getline` завершилась с ошибкой.

Перед вызовом `lineptr` может содержать указатель на буфер, выделенный с помощью `malloc` размером `n` байт. Если буфер недостаточно велик для размещения строки, то `getline` изменит размер буфера с помощью `realloc`, обновив `lineptr` и `n` при необходимости. Важно отметить, что

- если буфер был выделен статически, то программа может работать некорректно;
- буфер функцией `getline` выделяется «с запасом», чтобы избежать частых перевыделений памяти.

В любом случае, при успешном выполнении вызова `lineptr` и `n` будут содержать правильный адрес буфера и его размер соответственно.

При успешном выполнении `getline` возвращает количество считанных символов, включая символ разделителя, но не включая завершающий `\0`. При ошибках чтения строки (включая условие достижения конца файла) возвращается `-1`.

`snprintf` отличается от `sprintf` тем, что в массиве, адресуемом указателем `buf`, будет сохранено не более `num - 1` символов. По окончании работы функции этот массив будет содержать `\0`. Таким образом, функция `snprintf` позволяет предотвратить переполнение буфера `buf`. `snprintf` возвращает количество символов, которое было бы записано, если бы `n` было достаточно большим, не считая `\0`. Обратите внимание, что успешным завершением является неотрицательное, меньшее `n` число.

Функция `asprintf` выделяет в памяти строку, достаточную для размещения результата, включая `\0`, и возвращает указатель на эту строку через первый аргумент. Для высвобождения выделенной памяти указатель должен быть передан функции `free`.

При успешном завершении возвращается количество выведенных байт. Если выделить память не удалось, или произошла какая-либо другая ошибка, то возвращается `-1`. При этом содержимое `strp` не определено.

❖ Feature Test Macro.

Макросы тестирования свойств позволяют программисту контролировать, какие определения будут доступны из системных заголовочных файлов при компиляции программы.

Для корректной работы макрос тестирования свойств должен быть определён до включения всех заголовочных файлов.

Некоторые макросы тестирования свойств полезны для создания переносимых приложений: они позволяют блокировать нестандартные определения. Другие же макросы, наоборот, можно использовать для разблокировки этих определений.

Например, `_GNU_SOURCE` позволяет использовать нестандартную функцию `getline`.

О доступных макросах тестирования свойств можно узнать из содержимого заголовочного файла `features.h`.

❖ Функции `memcpy`, `memmove`, `memcmp`, `memset`.

```
void *memcpy(void *dst, const void *src, size_t n);
void *memmove(void *dst, const void *src, size_t n);
int memcmp(const void *arr1, const void *arr2, size_t n);
void *memset(void *dst, int c, size_t n);
```

Функция `memcpy` копирует `n` байт из области памяти `src` в область памяти `dst`. Если области перекрываются, результат копирования не определен. Функция возвращает указатель на область памяти, в которую скопированы данные.

Функция `memmove` копирует `n` байт из области памяти `src` в область памяти `dst`. При этом области могут перекрываться. Функция возвращает указатель на область памяти, в которую скопированы данные.

Функция `memcmp` побайтно сравнивает две области памяти, на которые указывают аргументы `arr1` и `arr2`. Каждый байт определяется типом `unsigned char`. Сравнение продолжается, пока не будут проверено `n` байт или пока не встретятся различающиеся байты.

Возвращается `0`, если сравниваемые области памяти идентичны; положительное число, если при сравнении встретился отличный байт и байт из `arr1` больше соответствующего байта из `arr2`; отрицательное число, если при сравнении встретился отличный байт и байт из `arr1` меньше соответствующего байта из `arr2`.

Функция `memset` заполняет первые `n` байт области памяти, на которую указывает аргумент `dst`, символом[3], код которого указывается аргументом `c`. Функция возвращает указатель на заполняемый массив.

❖ Структуры с полями-указателями и особенности их использования.

В Си определена операция присваивания для структурных переменных одного типа. Эта операция фактически эквивалента копированию области памяти, занимаемой одной переменной, в область памяти, которую занимает другая.

При этом реализуется стратегия так называемого «поверхностного» копирования (англ. shallow copying), при котором копируется только содержимое структурной переменной, но не копируется то, на что

[3] https://stackoverflow.com/a/37241654

могут ссылаться её поля. Иногда стратегия «поверхностного» копирования может приводить к ошибкам.

❖ «Поверхностное» копирование *vs* «глубокое» копирование.

При «поверхностном» копировании (англ. shallow copying) копируется содержимое структурной переменной, но не копируется то, на что могут ссылаться поля структуры.

Стратегия так называемого «глубокого» копирования (англ. deep copying) подразумевает создание копий объектов, на которые ссылаются поля структуры.

❖ «Рекурсивное» освобождение памяти для структур с динамическими полями.

Перед освобождением памяти из-под структуры необходимо освободить память из-под всех её динамических полей.

«Рекурсивность» заключается в том, что сначала необходимо освобождать память из-под вложенных полей, «поднимаясь» на уровни выше.

Например, если есть динамически созданная структура, которая ссылается на другую структуру, в которой есть динамическая строка, сначала необходимо освободить память из-под строки, потом из-под второй структуры, и лишь затем — из-под первой.

❖ Структуры переменного размера. Приведите примеры.

TLV (Type (или Tag) Length Value) — это схема кодирования произвольных данных в некоторых телекоммуникационных протоколах.

- Type — описание назначения данных
- Length — размер данных (обычно в байтах)

- Value — данные

Первые два поля имеют фиксированный размер.

TLV-кодирование используется в

- семействе протоколов TCP/IP;
- спецификации PC/SC (smart cards);
- ASN.1 (описание сертификатов, списков отзывов).

Преимущества TLV кодирования:

- Простота разбора.
- «Тройки» TLV с неизвестным типом (тегом) могут быть пропущены при разборе.
- «Тройки» TLV могут размещаться в произвольном порядке.
- «Тройки» TLV обычно кодируются двоично, что позволяет выполнять разбор быстрее и требует меньше объема по сравнению с кодированием, основанном на текстовом представлении.

Пример:

```
typedef struct tlv
{
    int8_t type; // тип
    int16_t size; // размер данных
    uint8_t *data; // указатель на данные
} tlv_t;
```

❖ Что такое «flexible array member»? Какие особенности использования есть у этих полей? Для чего они нужны? Приведите примеры.

Последний элемент структуры с более чем одним именованным членом может иметь неполный тип массива — это и называется «flexible array member» (поле «гибкий массив»).

Особенности:

- Такое поле должно быть последним.
- Нельзя создать массив структур с таким полем.
- Структура с таким полем не может использоваться как член в «середине» другой структуры.
- Операция `sizeof` не учитывает размер этого поля (возможно, за исключением выравнивания).
- Если в этом массиве нет элементов, то обращение к его элементам — неопределенное поведение.

Пример:

```
struct student
{
    int stud_id;
    int name_len;
    int struct_size;
    char stud_name[];
};
```

❖ Flexible array member до C99.

```
struct s
{
    int n;
    double d[1];
};

struct s *create_s(int n, const double *d)
{
    assert(n >= 0);

    struct s *elem = calloc(sizeof(struct s) + (n > 1 ?
                            (n - 1) * sizeof(double) : 0), 1);
```

```
    if (elem)
    {
        elem->n = n;
        memmove(elem->d, d, n * sizeof(double));
    }

    return elem;
}
```

❖ Flexible array member *vs* поле-указатель.

Преимущества использования flexible array member:

- Экономия памяти. Не используется дополнительная память для хранения указателя.
- Локальность данных (англ. data locality). Если хранение происходит при помощи указателя, то он хранится в одном месте, а данные — в другом, что негативно сказывается на скорости обработки.
- Атомарность выделения памяти. Память выделяется один раз.
- Не требует «глубокого» копирования и освобождения.

Динамически расширяемый массив. Односвязные списки. Двоичные деревья поиска

❖ Дайте определение массива.

Массив — это последовательность элементов одного типа, расположенных в памяти друг за другом.

Доступ к элементам массива, вне зависимости от их позиции, осуществляется за постоянное время.

❖ Дайте определение линейного односвязного списка.

Связный список — это набор однотипных элементов, причем каждый из них является частью узла, который также содержит ссылку на следующий и/или предыдущий узел списка.

Узел (элемент списка) — единица хранения данных, несущая в себе ссылки на связанные с ней узлы. Узел обычно состоит из двух частей:

- Информационная часть (данные)
- Ссылочная связь (связь с другими узлами)

Линейный односвязный список — это структура данных, состоящая из узлов, каждый из которых ссылается на следующий узел списка.

Узел, на который нет указателя, является первым элементом списка и называется головой. Последний элемент списка никуда не ссылается (ссылается на `NULL`) и называется хвостом.

❖ Сравните массив и линейный односвязный список.

В отличие от массива, память под линейный односвязный список выделяется отдельно для каждого его элемента, когда возникает соответствующая необходимость.

Основное преимущество связных списков перед массивами заключается в возможности эффективного изменения расположения элементов: при удалении одного элемента адреса других элементов не изменяются. За эту гибкость приходится жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в отслеживании связей от начала списка.

Двигаться по линейному односвязному списку можно только в одном направлении, в то время как массив можно обходить как с начала в конец, так и с конца в начало.

Находясь на определенном узле линейного односвязного списка, нельзя понять, какой узел ему предшествует (в случае с массивом это элемент, индекс которого на единицу меньше текущего).

❖ Динамически расширяемый массив: описание типа, добавление нового элемента, удаление элемента. Особенности использования.

Описание типа:

```
struct dyn_array
{
    int len; // количество элементов
    int allocated; // размер выделенной памяти
    int step; // шаг перевыделения памяти
    int *data; // указатель на данные
};

#define INIT_SIZE 1
```

При разработке имеет смысл сделать `INIT_SIZE` достаточно маленьким, чтобы проверить корректность работы функций.

Добавление элемента: проверить, выделена ли память под массив. Если нет, выделить (размером `INIT_SIZE` × `sizeof(int)`). Если память была выделена, но ее не хватает (`len` больше или равен `allocated`), то увеличить объем выделенной памяти в `step` × `sizeof(int)` раз. Если памяти хватает, просто добавить элемент.

Удаление элемента: проверить индекс. Если порядок элементов в массиве не важен, добавлять следующий на место «удаленного». Если порядок важен, сдвинуть элементы влево после удаленного. Для сдвига можно использовать обычный `for` или `memmove`. Использование `memcpy` (и `strcpy`) невозможно из-за перекрывания областей памяти.

Особенности использования:

- Удвоение размера массива при каждом вызове `realloc` сохраняет средние «ожидаемые» затраты на копирование элемента.

- Поскольку адрес массива может измениться, программа должна обращаться к элементами массива по индексам.

❖ Почему при добавлении нового элемента память необходимо выделять блоками, а не под один элемент?

Для уменьшения потерь при распределении памяти изменение размера должно происходить относительно крупными блоками. Так как работа с памятью является достаточно ресурсоемкой операцией, выполнять ее следует как можно реже.

❖ Линейный односвязный список: описание типа, добавление нового элемента в начало/конец списка, вставка элемента перед/после указанного, удаление элемента из списка, обход списка, удаление памяти из-под всего списка.

Описание типа:

```
typedef struct person_t person_t;

struct person_t
{
    const char *name;
    int birth_year;
    person_t *next;
};
```

Добавление нового элемента в начало списка: указатель на следующий узел добавляемого элемента необходимо установить так, чтобы он указывал на текущую голову списка. Головой нового списка станет добавленный элемент.

Добавление нового элемента в конец списка: указатель на следующий узел последнего элемента текущего списка начинает указывать на добавляемый элемент.

Чтобы удалить элемент из списка, его необходимо сначала найти, а затем обновить указатель на следующий узел элемента, предшествующего удаляемому, так, чтобы он указывал на следующий элемент текущего (удаляемого) узла. После этого необходимо освободить память из-под удаленного узла.

Вставить элемент до или после указанного просто: необходимо найти «опорный» элемент, и затем изменить или его указатель на следующий узел, или указатель на следующий узел предшествующего узла. Также потребуется изменить указатель у добавляемого узла. Задачу вставки элемента до указанного (со сложностью O(N)) можно свести к задаче вставке после указанного (со сложностью O(1)), поменяв местами информационные части «опорного» и вставляемого элементов.

Функции, изменяющие список, должны возвращать указатель на новый первый элемент.

Обход списка, как уже было сказано, возможен только в одном направлении — том, в котором узлы связаны друг с другом. Функция, осуществляющая обход списка, может принимать callback-функцию, которая будет вызвана для каждого узла списка. В таком случае прототип функции обхода будет выглядеть следующим образом:

```
void list_traverse(person_t *head,
                   void (*f)(person_t*, void*), void *arg)
```

Под удалением памяти из-под всего списка подразумевается освобождение памяти из-под всех его элементов.

- ❖ Возможные улучшения «классической» реализации линейного односвязного списка.

Сложность операции добавления в конец линейного односвязного списка составляет O(N); её можно уменьшить, введя дополнительный указатель на конец списка.

Создание универсального списка: указатель на данные имеет тип `void`.

Двусвязные списки дают возможность поиска последнего элемента и удаление текущего за O(1), но требуют больше ресурсов.

- ❖ Двоичное дерево поиска: описание типа, добавление элемента, поиск элемента (рекурсивный и нерекурсивный варианты), обход дерева, освобождение памяти из-под всего дерева.

Дерево — это связный ациклический граф.

Двоичным деревом поиска называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовем их левым и правым), и все вершины, кроме корня, имеют родителя.

Описание типа:

```
typedef struct node_t node_t;

struct node_t
{
    const char *name;
    // меньшие
    node_t *left;
    // большие
    node_t *right;
};
```

Добавление элемента в дерево происходит по следующему принципу: каждая вершина, начиная с корня, сравнивается с добавляемой. Если текущая вершина меньше добавляемой, то рассматривается правая ветвь, если больше — левая. Как только очередная ветвь оказывается пустой, новая вершина занимает ее место.

Поиск элемента происходит по аналогичному принципу до тех пор, пока не будет найдена искомая вершина, или ветвь не окажется пустой. В рекурсивном варианте функция запускается для каждого соответствующего поддерева, в нерекурсивном используется цикл с предпроверкой. Сложность поиска составляет $O(\log_2 N)$.

Обход дерева возможен тремя способами:

- Префиксным (или прямым). В этом случае сначала к вершине применяется функция, и лишь затем берется следующая. Сначала посещается левое поддерево, затем правое. Этот подход используется достаточно редко.
- Инфиксным (или фланговым, или поперечным). Сначала посещается левое поддерево, затем к вершине применяется функция, затем посещается правое поддерево. Используется для обхода дерева в установленном порядке (для двоичного дерева поиска этот обход даст отсортированный список).
- Постфиксным (или обратным). Сначала посещаются оба поддерева, затем к вершине применяется функция. Используется, когда необходимо знать характеристики обоих поддеревьев. Например, для поиска максимальной высоты дерева.

Под освобождением памяти из-под дерева подразумевается освобождение памяти из-под всех его вершин. Для этого используется постфиксный (обратный) обход.

❖ Язык DOT, примеры использования. Утилита Graphviz.

DOT — это язык описания графов.

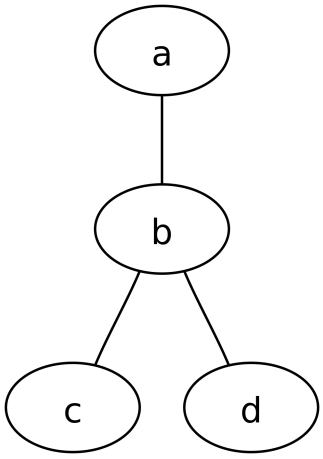
Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением `.gv` в понятном для человека и обрабатывающей программы формате.

В графическом виде графы, описанные на языке DOT, представляются с помощью специальных программ. Например, Graphviz.

Пример (имя, как и точки с запятой, можно не использовать):

```
graph graphname {
    a -- b -- c;
    b -- d;
}
```

Результат:



Область видимости, время жизни, связывание

❖ Что такое область видимости имени?

Область видимости (англ. scope) имени — это часть текста программы, в пределах которой имя может быть использовано.

❖ Какие области видимости есть в языке Си? Приведите примеры.

В языке Си выделяют 4 области видимости.

Блок:

- Переменная, определенная внутри блока, имеет область видимости в пределах блока.
- Формальные параметры функции имеют в качестве области видимости блок, составляющий тело функции.

```
double f(double a) // начало области видимости переменной a
{
    double b; // начало области видимости переменной b
    …
    return b;
} // конец области видимости переменных a и b
```

Файл:

- Область видимости в пределах файла имеют имена, описанные за пределами какой бы то ни было функции.
- Переменная с областью видимости в пределах файла видна на протяжении от точки ее описания и до конца файла, содержащего это определение.
- Имя функции всегда имеет файловую область видимости.

```
#include <stdio.h>

int max;

void f(void)
{
    printf("%d\n", max);
}

int main(void)
{
    max = 5;
    f();
}
```

Функция:

- Метки — это единственные идентификаторы, область действия которых — функция.
- Метки видны из любого места функции, в которой они описаны.
- В пределах функции имена меток должны быть уникальными.

```
void foo()
{
    {
        {
            goto b; // ок
        }
```

```
    }

    b: printf("Hello, World!\n");
}
```

Прототип функции:

- Область видимости в пределах прототипа функции применяется к именам переменных, которые используются в прототипах функций.
  `int f(int i, double d);`
- Область видимости в пределах прототипа функции простирается от точки, в которой объявлена переменная, до конца объявления прототипа.
  ```
  int f(int, double); // ок
  int f(int i, double i); // ошибка компиляции
  ```

❖ Какие правила перекрытия областей видимости есть в языке Си? Приведите примеры.

Переменные, определенные внутри некоторого блока, будут доступны из всех блоков, вложенных в данный:

```
{
    int a = 1;
    …
    {
        int b = 2;
        …
        printf("%d %d\n", a, b); // ок
    }

    printf("%d %d\n", a, b); // ошибка компиляции
}
```

Возможно определить в одном из вложенных блоков переменную с именем, совпадающим с именем одной из «внешних» переменных:

```
{
    int a = 1;

    {
        a += 5;

        double a = 2.0;

        printf("%g\n", a); // 2
    }

    printf("%d\n", a); // 6
}
```

❖ Что такое блок?

В языке Си блоком считается последовательность объявлений, определений и операторов, заключенная в фигурные скобки.

❖ Какие виды блоков есть в языке Си?

Существуют два вида блоков:

- Составной оператор
- Определение функции

Блоки могут включать в себя составные операторы, но не определения функций.

❖ Что такое объявление? Приведите примеры.

Объявление — это процесс связывания имени с некоторым типом.

```
float calculate(float, float);
float calculate(float a, float b);
```

❖ Что такое определение? Приведите примеры.

Определение — это процесс связывания имени с некоторым типом и выделение соответствующего объема памяти.

```
float calculate(float a, float b)
{
    return a * 0.5 + b / 2.0;
}

int b = 10;
```

❖ Для чего нужны объявления?

Объявления нужны для того, чтобы однопроходный компилятор языка Си корректно производил проверку типов и генерацию соответствующего объектного кода.

❖ Чем отличаются определения и объявления?

В объявлении, в отличие от определения, не выделяется память.

Объявлений может быть сколько угодно, а определений — только одно.

❖ Что такое время жизни программного объекта?

Время жизни (англ. storage duration) — это интервал времени выполнения программы, в течение которого программный объект существует.

Программный объект — это переменная или функция.

«Программный объект существует» означает, что под этот программный объект (переменную или функцию) выделена память.

Выделяют глобальное (статическое), локальное (автоматическое) и динамическое (выделенное) виды времени жизни.

❖ Какие виды времени жизни есть у переменных?

У переменных есть следующие виды времени жизни:

- Глобальное. Переменная существует на протяжении всего времени выполнения программы. Такую переменную можно получить, определив ее вне какой-либо функции.
- Локальное. Переменная существует во время выполнения блока, в котором определена. Такие переменные создаются при каждом входе в блок и уничтожаются при выходе из него. Пример: локальные переменные, параметры функции.

Переменные не могут обладать динамическим временем жизни, так как при помощи функций, выделяющих память, создать переменную невозможно.

❖ Какие виды времени жизни есть у функций?

Все функции обладают глобальным временем жизни.

❖ Как время жизни влияет на область памяти, в которой располагается программный объект?

Программные объекты, обладающие глобальным временем жизни, хранятся в секции данных.

Программные объекты, обладающие локальным временем жизни, хранятся в стеке (за исключением регистровых переменных).

Программные объекты, обладающие динамическим временем жизни, хранятся в куче.

❖ Что такое связывание?

Связывание (англ. linkage) определяет область программы, в которой программный объект может быть доступен другим функциям программы.

Области связывания — это не то же самое, что области видимости! Под областью связывания понимают или функцию, или файл, или всю программу целиком.

❖ Какие виды связывания есть в языке Си?

Стандарт языка Си определяет три формы связывания:

- Внешнее (англ. external)
- Внутреннее (англ. internal)
- Никакое (англ. none)

❖ Как связывание влияет на «свойства» объектного/исполняемого файла? Что это за «свойства»?

Имена с внешним связыванием доступны во всей программе. Подобные имена «экспортируются» из объектного файла, создаваемого компилятором.

Имена с внутренним связыванием доступны только в пределах файла, в котором они определены, но могут «разделяться» между всеми функциями этого файла.

Имена без связывания принадлежат одной функции и не могут разделяться вообще.

❖ Какими характеристиками (область видимости, время жизни, связывание) обладает переменная в зависимости от места своего определения?

Глобальная переменная (определенная вне какой-либо функции) обладает глобальным временем жизни, файловой областью видимости и внешним связыванием.

Локальная переменная (определенная на уровне блока) обладает локальным временем жизни, блочной областью видимости и отсутствием связывания.

❖ Какими характеристиками (область видимости, время жизни, связывание) обладает функция в зависимости от места своего определения?

Все функции, вне зависимости от места своего определения, имеют файловую область видимости, глобальное время жизни и внешнее связывание.

❖ Какие классы памяти есть в языке Си?

В языке Си существует четыре класса памяти: `auto`, `static`, `extern` и `register`.

❖ Для чего нужны классы памяти?

Классы памяти позволяют управлять временем жизни, областью видимости и связыванием переменной (до определенной степени).

❖ Какие классы памяти можно использовать с переменными? С функциями?

С переменными можно использовать `auto`, `static`, `extern` и `register`. С функциями — `static` и `extern`.

❖ Сколько классов памяти может быть у переменной? У функции?

Переменные и функции могут одновременно иметь не более одного класса памяти.

❖ Какие классы памяти по умолчанию есть у переменной? У функции?

По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к классу автоматической памяти.

По умолчанию любая переменная, объявленная вне какого-либо блока, не имеет класса памяти.

По умолчанию функция имеет класс памяти `extern`.

❖ Расскажите о классе памяти `auto`.

Применим только к переменным, определенным в блоке.

Переменная, принадлежащая к классу `auto`, имеет локальное время жизни, блочную область видимости и не имеет связывания.

По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к классу автоматической памяти.

Смысл указывать класс памяти `auto` есть только в том случае, когда требуется показать другому программисту необходимость использования именно локальной переменной.

❖ Расскажите о классе памяти `static`.

Класс памяти `static` может использоваться с любыми переменными независимо от места их расположения:

- Для переменной вне какого-либо блока `static` изменяет связывание этой переменной на внутреннее.
- Для переменной в блоке `static` изменяет время жизни с автоматического на глобальное.

Статическая переменная, определенная вне какого-либо блока, имеет глобальное время жизни, область видимости в пределах файла и внутреннее связывание.

```
static int i;

void f1(void)
{
    i = 1;
}

void f2(void)
{
    i = 5;
}
```

Этот класс памяти скрывает переменную в файле, в котором она определена.

Статическая переменная, определенная в блоке, имеет глобальное время жизни, область видимости в пределах блока и отсутствие связывания.

```
void f(void)
{
    static int j;
    …
}
```

Такая переменная сохраняет свое значение после выхода из блока.

Инициализируется только один раз.

Если функция вызывается рекурсивно, это порождает новый набор локальных переменных, в то время как статическая переменная разделяется между всеми вызовами.

Использования с функциями:

```
static int g(int i);
// g имеет внутреннее связывание. Из других файлов
// вызываться не может.
```

Использование класса памяти `static` для функций полезно, потому что:

- Функции, определенные со `static`, не видны в других файлах и могут безболезненно изменяться (инкапсуляция).
- Так как функция имеет внутреннее связывание, ее имя может использоваться в других файлах.

❖ Расскажите о классе памяти `extern`.

Класс памяти `extern` помогает разделить переменную между несколькими файлами.

Используется для переменных определенных как в блоке, так и вне блока:

- Переменная вне блока с классом памяти `extern` имеет глобальное время жизни, файловую область видимости и «непонятное» связывание (зависит от определения переменной, компилятор полагает, что связывание внешнее).
- Переменная в блоке с классом памяти `extern` имеет глобальное время жизни, видимость в блоке и «непонятное» связывание.

```
// file_1.c
int number;

// file_2.c
int process(void)
{
    if (number > 5) // ошибка
    …
```

Использование `extern`:

```
// file_1.c
int number;
```

```
// file_2.c
extern int number;

int process(void)
{
    if (number > 5) // ок
    …
```

Объявлений (`extern int number`) может быть сколько угодно.

Определение (`int number`) должно быть только одно.

Объявления и определение должны быть одинакового типа.

```
extern int number = 5; // определение
```

Использование с функциями:

```
extern int f(int i);
// f имеет внешнее связывание. Может вызываться из других
// файлов. Не имеет смысла: значение для функций по
// умолчанию
```

❖ Расскажите о классе памяти `register`.

Использование класса памяти `register` — просьба к компилятору разместить переменную не в памяти, а в регистре процессора. Это может быть полезно, если к переменной обращаются достаточно часто, а значит сокращение времени чтения и записи ускорит программу. Пример: счетчик цикла.

Используется только для переменных, определенных в блоке (например, параметры функции).

Задает локальное время жизни, видимость в блоке и отсутствие связывания.

Обычно не используется.

К переменным с классом памяти `register` нельзя применять операцию получения адреса `&`.

❖ Для чего используется ключевое слово `extern`?

Ключевое слово `extern` используется для объявления и определения внешних переменных.

❖ Особенности совместного использования ключевых слов `static` и `extern`.

Если компилятор сначала встречает `extern`, то предполагает, что переменная имеет внешнее связывание. Если затем встречается `static`, то происходит ошибка (связывание меняться не может):

«If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.» (C99 6.2.2 #7)

❖ Как описать автоматическую глобальную переменную?

Глобальные переменные имеют внешнее связывание, а автоматические не имеют связывания вообще. Поэтому описать автоматическую глобальную переменную невозможно.

❖ Какая переменная называется глобальной?

Переменная, определенная вне какой-либо функции, называется глобальной.

❖ Какая переменная называется локальной?

Переменная, определенная на уровне какого-либо блока или прототипа функции, называется локальной.

❖ Каким значением по умолчанию инициализируются автоматические переменные?

Автоматические переменные по умолчанию не инициализируются (содержат «мусор»).

❖ Каким значением по умолчанию инициализируются переменные с глобальным временем жизни?

Переменные с глобальным временем жизни по умолчанию инициализируются нулём.

❖ Какие недостатки есть у использования глобальных переменных?

Если глобальная переменная получает неверное значение, трудно понять какая функция работает неправильно.

Изменение глобальной переменной требует проверки правильности работы всех функций, которые ее используют.

Функции, которые используют глобальные переменные, трудно использовать в других программах.

❖ Объектный файл, секции, таблица символов.

Объектный файл представляют собой блоки машинного кода и данных с неопределенными адресами ссылок на данные и подпрограммы в других объектных модулях, а также список своих подпрограмм и данных.

Объектный файл состоит из секций, которые содержат данные в широком смысле этого слова:

- Заголовки (метаинформация, необходимая для организации самого файла)
- Код (`.text`)
- Данные (`.data`, `.rodata`, `.bss`)

- Таблицу символов (`.symtab`)
- и другое...

Типы символов

Буква	Расположение
`B`, `b`	Секция неинициализированных данных (`.bss`)
`D`, `d`	Секция инициализированных данных (`.data`)
`R`, `r`	Секция данных только для чтения (`.rodata`)
`T`, `t`	Секция кода (`.text`)
`U`	Символ не определен, но ожидается, что он появится.

Строчные буквы означают «локальные» символы, «заглавные» — внешние (глобальные).

❖ Что делает компоновщик?

Задача компоновщика состоит в том, чтобы соединить несколько объектных файлов в двоичный исполняемый файл, добавив системный код, который подготавливает программу к запуску и последующему завершению. Процесс компоновки включает в себя, главным образом, преобразование символьных адресов в числовые. Результатом процесса компоновки обычно является исполняемая программа.

Процесс компоновки состоит из нескольких шагов:

- Перемещение (англ. relocation). «Склеивание» нескольких секций в одну.
- Разрешение ссылок (англ. reference resolving). Ссылки внутри каждого файла уже разрешены.
    - Поиск кода, который вызывает что-то за пределами своей исходной секции.

- Поиск места, где теперь располагается вызываемый код.
- Замена «поддельного» адреса на настоящий.

❖ Журналирование, подходы к реализации.

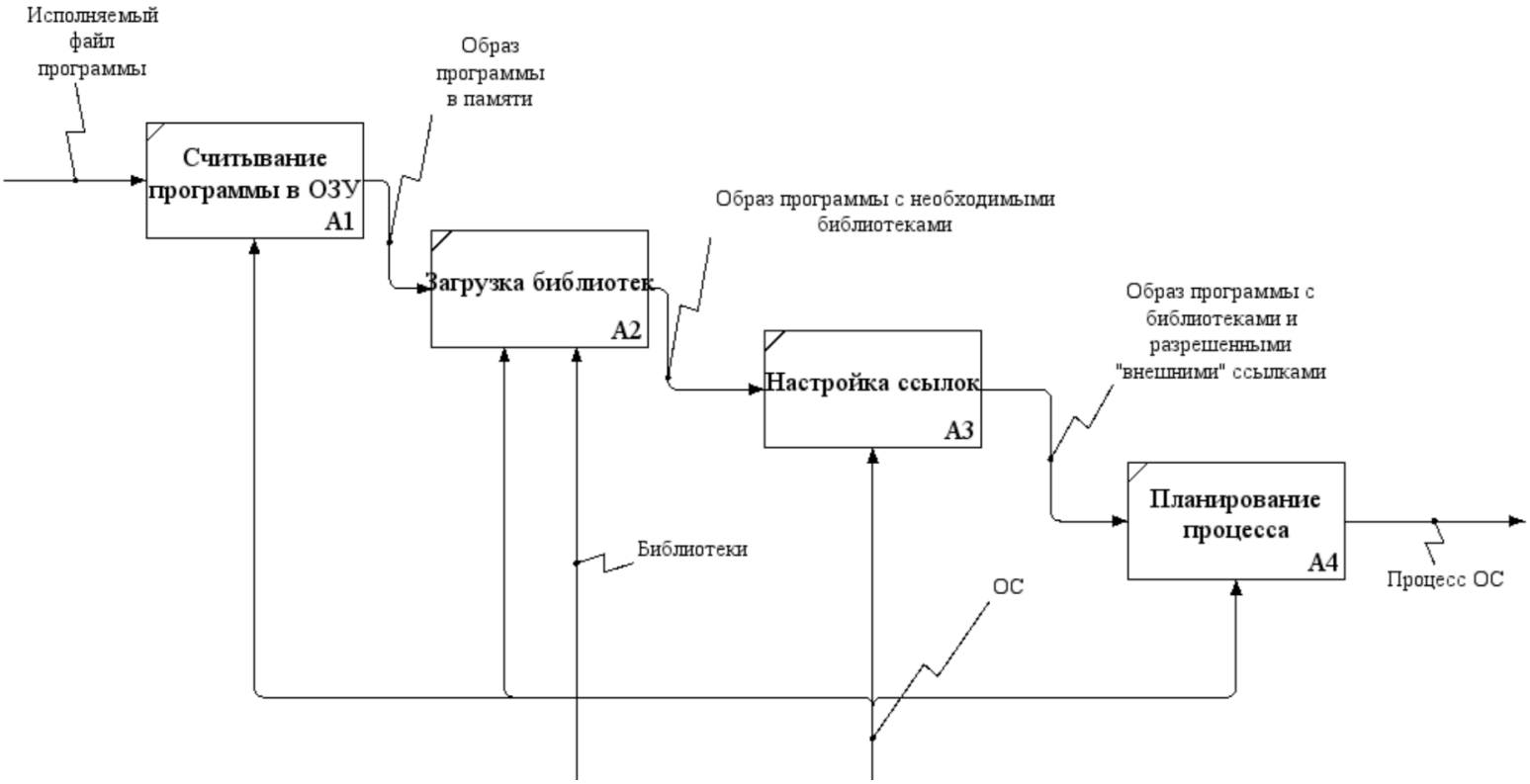
Первый подход — использование глобальной файловой переменной. Недостаток — неконтролируемый доступ к этой переменной.

Второй подход — использование класса памяти `static` с файловой переменной. Теперь «снаружи» обратиться к этой переменной нельзя. Можно, конечно, при помощи функции `log_get` получить доступ к этой переменной, а затем её изменить, однако случайно это сделать вряд ли получится.

Третий подход (идеальный) — введение функции записи в журнал. В таком случае изменить файловую переменную никаким образом не получится.

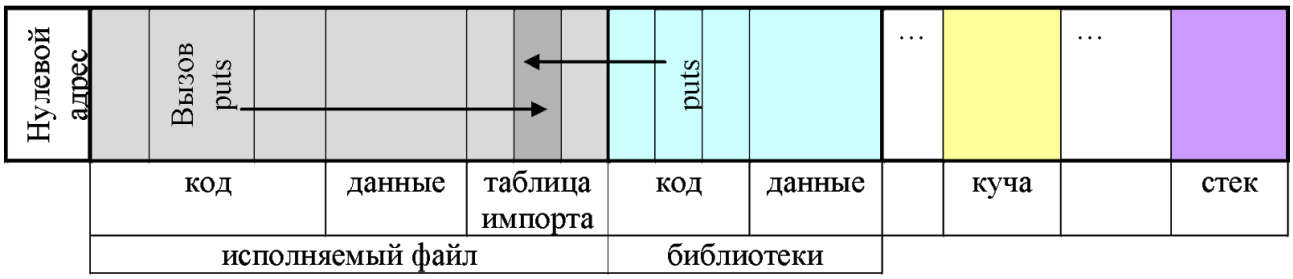
Схема распределения памяти в программе на языке Си. Стек. Куча

❖ Процесс запуска программы («превращения в процесс»).



❖ Абстрактное адресное пространство программы.

Программа из исполняемого файла помещается в так называемую «абстрактную» память (виртуальное адресное пространство). Эта память устроена таким образом, что программе «кажется», что в памяти располагается только она и библиотеки, которые ей нужны.



Нулевой адрес обладает следующим свойством: гарантируется, что ни одна переменная или функция программы не будет расположена по этому адресу.

Таким образом, у программы на языке Си есть три источника памяти:

- Секции данных (глобальные переменные)
- Куча (с ней работают `calloc`, `malloc`, `realloc`)
- Стек

❖ Опишите достоинства и недостатки локальных переменных.

Достоинства:

- Память под локальные переменные выделяет и освобождает компилятор.

Недостатки:

- Время жизни локальной переменной «ограничено» блоком, в котором она определена.
- Размер размещаемых в автоматической памяти объектов должен быть известен на этапе компиляции.
- Размер автоматической памяти в большинстве случаев ограничен.

❖ Локальные переменные создаются в так называемой «автоматической памяти». Почему эта память так называется?

Память называется автоматической, так как ее выделяет и освобождает компилятор автоматически, без участия программиста.

❖ Для чего в программе используется аппаратный стек?

Аппаратный стек используется для

- вызова функции (`call name`). В стек помещается адрес команды, следующей за командой `call`, и управление передается по адресу метки `name`;
- возврата из функции (`ret`). Из стека извлекается адрес возврата `address`, на который передается дальнейшее управление;
- передачи параметров в функцию;
- выделения и освобождения памяти под локальные переменные.

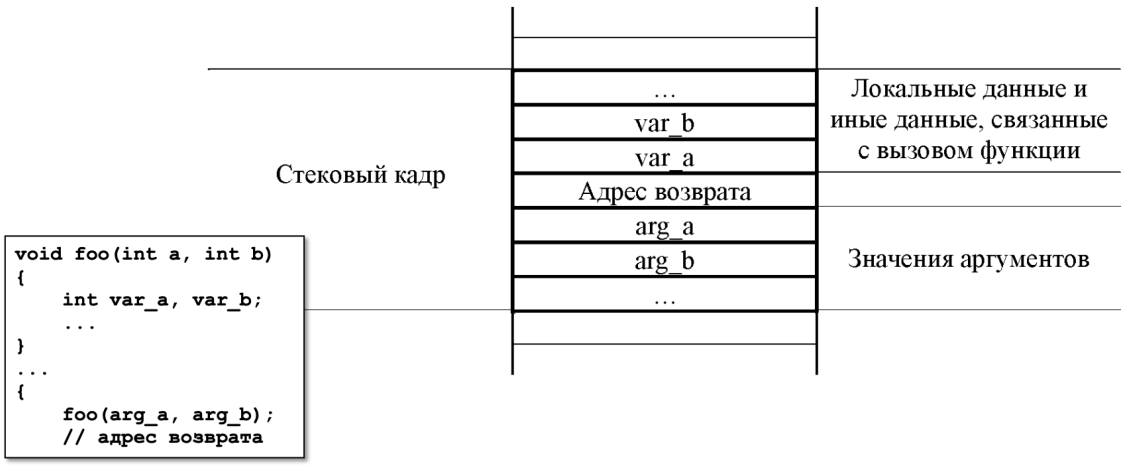
❖ Что такое кадр стека?

Стековый кадр или кадр стека (фрейм) — это механизм передачи аргументов и выделения временной памяти с использованием аппаратного стека.

❖ Для чего в программе используется кадр стека? Приведите примеры.

В стековом кадре размещаются

- значения фактических аргументов функции;
- адрес возврата;
- локальные переменные;
- иные данные, связанные с вызовом функции.



❖ Какие преимущества и недостатки есть у использования кадра стека?

Преимущества:

- Удобство и простота использования.

Недостатки:

- Производительность. Передача данных через память без необходимости замедляет выполнение программы.
- Безопасность. Стековый кадр перемежает данные приложения с критическими данными: указателями, значениями регистров и адресами возврата.

❖ Что такое соглашение о вызове?

Соглашение о вызове (англ. calling convention) — описание технических особенностей вызова подпрограмм, определяющее

- способы передачи параметров подпрограммам;
- способы вызова (передачи управления) подпрограмм;
- способы передачи результатов вычислений, выполненных подпрограммами, в точку вызова;
- способы возврата (передачи управления) из подпрограмм в точку вызова.

❖ Какое соглашение о вызове используется в языке Си? В чем оно заключается?

В языке Си используется соглашение о вызове под названием cdecl (C declaration). Оно заключается в том, что

- аргументы передаются через стек справа налево (т.е. в обратном порядке);
- очистку стека производит вызывающая сторона;

- результат функции возвращается через регистр `EAX`, но, так как он имеет ограниченный размер, используются дополнительные договоренности о возврате, например, структур.

❖ Что такое переполнение буфера? Чем оно опасно?

Так как буфер выделяется на стеке, при переполнении происходит разрушение его состояния, что может привести к неопределенному поведению программы.

Могут быть испорчены данные, идущие непосредственно за буфером.

Переполнение буфера является одним из наиболее популярных способов взлома компьютерных систем, так как большинство языков высокого уровня использует технологию стекового кадра — размещение данных в стеке процесса, смешивая данные программы с управляющими данными (в том числе адреса начала стекового кадра и адреса возврата из исполняемой функции).

Переполнение буфера может вызывать аварийное завершение или зависание программы, ведущее к отказу обслуживании. Отдельные виды переполнений, например переполнение в стековом кадре, позволяют злоумышленнику загрузить и выполнить произвольный машинный код от имени программы и с правами учетной записи, от которой она выполняется.

В архитектуре x86 стек растёт от бóльших адресов к меньшим, то есть новые данные помещаются перед теми, которые уже находятся в стеке. Записывая данные в буфер, можно осуществить запись за его границами и изменить находящиеся там данные, в частности, изменить адрес возврата.

❖ Почему нельзя из функции возвращать указатель на локальную переменную, определенную в этой функции?

Из функции нельзя возвращать указатель на локальную переменную, определенную в этой функции, так как при выходе из нее эта переменная перестанет существовать и указатель превратится в «висящий».

❖ Для чего в программе используется куча?

Куча используется для динамического выделения памяти.

❖ Происхождение термина «куча».

Согласно Дональду Кнуту, «Several authors began about 1975 to call the pool of available memory a heap [куча]».

Скорее всего, происхождение термина связано с тем, что авторы противопоставляли его стеку, в котором есть определенная организация и алгоритм обслуживания (первым вошел — последним вышел).

❖ Свойства области памяти, которая выделяется динамически.

- Для хранения данных используется «куча». Создать переменную в «куче» нельзя, но можно выделить память под нее.
- `malloc` выделяет по крайней мере указанное количество байт (меньше нельзя, больше можно).
- Указатель, возвращенный `malloc`, указывает на выделенную область (т.е. область, в которую программа может писать и из которой может читать данные).
- Ни один другой вызов `malloc` не может выделить эту область или ее часть, если только она не была освобождена с помощью `free`.

❖ Как организована куча?

В куче нет определенного порядка расположения элементов.

❖ Алгоритм работы функции `malloc`.

- Просмотреть список занятых/свободных областей памяти в поисках свободной области подходящего размера.
- Если область имеет точно такой размер, как запрашивается, пометить найденную область как занятую и вернуть указатель на начало области памяти.
- Если область имеет больший размер, разделить ее на части, одна из которых будет занята (выделена), а другая останется свободной.
- Если область не найдена, вернуть нулевой указатель.

❖ Алгоритм работы функции `free`.

- Просмотреть список занятых/свободных областей памяти в поисках указанной области.
- Пометить найденную область как свободную.
- Если освобожденная область вплотную граничит со свободной областью с какой-либо из двух сторон, то объединить их в единую область большего размера.

❖ Какие гарантии относительно выделенного блока памяти даются программисту?

Каждый новый выделенный блок не будет совпадать или быть частью ранее выделенных, но не освобожденных блоков памяти.

Объем выделенной памяти будет не меньше запрошенного (но может быть больше).

Выделенная область доступна для чтения и записи данных.

❖ Что значит «освободить блок памяти» с точки зрения функции `free`?

«Освободить блок памяти» с точки зрения функции `free` значит пометить его как освобожденный.

❖ Преимущества и недостатки использования динамической памяти.

Преимущества:

- Размер выделяемой области определяется во время выполнения программы. Возможность выделить необходимое количество памяти.
- Место выделения памяти не влияет на время её жизни: область память будет существовать до момента освобождения программистом.
- Размер динамической памяти на несколько порядков больше статической, создаваемой на стеке.

Недостатки:

- Ручное управление временем жизни. Необходимость вручную выделять, проверять успешность и освобождать память.

❖ Что такое фрагментация памяти?

Фрагментация памяти — это состояние памяти, при котором свободные области сильно разрозненны и непригодны для записи больших объектов, хотя суммарного объема свободной памяти для этого достаточно.

❖ Выравнивание блока памяти, выделенного динамически.

Для хранения произвольных объектов блок должен быть правильно выровнен[4]. В каждой системе есть самый «требовательный» тип

[4] https://en.wikipedia.org/wiki/Data_structure_alignment

данных — если элемент этого типа можно поместить по некоторому адресу, то любые другие элементы тоже можно поместить туда.

❖ Что такое variable length array?

Variable length array (англ. массив переменной длины) — это массив, размер которого может задавать целочисленное выражение (переменная) во время выполнения программы.

❖ Чем отличается статический массив от variable length array?

Размер статического массива должен быть известен на этапе компиляции, размер массива переменной длины — нет.

Статический массив можно проинициализировать, массив переменной длины — нет.

К массивам переменной длины неприменимы классы памяти `static` и `extern`.

❖ Какую операцию языка Си пришлось реализовывать по-другому (не как для встроенных типов) специально для variable length array?

По другому пришлось реализовывать операцию `sizeof`, которая для встроенных типов работает во время компиляции, а для variable length array — во время выполнения программы (размер массива переменной длины не известен на этапе компиляции).

❖ Особенности использования variable length array.

- Длина такого массива вычисляется во время выполнения программы, а не во время компиляции.
- Память под элементы массива выделяется на стеке.
- Массивы переменного размера нельзя инициализировать при определении.

❖ Справедлива ли для variable length array адресная арифметика?

Да, справедлива.

❖ Как вы думаете, почему variable length array нельзя инициализировать?

Массив переменной длины нельзя инициализировать, так как во время выполнения его размер может оказаться меньше, чем при инициализации, вследствие чего произойдет переполнение стека.

❖ Для чего используется variable length array? Приведите примеры.

Массивы переменной длины облегчают описание заголовков функций, которые обрабатывают массивы.

```
void print(int n, int m, int a[n][m])
// но не void print(int a[n][m], int n, int m)
{
    …
}
```

❖ В какой области и «кем» выделяется память под массив переменной длины?

Память под массив переменной длины выделяется компилятором на стеке.

❖ Функция `alloca`.

```
#include <alloca.h>
void *alloca(size_t size);
```

Функция `alloca` выделяет область памяти размером `size` байт на стеке. Функция возвращает указатель на начало выделенной области. Эта область автоматически освобождается, когда функция, которая вызвала `alloca`, возвращает управления вызывающей стороне.

Если выделение вызывает переполнение стека, поведение программы не определено.

❖ `alloca` *vs* VLA.

Функция `alloca` является нестандартной (не входит даже в POSIX), в то время как VLA входят в стандарт C99.

```
void foo(int size)
{
    …
    while (b)
    {
        char tmp[size];
        // Будет уничтожен при выходе из области видимости
        // (в условии цикла).
        char *tmp2 = alloca(size);
        // Будет существовать на протяжении времени
        // выполнения всей функции. Опасность переполнения
        // стека: на каждой итерации память будет выделяться
        // снова и снова.
        …
    }
}
```

Функции с переменным числом параметров

❖ Можно ли реализовать в языке Си функцию со следующим прототипом: `int f(…)`? Почему?

Стандарт требует как минимум один фиксированный аргумент, чтобы передать его функции `va_start`[5].

[5] http://c-faq.com/varargs/onefixedarg.html

Без фиксированного параметра нельзя понять, где в стеке начинаются переданные такой функции аргументы[6].

❖ Покажите идею реализации функций с переменным числом параметров.

Использование возможностей языка (также возможен подход, когда единственным явным параметром является первое число, а конец списка параметров обозначается нейтральным значением — в данном случае нулем):

```
#include <stdio.h>

double avg(int n, …)
{
    int *p_i = &n;
    double *p_d = (double*) (p_i + 1);
    double sum = 0.0;

    if (!n)
        return 0;

    for (int i = 0; i < n; i++, p_d++)
        sum += *p_d;

    return sum / n;
}
```

[6] https://rsdn.org/forum/cpp/418970.1

Использование стандартной библиотеки:

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

double avg(int n, …)
{
    va_list vl;
    double sum = 0, num;

    if (!n)
        return 0.0;

    va_start(vl, n);

    for (int i = 0; i < n; i++)
    {
        num = va_arg(vl, double);

        printf("%f\n", num);

        sum += num;
    }

    va_end(vl);

    return sum / n;
}
```

- Почему для реализации функций с переменным числом параметров нужно использовать возможности стандартной библиотеки?

Поведение программы, не использующей стандартную библиотеку для реализации функций с переменным числом параметров, сильно зависит от платформы и компилятора, так как размер параметров и их расположение в памяти разнится от системы к системе.

❖ Опишите подход к реализации функций с переменным числом параметров с использованием стандартной библиотеки. Какой заголовочный файл стандартной библиотеки нужно использовать? Какие типы и макросы из этого файла вам понадобятся? Для чего?

Для реализации функций с переменным числом параметром с использованием стандартной библиотеки понадобится заголовочный файл `stdarg.h`. В нем определен тип `va_list` и следующие макросы:

```
[type] va_arg(va_list ap, [type]);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, [parmN]);
```

Макрос `va_start` служит для инициализации списка переменных аргументов и должен иметь соответствующий вызов `va_end`. Макрос `va_arg` используется для получения доступа к очередному аргументу, а `va_copy` — для копирования объектов типа `va_list`.

❖ Какая особенность языка Си упрощает реализацию функций (с точки зрения компилятора) с переменным числом параметров?

Соглашение о вызове cdecl позволяет понять, где и как расположены параметры, переданные в функцию.

❖ Почему при вызове `va_arg(argp, short int)` (или `va_arg(argp, float)`) выдается предупреждение?

В функциях с переменным числом параметров `float` «расширяется» до `double`, `char` и `short int` — до `int`[7]. Это поведение в большой мере обусловлено историей языка Си[8] и является стандартным[9].

---

[7] http://c-faq.com/varargs/float.html

[8] https://stackoverflow.com/a/1256122

[9] http://givi.olnd.ru/wclr/05_typecast.html

❖ Какая «опасность» существует при использовании функций с переменным числом параметров?

Некорректное использование функций с переменным числом параметров может привести к выходу за границы стека, его порче и неопределенному поведению[10].

❖ Как написать функцию, которая получает строку форматирования и переменное число параметров (как функция `printf`), и передает эти данные функции `printf`?

Для этого можно использовать функции `vprintf`, `vfprintf` или `vsprintf`. Они похожи на свои аналоги `printf`, `fprintf` и `sprintf`, за исключением того, что вместо списка аргументов переменной длины они принимают единственный указатель типа `va_list`.

В качестве примера можно привести функцию `error`, которая печатает сообщение об ошибке:

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

void error(const char *fmt, …)
{
    va_list argp;
    fprintf(stderr, "error: ");
    va_start(argp, fmt);
    vfprintf(stderr, fmt, argp);
    va_end(argp);
    fprintf(stderr, "\n");
}
```

Аналогично можно поступить с функцией `scanf` и функциями `vscanf`, `vfscanf` и `vsscanf`.

---

[10] https://en.wikipedia.org/wiki/Variadic_function#Overview

Препроцессор. inline-функции

❖ Что делает препроцессор? В какой момент в процессе получения исполняемого файла вызывается препроцессор?

Препроцессор выполняет 4 основные функции:

- удаление комментариев;
- включение файлов (директива `#include`);
- текстовые замены (директива `#define`);
- условная компиляция (например, include guard).

Препроцессор вызывается в самом начале процесса получения исполняемого файла. Посмотреть результат работы препроцессора GCC можно при помощи флага `-E`:

```
gcc -E main.c > main.i
```

❖ На какие группы можно разделить директивы препроцессора?

Все директивы препроцессора можно разделить на 3 группы:

- Макроопределения (макросы): `#define`, `#undef`;
- Директива включения файлов: `#include`;
- Директивы условной компиляции: `#if`, `#ifdef`, `#endif` и др.

Остальные директивы (`#pragma`, `#error`, `#line` и др.) используются реже.

❖ Какие правила справедливы для всех директив препроцессора?

Директивы всегда начинаются с символа «#».

Директивы могут появляться в любом месте программы.

Любое количество пробельных символов может разделять лексемы в директиве:

```
#  define   N   1000
```

Директива заканчивается на символе «\n»:

```
#define DISK_CAPACITY  (SIDES *              \
                        TRACKS_PER_SIDE *    \
                        SECTORS_PER_TRACK *  \
                        BYTES_PER_SECTOR)
```

❖ Что такое простой макрос? Как такой макрос обрабатывается препроцессором? Приведите примеры.

Простой макрос (object-like macro — C99):

```
#define идентификатор список-замены
```

Когда препроцессор встречает простой макрос, он заменяет его идентификатор на список-замены.

Примеры простых макросов:

```
#define PI 3.14
#define EOS '\0'
#define MEM_ERR "Memory allocation error."
```

❖ Для чего используются простые макросы?

Простые макросы используются

- в качестве имен для числовых, символьных и строковых констант;
- незначительного изменения синтаксиса языка:
  ```
  #define BEGIN {
  #define END }
  #define INF_LOOP for( ; ; )
  ```

- переименования типов:
  `#define BOOL int`
- управления условной компиляцией.

❖ Что такое макрос с параметрами? Как такой макрос обрабатывается препроцессором? Приведите примеры.

Макрос с параметрами (function-like macro — C99):

```
#define идентификатор(x1, x2, ... , xn) список-замены
```

Между именем макроса и открывающей скобкой не должно быть пробела. Список параметров может быть пустым:

```
#define getchar() fgetc(stdin)
```

Когда препроцессор встречает макрос с параметрами, он сначала заменяет формальные параметры фактическими, а затем подставляет полученное выражение на место макроса в коде.

Примеры:

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define IS_EVEN(x) ((x) % 2 == 0)
```

❖ Макросы с параметрами *vs* функции: преимущества и недостатки.

Преимущества:

- Программа может работать немного быстрее.
- Макросы «универсальны».

Недостатки:

- Скомпилированный код становится больше:
  `n = MAX(i, MAX(j, k));`
- Типы аргументов не проверяются.
- Нельзя объявить указатель на макрос.

- Макрос может вычислять аргументы несколько раз:
  `n = MAX(i++, j);`

❖ Макросы с переменным числом параметров. Приведите примеры.

С появлением стандарта C99 стало возможным использование макросов с переменным числом параметров:

```
#ifndef NDEBUG
#define DBG_PRINT(s, …) printf(s, __VA_ARGS__)
#else
#define DBG_PRINT(s, …) ((void) 0)
#endif
```

Определить макрос `NDEBUG` при компиляции можно при помощи флага `-D`:

```
gcc -std=c99 … -D NDEBUG
```

❖ Какими общими особенностями/свойствами обладают все макросы?

Список-замены макроса может содержать другие макросы.

Препроцессор заменяет только целые лексемы, а не их части.

Определение макроса остается «известным» до конца файла, в котором этот макрос объявляется.

Макрос не может быть объявлен дважды, если эти объявления не тождественны.

Макрос может быть «разопределен» с помощью директивы `#undef`.

❖ Объясните правила использования скобок внутри макросов. Приведите примеры.

Если список-замены содержит операции, он должен быть заключен в скобки.

Если у макроса есть параметры, они должны быть заключены в скобки в списке-замены.

Эти правила обеспечивают правильный порядок выполнения кода, заключенного в макрос.

```
#define TWO_PI 2 * 3.14

f = 360.0 / TWO_PI;
// f = 360.0 / 2 * 3.14;

#define SCALE(x) (x * 10)

j = SCALE(i + 1);
// j = (i + 1 * 10);
```

❖ Какие подходы к написанию «длинных» макросов вы знаете? Опишите их преимущества и недостатки. Приведите примеры.

Первый подход — заключение списка-замены в фигурные скобки:

```
#define ECHO(s) { gets(s); puts(s); }
```

Недостаток:

```
if (echo_flag)
    ECHO(str); // { gets(s); puts(s); };
               // точка с запятой ⇒ ошибка компиляции
else
    gets(str);
```

Второй подход — заключение списка-замены в круглые скобки и использование запятой:

```
#define ECHO(s) (gets(s), puts(s))
```

Недостаток: невозможность использования, например, условных операторов.

Третий подход — использование символов \ и цикла с постпроверкой, выполняющегося один раз:

```
#define ECHO(s) \
do              \
{               \
    gets(s);    \
    puts(s);    \
}               \
while(0)        \
```

❖ Какие предопределенные макросы вы знаете? Для чего эти макросы могут использоваться?

Существуют следующие предопределенные макросы:

- `__LINE__` заменяется номером текущей строки (десятичная константа).
- `__FILE__` заменяется именем компилируемого файла.
- `__DATE__` заменяется датой компиляции.
- `__TIME__` заменяется временем компиляции.

Эти идентификаторы нельзя переопределять или отменять директивой `#undef`.

`__func__` заменяется именем функции (как строка). Работает только с GCC и C99, не является макросом.

❖ Для чего используется условная компиляция? Приведите примеры.

Использование условной компиляции:

- Программа, которая должна собираться различными компиляторами.

- Начальное значение макросов:
```
#ifndef BUF_SIZE
#define BUF_SIZE 256
#endif
```
- Программа, которая должна работать под несколькими операционными системами:
```
#if defined(OS_WIN)
…
#elif defined(OS_LIN)
…
#elif defined(OS_MAC)
…
#endif
```
- Временное выключение кода (может пригодиться для комментирования кода с многострочными комментариями):
```
#if 0
for (int i = 0; i < n; i++)
    a[i] = 0.0;
#endif
```

❖ Директива `#if` *vs* директива `#ifdef`.

Директива `#ifdef` может проверять только логические переменные, т.е. только два значения. Директива `#if` позволяет проверять любые условия. Ошибка в идентификаторе `#ifdef` не вызовет ошибку компиляции, а значит понять, включилась необходимая функциональность или нет, будет не так просто.

```
#ifdef FEATURE_X
…
#endif
```

```
#if INTERFACE_VERSION == 0
…
#elif INTERFACE_VERSION == 1
…
#else
…
#endif
```

❖ Операция #. Примеры использования.

Операция # конвертирует аргумент макроса в строковый литерал.

```
#define PRINT_INT(n) printf(#n " = %d\n", (n))

#define TEST(condition, …) ((condition) ?         \
        printf("Passed test %s\n", #condition) :  \
                                printf(__VA_ARGS__))

PRINT_INT(i / j);
// printf("i/j" " = %d", i/j);

TEST(voltage ⩽ max_voltage, "Voltage %d exceed %d",
                                      voltage, max_voltage);
```

❖ Операция ##. Примеры использования.

Операция ## объединяет две лексемы в одну.

```
#define MK_ID(n) i##n

int MK_ID(1), MK_ID(2);
// int i1, i2;
```

❖ Особенности использования операций.

Заключать параметры, к которым применяются операции, в круглые скобки нельзя — возникнет ошибка компиляции.

❖ Директива `#error`. Примеры использования.

Когда препроцессор встречает директиву `#error`, он выводит сообщение об ошибке и прерывает компиляцию. Вид сообщения зависит от конкретного компилятора. Пример:

```
#if defined(OS_WIN)
…
#elif defined(OS_LIN)
…
#elif defined(OS_MAC)
…
#else
#error Unsupported OS!
#endif
```

❖ Директива `#pragma` (на примере `once` и `pack`). Примеры использования.

Директива `#pragma` позволяет добиться от компилятора специфичного поведения. Является стандартной, но ее реализация в C99 не описана.

Если при просмотре заголовочного файла препроцессор встречает `#pragma once`, то этот файл будет им дальше игнорироваться. Директива может использоваться для защиты от множественных включений (include guard), однако является менее переносимой альтернативой `#ifndef`.

`#pragma pack(размер)` заставляет компилятор упаковывать члены структуры с выравниванием указанного размера. Использование этой директивы может замедлить работу программы, а также привести к ошибкам на некоторых платформах[11].

[11] https://stackoverflow.com/a/3318475

❖ Ключевое слово `inline`.

`inline` — это пожелание компилятору заменить вызовы функции последовательной вставкой кода самой функции.

inline-функции по-другому называют встраиваемыми или подставляемыми.

В C99 `inline` означает, что определение функции предоставляется только для подстановки и где-то в программе должно быть другое такое же определение этой же функции.

❖ Назовите основную причину, по которой ключевое слово `inline` было добавлено в язык Си.

До появления ключевого слова `inline` использовался следующий прием: определение функции выносили в заголовочный файл, добавляя класс памяти `static`. Так компилятор имел возможность заменить вызов функции на её тело, однако делал это не всегда, что могло привести к неконтролируемому росту размера исполняемого файла.

Ключевое слово `inline` было добавлено, чтобы исправить эту ситуацию.

❖ Подходы к реализации ключевого слова `inline` компилятором. Проанализируйте их недостатки.

Если функция объявлена как `static inline` и её адрес нигде не используется, компилятор заменяет вызов функции её телом. Недостаток: необходимость в нескольких определениях одной функции.

Если функция объявлена как `inline` (без `static`), компилятор предполагает, что к ней могут быть вызовы из других файлов исходного кода. Поскольку такая функция может быть определена только один раз в программе (из-за внешнего связывания), то её вызов не может быть заменен её телом. Недостаток: ключевое слово используется, однако пользы от него нет.

Если функция объявлена как `extern inline`, компилятор считает, что это определение используется только для встраивания. В этом случае все её вызовы будут заменены на её тело, даже если где-то в программе используется адрес этой функции (поведение будет таким же, как и в ситуации, когда функция была объявлена, но не определена).

Комбинация `inline` и `extern` имеет эффект макроса. Одним из способов использования этой комбинации является добавление объявления функции как `extern inline` в заголовочный файл и обычного объявления (без классов памяти) в файл исходного кода.

Подробнее: https://gcc.gnu.org/onlinedocs/gcc/Inline.html

- ❖ В чем разница между использованием <...> и "..." в директиве `include`?

<...> используется для включения стандартных или системных заголовочных файлов, а "..." — для локальных. Отличаются пути поиска, по которым препроцессор будет искать необходимые файлы[12].

- ❖ Можно ли операцию `sizeof` использовать в директивах препроцессора? Почему?

Нет, потому что препроцессирование происходит до разбора типов. Иначе говоря, ни переменные, ни их типы не известны препроцессору на этапе раскрытия макросов. Если необходимо, следует использовать значения из `limits.h`.

[12] http://c-faq.com/cpp/inclkinds.html

Библиотеки

❖ Что такое библиотека?

Библиотека — это набор связанных функций. Фактически, это набор объектных файлов, оформленных специальных образом.

❖ Какие функции обычно выносят в библиотеку?

В библиотеку обычно выносят часто используемые функции, связанные некоторым признаком. Пример: функции, обрабатывающие определенный тип данных.

❖ В каком виде распространяются библиотеки? Что обычно входит в их состав?

Библиотеки распространяются в скомпилированном (двоичном) виде. Библиотеки меняются редко — нет причин перекомпилировать каждый раз. Двоичный код предотвращает доступ к исходному коду.

Библиотека включает в себя

- заголовочный файл;
- откомпилированный файл самой библиотеки.

❖ Какие виды библиотек вы знаете?

Библиотеки делятся на статические, которые включаются в исполняемый файл программы во время компоновки, и динамические (разделяемые), загружаемые программой во время выполнения.

❖ Преимущества и недостатки, которые есть у статических/динамических библиотек.

Преимущества статических библиотек:

- Исполняемый файл включает в себя все необходимое. Программа может работать на любой поддерживаемой системе вне зависимости от того, установлена на ней библиотека или нет.
- Не возникает проблем с использованием не той версии библиотеки.

Недостатки статических библиотек:

- «Размер». Если несколько приложений на одной системе используют одну и ту же библиотеку, то память используется нерационально.
- При обновлении библиотеки программу нужно пересобрать.

Преимущества динамических библиотек:

- Несколько программ могут «разделять» одну библиотеку.
- Меньший размер приложения (по сравнению с приложением со статической библиотекой).
- Средство реализации плагинов.
- Модернизация библиотеки не требует перекомпиляции программы.
- Могут использоваться программами на разных языках.

Недостатки динамических библиотек:

- Требуется наличие библиотеки на компьютере.
- Версионность библиотек. Необходимость обеспечивать обратную совместимость.

❖ Как собрать статическую библиотеку?

Компиляция

```
gcc -std=c99 -Wall -Werror -c arr_lib.c
```

Упаковка (по сути, создание архива из объектных файлов). Флаг `c` отвечает за создание файла библиотеки, если он еще не существует, флаг `r` позволяет обновлять функции в библиотеке, если они были изменены.

```
ar rc libarr.a arr_lib.o
```

Индексирование. Позволяет ускорить работу компоновщика, если используется статическая библиотека с большим количеством функций.

```
ranlib libarr.a
```

❖ Нужно ли «оформлять» каким-то специальным образом функции, которые входят в состав статической библиотеки?

Нет, не нужно.

❖ Как собрать приложение, которое использует статическую библиотеку?

Существует два способа собрать приложение, использующее статическую библиотеку. Первый:

```
gcc -std=c99 -Wall -Werror main.c libarr.a -o test.exe
```

Второй:

```
gcc -std=c99 -Wall -Werror main.c -L. -larr -o test.exe
```

❖ Нужно ли «оформлять» каким-то специальным образом исходный код приложения, которое использует статическую библиотеку?

Нет, не нужно.

❖ Как собрать динамическую библиотеку (Windows/Linux)?

Компиляция (Windows):

```
gcc -std=c99 -Wall -Werror -c arr_lib.c
```

Компоновка (Windows):

```
gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll
```

Компиляция (Linux):

```
gcc -std=c99 -Wall -Werror -fPIC -c arr_lib.c
```

Компоновка (Linux):

```
gcc -shared arr_lib.o -o libarr.so
```

Обновление путей (Linux):

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

❖ Нужно ли «оформлять» каким-то специальным образом функции, которые входят в состав динамической библиотеки (Windows/Linux)?

С точки зрения Windows все функции библиотеки делятся на те, которые доступны «наружу» и те, которые предназначены для внутреннего использования и стороннему разработчику не видны. Для этого используются следующие конструкции:

```
// arr_lib.h
__declspec(dllexport) void __cdecl arr_print(…);
// функция доступна «наружу»
```

```
// arr_lib.c
__declspec(dllexport) void __cdecl arr_print(…)
{
    …
}
```

В случае с Linux «оформлять» каким-то специальным образом функции не нужно.

❖ Какие способы компоновки приложения с динамической библиотекой вы знаете? Назовите их преимущества и недостатки.

Выделяют динамическую компоновку и динамическую загрузку.

Преимущества динамической компоновки:

- Исходный код программы изменяется не сильно.

Недостатки динамической компоновки:

- Доверие компоновщику. Риск возникновения ошибок.
- Невозможность запустить программу при отсутствии необходимой библиотеки (программа сразу же «упадет»).

Преимущества динамической загрузки:

- Возможность контроля над процессом.
- Программа может работать, даже если не удалось подключить какую-нибудь библиотеку. Ошибка произойдет при попытке использовать функции из нее, однако у разработчика есть возможность показать соответствующее сообщение.

Недостатки динамической загрузки:

- Бóльшая, по сравнению с динамической компоновкой, сложность подключения.

❖ Что такое динамическая компоновка?

Динамическая компоновка — это способ использования динамических библиотек, при котором часть функций поиска и загрузки библиотеки делегируется компилятору.

❖ Что такое динамическая загрузка (Windows/Linux)?

Динамическая загрузка — это способ использования динамических библиотек, при котором для подключения библиотеки во время выполнения программы используется интерфейс операционной системы (API).

В случае с Windows необходимо подключить заголовочный файл `windows.h`, определяющий прототипы следующих функций:

- `HMODULE LoadLibrary(LPCSTR)`. С помощью этой функции загружается динамическая библиотека. Единственный параметр — название загружаемой библиотеки (абсолютный путь или просто имя). В случае успешной загрузки возвращает описатель библиотеки типа `HMODULE`.
- `FARPROC GetProcAddress(HMODULE, LPCSTR)`. Позволяет найти нужные функции в библиотеке. Получает описатель библиотеки и имя необходимой функции.
- `FreeLibrary(HMODULE)`. Выгружает динамическую библиотеку, однако делает это не сразу: другие запущенные программы могут от нее зависеть.

Для динамической загрузки на Linux понадобится заголовочный файл `dlfcn.h`, который определяет следующие функции:

- `void *dlopen(const char *, int)`. Позволяет загрузить динамическую библиотеку. Принимает её название и режим релокации[13]. Возвращает описатель библиотеки как указатель на `void`.

[13] https://pubs.opengroup.org/onlinepubs/7908799/xsh/dlfcn.h.html

- `void *dlsym(void *, const char *)`. Позволяет загрузить нужную функцию библиотеки. Принимает описатель библиотеки.
- `int dlclose(void *)`. Выгружает динамическую библиотеку.
- `char *dlerror(void)`. Если произошла ошибка, данная функция вернет указатель на строку с сообщением.

❖ Нужно ли «оформлять» каким-то специальным образом исходный код приложения, которое использует динамическую библиотеку (Windows/Linux)?

Для использования функций из динамической библиотеки на Windows необходимо добавить следующую конструкцию:

```
__declspec(dllimport) void __cdecl arr_print(…);
```

В случае с Linux «оформлять» каким-то специальным образом исходный код приложения не нужно.

❖ Особенности реализации функций, использующих динамическое выделение памяти, в динамических библиотеках.

Для реализации функций, использующих динамическое выделение памяти, есть два подхода:

- Делегировать операцию выделения и освобождения памяти вызывающей стороне. Функции, использующие этот подход, должны предоставить возможность определить объем памяти, который необходимо выделить для их работы.
- Реализация дополнительных функций, освобождающих память.

❖ Ключи `-I`, `-l`, `-L` компилятора gcc.

`-I ./folder/` позволяет указать компилятору путь до заголовочных файлов, используемых программой.

`-llibrary` или `-l library` используется для подключения библиотеки `library` при компоновке. Второй вариант использования (через пробел) не рекомендуется и реализован только для соответствия стандарту POSIX.

Флаг `-L` позволяет указать путь до директории с файлами библиотек, которые использует программа. Пример использования:

```
gcc … -L./libs/
```

❖ Проблемы использования динамической библиотеки, реализованной на одном языке программирования, и приложения, реализованного на другом языке программирования.

Несоответствие «философий» и типов двух языков. Например, использование динамической библиотеки на Си с указателями и программы на Python, где указателей нет.

Работа с ресурсами. Необходимо определить, например, кто будет выделять и освобождать память.

❖ Модуль ctypes. Загрузка библиотеки. Представление стандартных типов языка Си. Импорт функций из библиотеки. Проблемы, которые при этом возникают.

Модуль ctypes позволяет использовать динамические библиотеки на языке Си в программах на языке Python.

Загрузка библиотеки:

```
lib = ctypes.CDLL('name')
```

Классов для работы с библиотеками в модуле ctypes несколько:

- `CDLL` (cdecl и возвращаемое значение `int`)
- `OleDLL` (stdcall и возвращаемое значение `HRESULT`)
- `WinDLL` (stdcall и возвращаемое значение `int`)

Класс выбирается в зависимости от соглашения о вызовах, которое использует библиотека.

Получить доступ к стандартным типам языка Си можно следующим образом:

```
ctypes.c_int  # и так далее
```

Импорт функций:

```
add = lib.add
add.argtypes = (ctypes.c_int, ctypes.c_int)
add.restype = ctypes.c_int
```

Проблемы:

- Написание большого количества сигнатур для функций и, в зависимости от сложности функций, функций-оберток.
- Необходимость детально представлять внутреннее устройство типов Python и то, каким образом они могут быть преобразованы в типы Си.

❖ Написание модуля расширения для Python (основные шаги).

Подключение библиотеки `Python.h`.

Написание функций, принимающих два указателя на `PyObject` (для вызвавшего функцию объекта и кортежа аргументов) и возвращающих указатель на `PyObject`.

Преобразование объектов языка Python в типы языка Си при помощи функции `PyArg_ParseTuple` (управляется строкой форматирования):

```
int a, b;
if (!PyArg_ParseTuple(args, "ii", &a, &b))
    return NULL;
```

Обработка данных и преобразование результата обратно в `PyObject` при помощи функции `Py_BuildValue`:

```
return Py_BuildValue("i", a + b);
```

Добавить в конец модуля расширения таблицу методов `PyMethodDef` и структуру `PyModuleDef`, которая описывает модуль в целом и используется для его загрузки:

```
static PyMethodDef example_methods[] =
{
    {"add", py_add, METH_VARARGS, "Integer addition"},
    {"divide", py_divide, METH_VARARGS, "Integer division"},
    { NULL, NULL, 0, NULL} // обозначение конца массива
};

static struct PyModuleDef example_dll_module =
{
    PyModuleDef_HEAD_INIT,
    "example_dll", // Имя модуля
    "An example_dll module", // Строка для документации
    -1,
    example_methods // Таблица методов
};
```

Добавить функцию инициализации модуля:

```
PyMODINIT_FUNC PyInit_example_dll(void)
{
    return PyModule_Create(&example_dll_module);
}
```

Написать конфигурационный файл `setup.py`.

Бинарные операции. Битовые поля

❖ К каким типам в языке Си применимы битовые операции?

Битовые операции применимы только к целочисленным переменным. Обычно эти операции стараются выполнять над беззнаковыми целыми, чтобы

- повысить переносимость программы;
- не было путаницы с битом, который отвечает за знак.

❖ Особенности использования битовых операций со знаковыми целыми типами.

Поведение программы при работе со знаковым битом зависит от реализации.

❖ Какие битовые операции есть в языке Си? Приведите примеры.

Битовые операции позволяют оперировать отдельными битами или группами битов, из которых формируются байты (и машинные слова).

Язык Си поддерживает все битовые операции:

- `&` «и»
- `|` «или»
- `^` «исключающее или»
- `~` «дополнение»
- `>>` «сдвиг вправо»
- `<<` «сдвиг влево»

❖ Как установить указанный бит?

Для установления указанного бита к исходному числу применяется операция «и» с маской, в которой указанный бит равен единице, а остальные биты — нулю. Пример:

```
unsigned char a;
a = 0x00;    // 0000 0000
mask = 0x10; // 0001 0000
a |= mask;   // 0001 0000
```

Универсальный подход: `a |= (1 << n)`

❖ Как сбросить указанный бит?

Чтобы сбросить указанный бит, необходимо к исходному числу применить операцию «или» с маской, в которой указанный бит равен нулю, а все остальные биты — единице. Пример:

```
unsigned char a;
a = 0xFF;     // 1111 1111
mask = ~0x10; // 1110 1111
a &= mask;    // 1110 1111
```

Универсальный подход: `a &= ~(1 << n)`

❖ Как проверить, что указанный бит установлен?

Чтобы проверить, что указанный бит установлен, применяется операция «и» к исходному числу и маске, в которой в указанной позиции установлена единица, а во всех остальных — нули.

```
if (a & (1 << n))
```

❖ Как изменить значение указанного бита на противоположное?

Чтобы изменить значение указанного бита на противоположное, необходимо применить операцию «исключающее или» к исходному

числу и маске, в которой в указанной позиции установлена единица, а во всех остальных — нули.

```
inverted = number ^ (1 << n)
```

❖ Как установить сразу несколько бит?

Чтобы установить сразу несколько бит, необходимо сначала их «занулить» (применить операцию «и» к исходному числу и маске, в которой на нужных позициях находятся нули, а на остальных — единицы), а затем применить операцию «или» к исходному числу и маске, в которой на нужных позициях находятся устанавливаемые биты, а на остальных — единицы.

❖ Как получить значение нескольких бит?

Чтобы получить значение нескольких бит, необходимо применить операцию «и» к исходному числу и маске, в которой на нужных позициях находятся единицы, а на остальных — нули.

Если необходимо получить значение бит, которые не находятся у правой границы числа, следует воспользоваться сдвигом вправо.

❖ С помощью какой битовой операции можно разделить целое число на 2^n?

Разделить целое число на 2^n можно при помощи битовой операции «сдвиг вправо» на n позиций.

❖ С помощью какой битовой операции можно умножить целое число на 2^n?

Умножить целое число на 2^n можно при помощи битовой операции «сдвиг влево» на n позиций.

❖ Битовые операции *vs* логические операции.

Операнды битовых операций рассматриваются как числа, логических — как истина или ложь (единица или ноль).

Результатом битовых операций может быть любое число, в то время как логические операции возвращают только единицу или ноль (для истины и лжи соответственно).

```
8 && 3 // 1
8 & 3 // 0
```

❖ Что такое битовое поле?

Битовое поле — это особый тип структуры, определяющей, какую длину имеет каждый член в битах.

Стандартный вид объявления битовых полей:

```
struct имя_структуры
{
   тип имя1: длина;
   тип имя2: длина;
   …
   тип имяN: длина;
};
```

Битовые поля должны объявляться как целые, `unsigned` или `signed`.

❖ Преимущества и недостатки битовых полей по сравнению с битовыми операциями.

Преимущества:

- Наглядность.
- Абстракция над двоичным представлением числа.

Недостатки:

- Переносимость (стандарт не накладывает строгих ограничений на хранение битовых полей).
- Влияние выравнивания и порядка бит (англ. endianness) на расположение битовых полей в памяти (C99 6.7.2.1 #10).
- «Знаковость» типа `int` в битовом поле зависит от реализации.

❖ Что задает значение `CHAR_BIT`? В каком стандартном заголовочном файле его можно найти?

Значение `CHAR_BIT` задает количество бит наименьшего объекта, не являющегося битовым полем (C99 5.2.4.2.1 #1). Определение этого значения находится в заголовочном файле `limits.h`.

❖ Существуют ли значения `SHORT_INT_BIT`, `INT_BIT` и т.д.? Почему?

Указанные значения не существуют, так как они могут быть вычислены при помощи `CHAR_BIT`:

```
sizeof(short int) * CHAR_BIT // SHORT_INT_BIT
sizeof(int) * CHAR_BIT // INT_BIT
```

❖ Напишите функцию, которая использует битовые операции для вывода числа в двоичной системе счисления.

```
void print_binary(unsigned int number)
{
    if (number > 1)
        print_binary(number >> 1);

    printf("%d", number & 1);
}
```

❖ «Упакуйте» четыре символа в беззнаковое целое. Длина беззнакового целого равна 4.

```
unsigned char i = 'i'; // 105 ⟷ 01101001
unsigned char d = 'd'; // 100 ⟷ 01100100
unsigned char e = 'e'; // 101 ⟷ 01100101
unsigned char a = 'a'; // 97  ⟷ 01100001

unsigned int result = a;

result ⟸ 8;
result ⊨ e;
result ⟸ 8;
result ⊨ d;
result ⟸ 8;
result ⊨ i; // 01100001011001010110010001101001 ⟷
            // 1634034793 (Big-endian)
```

❖ «Распакуйте» беззнаковое целое число в четыре символа. Длина беззнакового целого равна 4.

```
unsigned int result = 1634034793;

for (size_t i = 0; i < 4; i++, result ⟹ 8)
    printf("%c", result & 0xFF);
```

❖ Напишите функцию для циклического сдвига значения целочисленной величины на n позиций вправо/влево.

```
uint32_t shift_left(uint32_t number, unsigned int n)
{
    unsigned int mask = CHAR_BIT * sizeof(number) - 1;
    n &= mask;
    return (number << n) | (number >> (-n & mask));
}
```

```
uint32_t shift_right(uint32_t number, unsigned int n)
{
    unsigned int mask = CHAR_BIT * sizeof(number) - 1;
    n &= mask;
    return (number >> n) | (number << (-n & mask));
}
```

❖ Запишите в одно беззнаковое целое число (длина беззнакового целого равна 4 байт) структуру, содержащую данные о файле аудиозаписи: жанр (народная, классическая, кантри, джаз, шансон, бардовская, поп, рэп, рок-н-ролл, рок, электронная, экзотическая, церковная, военная, детская, аудиокнига), стерео/моно, длительность в секундах (от 1 до 8192), размер файла в Кбайт (от 1 до 16384). Решите задачу несколькими способами.

Для решения задачи понадобится 4 бита под жанр, 1 бит под канальность, 13 бит под длительность и 14 бит под размер файла. Всего потребуется 32 бита или 4 байта, что соответствует условию.

```
struct audio_file
{
    unsigned int genre : 4;
    unsigned int channel : 1;
    unsigned int duration : 13;
    unsigned int size : 14;
};

struct audio_file info = { … };
int result = info.size << 18 | info.duration << 5 |
             info.channel << 4 | info.genre;
```

Неопределенное поведение

❖ Что такое побочный эффект?

Побочные эффекты (англ. side effects) — это любые действия работающей программы, изменяющие среду выполнения (англ. execution environment).

❖ Какие выражения стандарт C99 относит к выражениям с побочным эффектом?

К выражениям с побочным эффектом относятся:

- Модификация данных.
- Обращение к переменным, объявленным как `volatile`.
- Вызов системной функции, которая производит побочные эффекты (например, файловый ввод или вывод).
- Вызов функций, выполняющих любое из вышеперечисленных действий.

❖ Почему порядок вычисления подвыражений в языке Си не определен?

Порядок вычисления подвыражений в языке Си не определен, чтобы компиляторы могли оптимизировать результирующий код.

❖ Порядок вычисления каких выражений в языке Си определен?

В языке Си определен порядок вычисления логических выражений, тернарного оператора и оператора «запятая». Все они вычисляются слева направо.

❖ Что такое точка следования?

Точка следования — это точка в программе, в которой программист знает какие выражения (или подвыражения) уже вычислены, а какие выражения (или подвыражения) — еще нет.

❖ Какие точки следования выделяет стандарт C99?

Определены следующие точки следования:

- Между вычислением левого и правого операндов в операциях `&&`, `||` и `,`:
  ```
  *p++ != 0 && *q++ != 0
  ```
- Между вычислением первого и второго или третьего операндов в тернарной операции:
  ```
  a = (*p++) ? (*p++) : 0;
  ```
- В конце полного выражения.
  ```
  a = b;
  if ()
  switch ()
  while ()
  do {} while ()
  for (x; y; z)
  return x
  ```
- Перед входом в вызываемую функцию. Порядок, в котором вычисляются аргументы не определен, но эта точка следования гарантирует, что все ее побочные эффекты проявятся на момент входа в функцию;
- В объявлении с инициализацией на момент завершения вычисления инициализирующего значения:
  ```
  int a = (1 + i++);
  ```

❖ Почему необходимо избегать выражений, которые дают разный результат в зависимости от порядка их вычисления?

Такие выражения необходимо избегать, так как нельзя предугадать, какой результат будет использован программой.

❖ Какие виды «неопределенного» поведения есть в языке Си?

В языке Си выделяют 3 вида «неопределенного» поведения:

- Unspecified (англ. неспецифицированное). Стандарт предлагает несколько вариантов на выбор. Компилятор может реализовать любой вариант. При этом на вход компилятора подается корректная программа.
- Implementation-defined (англ. поведение, зависящее от реализации). Похоже на неспецифицированное поведение, но в документации к компилятору должно быть указано, какое поведение реализовано.
- Undefined (англ. неопределенное). Такое поведение возникает как следствие неправильно написанной программы или некорректных данных. Стандарт ничего не гарантирует.

❖ Почему «неопределенное» поведение присутствует в языке Си?

«Неопределенное» поведение присутствует в языке Си, чтобы

- освободить разработчиков компиляторов от необходимости обнаруживать ошибки, которые трудно диагностировать;
- избежать предпочтения одной стратегии реализации другой;
- отметить области для расширения языка (англ. language extension).

❖ Какой из видов «неопределенного» поведения является самым опасным? Чем он опасен?

Самым опасным видом является неопределенное поведение, так как при его возникновении нельзя предсказать, что произойдет.

❖ Как бороться с неопределенным поведением?

Включайте все предупреждения компилятора, внимательно читайте их.

Используйте возможности компилятора (`-ftrapv`).

Используйте несколько компиляторов.

Используйте статические анализаторы кода: clang, cppcheck, PVS-Studio.

Используйте инструменты: Valgrind, Dr. Memory и др.

Используйте утверждения.

❖ Приведите примеры неопределенного поведения.

Использование переменной до её инициализации. Неопределённое поведение возникает при попытке обращения к переменной.

Переменная несколько раз изменяется в рамках одной точки следования. Часто в качестве канонического примера приводят выражение `i = i++`, в котором происходит присваивание переменной `i` и её же инкремент.

Переполнение знаковых целых типов (C99 3.4.3 #3).

Выход за границы массива.

Использование «диких» указателей.

❖ Приведите примеры поведения, зависящего от реализации.

Результат `x % y`, где `x` и `y` целые, а `y` отрицательное, может быть как положительным, так и отрицательным.

Сдвиг вправо бита, отвечающего за знак целого числа (C99 3.4.1 #2).

❖ Приведите примеры неспецифицированного поведения.

Все аргументы функции должны быть вычислены до ее вызова, но они могут быть вычислены в любом порядке (C99 3.4.4 #2).

АТД

❖ Что такое модуль?

Модуль — функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом.

❖ Из каких частей состоит модуль? Какие требования предъявляются к этим частям?

Модуль состоит из двух частей: интерфейса и реализации.

Интерфейс описывает, что модуль делает. Он определяет идентификаторы, типы и подпрограммы, которые будут доступны коду, использующему этот модуль.

Реализация описывает, как модуль выполняет то, что предлагает интерфейс.

У модуля есть один интерфейс, но реализаций, удовлетворяющих этому интерфейсу, может быть несколько.

Часть кода, которая использует модуль, называют клиентом.

Клиент должен зависеть только от интерфейса, но не от деталей его реализации.

❖ Назовите преимущества модульной организации программы. Приведите примеры.

Преимущества:

- Абстракция (как средство борьбы со сложностью). Когда интерфейсы модулей согласованы, ответственность за реализацию каждого модуля делегируется определенному разработчику.
- Повторное использование. Модуль может быть использован в другой программе.
- Сопровождение. Можно заменить реализацию любого модуля, например, для улучшения производительности или переноса программы на другую платформу.

❖ Какие виды модулей вы знаете? Приведите примеры.

Набор данных. Набор связанных переменных и/или констант. В Си модули этого типа часто представляются только заголовочным файлом. Примеры: `float.h`, `limits.h`.

Библиотека. Набор связанных функций. Примеры: `stdlib.h`, `math.h`.

Абстрактный объект. Набор функций, который обрабатывает скрытые данные.

Абстрактный тип данных. Пример: `FILE`.

❖ Что такое тип данных?

Тип данных — это множество значений и операций над этими значениями.

❖ Что такое абстрактный тип данных?

Абстрактный тип данных — это интерфейс, который определяет тип данных и операции над этим типом. Тип данных называется

абстрактным, потому что интерфейс скрывает детали его представления и реализации.

❖ Какие требования выдвигаются к абстрактному типу данных?

Абстрактный тип данных должен скрывать детали реализации и предоставлять полный набор функций, позволяющих обрабатывать этот тип.

❖ Абстрактный объект *vs* абстрактный тип данных.

Абстрактный объект, в отличие от абстрактного типа данных, не определяет тип данных (он скрыт).

❖ Средства реализации модулей в языке Си.

В языке Си интерфейс описывается в заголовочном файле.

В заголовочном файле описываются макросы, типы, переменные и функции, которые клиент может использовать.

Клиент импортирует интерфейс с помощью директивы препроцессора `include`.

Реализация интерфейса в языке Си представляется одним или несколькими файлами с расширением `*.c`.

Реализация определяет переменные и функции, необходимые для обеспечения возможностей, описанных в интерфейсе.

Реализация обязательно должна включать файл описания интерфейса, чтобы гарантировать согласованность интерфейса и реализации.

❖ Что такое неполный тип данных в языке Си?

Стандарт Си описывает неполные типы как «типы, которые описывают объект, но не предоставляют информацию, нужную для определения его размера».

❖ Приведите примеры описания неполного типа данных. А кроме структур?

```
struct a;
int b[];
union c;
```

❖ Какие действия можно выполнять с неполным типом данных?

Допустимо определять указатель на неполный тип:

```
typedef struct t *T;
```

Можно определять переменные типа `T`.

Можно передавать эти переменные как аргументы в функцию.

❖ Для чего при реализации абстрактного типа данных используется неполный тип данных языка Си?

Неполный тип данных используется для скрытия реализации. Пример: поля структуры.

❖ Проблемы реализации АТД на языке Си.

- Именование. Если используются такие имена функций, как `create`, `read` или `write`, то реализовать несколько АТД в рамках одной программы будет сложно.
- Обработка ошибок. Интерфейс обычно описывает проверяемые ошибки времени выполнения и непроверяемые ошибки времени выполнения и исключения. Реализация не гарантирует обнаружение непроверяемых ошибок времени выполнения. Хороший интерфейс избегает таких ошибок, но должен описать их. Реализация гарантирует обнаружение проверяемых ошибок времени выполнения и информирование клиентского кода.

- «Общий АТД». Хотелось бы, чтобы, например, абстрактный тип данных «стек» мог хранить в себе данные любого типа без изменения заголовочного файла.

❖ Есть ли в стандартной библиотеке языка Си примеры абстрактных типов данных?

Примерами абстрактного типа данных в стандартной библиотеке языка Си являются `va_list` и `FILE`.