1 - Указатель на void. стандартные функции обработки областей памяти

Для чего используется указатель на void, примеры. особенности использования + примеры. Про функции обработки областей памяти memcpy, memset, memmove

Для чего используется указатель на void, примеры.

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

- полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов;

```
void *a;
int num = 52;
a = &num;
printf("%d", *(int*)a);
```

- позволяет передавать в функцию указатель на объект любого типа.

```
void print(void *ptr, char type)
{
    if (type == 'i')
        printf("%d\n", *(int*)ptr);
    if (type == 'f')
        printf("%.2f\n", *(float*)ptr);
}
```

особенности использования + примеры

Указатель типа void нельзя разыменовывать.

```
void *a;
int num = 52;
a = &num;
printf("%d", *a); // ПИСЮН - логично так как не понятно к какому типу мы приведём переменную
```

К указателям типа void не применима адресная арифметика

```
void *a;
int vodstok_data[6] = {1969, 1979, 1989, 1994, 1999, 2009};
```

```
a = vodstok_data;
a++; // ПИСЮН - логично так как не понятно на сколько ячеек памяти мы должны двинуться (size же разный)
```

memcpy

`memcpy` копирует данные побайтово

Сигнатура

```
void *memcpy(void *dest, const void *src, size_t count);
```

- dest — указатель на область памяти, куда будут скопированы данные.
- src — указатель на область памяти, откуда будут скопированы данные.
- n — количество байт для копирования.

Если области памяти `src` и `dest` перекрываются, поведение memcpy не определено

Пример

```
int src[5] = {1, 2, 3, 4, 5};
int dest[5];

// Копируем 5 элементов (по размеру int) из src в dest
memcpy(dest, src, 5 * sizeof(int));
```

memset

'memset' заполняет блок памяти указанным значением.

Сигнатура

```
int memcmp(const void *s1, const void *s2, size_t n);
```

- s1 и s2 — указатели на блоки памяти для сравнения.
- n — количество байт для сравнения.

Полезен для работы с сырыми данными.

memmove

Сигнатура

```
void *memmove(void *destptr, const void * srcptr, size_t num);
```

Переместить блок памяти. Функция копирует num байтов из блока памяти источника, на который ссылается указатель `srcptr`, в блок памяти назначения, на который указывает указатель `destptr`. Копирование происходит через промежуточный буфер, что, в свою очередь, не позволяет destination и `srcptr` пересекаться.

Пример

```
int src[5] = {1, 2, 3, 4, 5};
int dest[5];

// Копируем 5 элементов (по размеру int) из src в dest
memmove(dest, src, 5 * sizeof(int));
```

2 - Функции динамического выделения памяти

Про malloc, calloc, free. Порядок работы с функциями, особенности их работы. realloc и основные ошибки с ней. Явное приведение типа (за и против). Вопрос выделения 0 байт памяти ++ общие свойства, присущие функциям calloc, malloc, realloc.

Особенности malloc, calloc, realloc

Библиотека `#include <stdlib.h>`

- Указанные функции не создают переменную, они лишь выделяют область памяти. В качестве результата функции 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.
- После использования блока памяти он должен быть освобожден. Сделать это можно с помощью функции free.

malloc

Сигнатура

```
void* malloc(size_t size);
```

- Функция malloc (C99 7.20.3.3) выделяет блок памяти указанного размера size. Величина size указывается в байтах.

- Выделенный блок памяти не инициализируется (т.е. содержит «мусор»).

Пример

```
void *a = malloc(sizeof(int) * 3);
```

---

malloc и явное приведение типа

```
a = (int*) malloc(n * sizeof(int));
```

Преимущества явного приведения типа:

- компиляции с помощью c++ компилятора;
- у функции `malloc` до стандарта `ANSI C` был другой прототип `(char* malloc(size_t size))`
- дополнительная «проверка» аргументов разработчиком.

Недостатки явного приведения типа:

- начиная с `ANSI C` приведение не нужно
- может скрыть ошибку, если забыли подключить `stdlib.h`
- в случае изменения типа указателя придется менять и тип в приведении.

Короче щас это не нужно это нужно было только до стандарта `ANSI C` из за другой сигнатуры функции `malloc`

Нужно только если мы хотим делать какие либо шуры муры с c++

calloc

Сигнатура

```
void* calloc(size_t nmemb, size_t size);
```

- Функция calloc (C99 7.20.3.1) выделяет блок памяти для массива из nmemb элементов, каждый из которых имеет размер size байт.
- Выделенная область памяти инициализируется таким образом, чтобы каждый бит имел значение 0. (и дальше уже будет преборазование в зависимости от типа то есть если мы будет массив char он будет заполнен пустыми символами возмножно хуйню несу)

Пример

```
a = calloc(n, sizeof(int));
```

---

realloc

Перевыделение памяти

```
void* realloc(void *ptr, size_t size);
```

- `ptr == NULL && size != 0` Выделение памяти (как malloc)
- `ptr != NULL && size == 0` Освобождение памяти (как free).
- `ptr != NULL && size != 0` Перевыделение памяти

Перевыделение памяти (в худшем случае)

- выделить новую область
- скопировать данные из старой области в новую
- освободить старую область

возвращает указатель на новую ячейку памяти

если выделение не удалось возвращает `NULL`

При использовании обязательно нужно сделать вспомогательный указатель в который передать возвращаемое значение realloc и в случае если значение не `NULL` присвоить указатель

```
void *ptmp = realloc(pbuf, 2 * n);
if (ptmp)
    pbuf = ptmp;
else
    // обработка ошибочной ситуации
```

Что будет, если запросить 0 байт?

зависит от реализации (implementation-defined C99 7.20.3)

- вернется нулевой указатель;
- вернется «нормальный» указатель, но его нельзя использоваться для разыменования.

Короче или `NULL` или норм указатель но с которым нихуя нельзя сделать

free

```

```

```
void free(void *ptr);
```

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 попытаться освободить дважды указатель выделенные одной из функций выделения памяти, то `UB`

3 - Выделение памяти под динамический массив. Типичные ошибки при работе динамической памяти.

---

2 способа возврата динамического массива из функции. Реализовать функции.

О типичных ошибках при работе с динамической памятью.

Подходы к обработке ситуации когда ф-ции дин. памяти вернули null.

2 способа возврата динамического массива из функции. Реализовать функции.

- Как возвращаемое значение

```
int* create_array(FILE *f, size_t *n);
```

- Как параметр функции

```
int create_array(FILE *f, int **arr, size_t *n);
```

О типичных ошибках при работе с динамической памятью.

- Неверный расчет количества выделяемой памяти.
- Отсутствие проверки успешности выделения памяти
- Утечки памяти
- Логические ошибки

Логические ошибки

- Wild (англ., дикий) pointer: использование непроинициализированного указателя.
- Dangling (англ., висящий) pointer: 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 когда ф-ции дин. памяти вернули null.

- Возвращение ошибки (англ., return failure) – Подход, который используем мы
- Ошибка сегментации (англ., segfault) – Обратная сторона - проблемы с безопасностью
- Аварийное завершение (англ., abort) – Идея принадлежит Кернигану и Ритчи (xmalloc)
- Восстановление (англ., recovery) – xmalloc из git

4 - Указатели на функции. Функция qsort

Для чего в Си используются указатели на функции + примеры.

Как описывается, инициализируется указатель на функцию.

Как с его помощью вызывается сама функция.

Про qsort, примеры использования.

Особенности использования указ. на функцию (про адресную арифметику).

Про указатели на функцию и указатели на void

Для чего в Си используются указатели на функции + примеры.

- функции обратного вызова (англ., callback);
- таблицы переходов (англ., jump table);
- динамическое связывание (англ., binding).

Callback (англ, функция обратного вызова) - передача исполняемого кода в качестве одного из параметров другого кода.

Пример

`qsort`

Jump table (или таблица переходов) — это структура данных, которая используется для быстрого перехода между различными точками в программе.

Пример

```
void func1() {
    printf("Function 1\n");
}
```

```c
void func2() {
    printf("Function 2\n");
}

int main() {
    void (*jumpTable[3])() = {func1, func2};  // Jump table

    int choice;
    printf("Enter a number (0-1): ");
    scanf("%d", &choice);

    if (choice >= 0 && choice < 2) {
        jumpTable[choice]();  // Вызов функции по индексу
    } else {
        printf("Invalid choice\n");
    }

    return 0;
}
```

Прочекать исправить

Binding в программировании — это процесс связывания имен (или переменных) с объектами, функциями или значениями

Как описывается, инициализируется указатель на функцию.

Общее описание

```c
return_type (*pointer_name)(parameter_types);
```

Инициализация

```c
int add(int a, int b) {
    return a + b;
}

int (*func_ptr)(int, int) = add; // Инициализация указателя адресом функции
```

Как с его помощью вызывается сама функция.

Как та же функция только с другим именем

Про qsort, примеры использования.

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

Для функции `qsort` небоходимо разработать свой comparator (очевидно который соответствует сигнатруре приведённой в функции)

`qsort` может обрабатывать любой тип данных так как мы передаём размер одной ячейки нашего массива и функцию для сравнения этих числе

comparator ОБЯЗАТЕЛЬНО ДОЛЖЕН ВОЗВРАЩАТЬ

- ЧИСЛО 0 ПРИ РАВЕНСТВЕ ЧИСЛО
- БОЛЬШЕ 0 ЕСЛИ ПЕРВОЕ ЗНАЧЕНИЕ БОЛЬШЕ (НУ ИЛИ МЕНЬШЕ ЕСЛИ ВЫ ХОТИТЕ СОРТИРОВАТЬ ПО УБЫВАНИЯ)
- МЕНЬШЕ НУЛЯ ЕСЛИ ПЕРВОЕ МЕНЬШЕ (НУ ИЛИ БОЛЬШЕ ЕСЛИ ВЫ ХОТИТЕ СОРТИРОВАТЬ ПО УБЫВАНИЯ)

Особенности использования указ. на функцию (про адресную арифметику).

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

```
int (*p2)(int, int) = &add;
```

Короче это нахуй не нужно очередная устаревшая параша

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

```
int (*p3)(int, int) = *add;
int (*p4)(int, int) = *****add;
```

Даже не устаревшая а просто параша нахуй не нужна отпиздите меня санными тряпками если я не прав

Указатель на функцию может быть типом возвращаемого значения функции

```
#include <stdio.h>

// Функции
int add(int a, int b) { return a + b; }
int multiply(int a, int b) { return a * b; }

// Функция, возвращающая указатель на функцию
```

```
//  то что передалось в самау изнчальную функцию и её название
//                      |
// то соответтсвует   |  сигнатуре возращаемой функции
// |                  |          |    |
// v                 v          v    v
int (*get_operation(char op))(int, int) {
    return (op == '+') ? add : multiply;
}

int main() {
    // Получаем указатель на функцию add
    int (*operation)(int, int) = get_operation('+');

    // Вызов функции через указатель
    printf("%d\n", operation(3, 4)); // Вывод: 7

    // Получаем указатель на функцию multiply
    operation = get_operation('*');

    // Вызов функции через указатель
    printf("%d\n", operation(3, 4)); // Вывод: 12

    return 0;
}
```

Про указатели на функцию и указатели на void

Функция - не объект в терминологии стандарта.

Указатель на функцию одного типа может быть преобразован в указатель на функцию другого типа и обратно; результат должен быть равен исходному указателю. Если преобразованный указатель используется для вызова функции, тип которой несовместим с указанным типом, поведение не определено.

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.

5 - Утилита make. Назначение, простой сценарий сборки

Утилита make

Автоматизирует процесс преобразования файлов из одной формы в другую.
Разновидности:

- GNU Make (наша радость)

- BSD Make
- Microsoft Make

Для утилиты `make` входным данным является `makefile` - текстовый файл определенного формата, описывающий

- отношения между файлами программы
- команды для обновления каждого файла

Также `make` использует время последнего изменения файла чтобы решить, какие файлы надо обновить

Простой сценарий сборки:

```
app.exe: main.o list.o
    gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -o app.exe main.o list.o

main.o: main.c
    gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -c main.c

list.o: list.c list.h
    gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -c main.c

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

Как работает данный `makefile`:
Пока что в программе находятся только исходники. По запуску утилиты

```
make
```

она по умолчанию выполняет первую цель (`app.exe`). Видит зависимости, которые должны присутствовать в проекте, но их пока что нет, поэтому идет по очереди их выполнять.
Для зависимости `main.o` есть все необходимое (`main.c`), поэтому сначала выполнится команда цели `main.o`.
После этого выполнится `list.o`, затем `app.exe`.

Допустим, мы поменяли файл `list.c` и собираем программу. Программа make проверит, что цели содержат актуальные файлы. Для цели `main.o` все так, для `list.o` нет, так как `list.c` был изменен позже, чем имеющийся `list.o` -> он не актуальный и нужно его обновить. После обновления все объектники актуальные и тогда соберется `app.exe`.

Ключи утилиты

Указание конкретного мейкфайла

```
make -f makefile_2
```

Безусловное выполнение правил

```
make -B
```

Вывод команд без их выполнения

```
make -n
```

Игнорирование ошибок при выполнении команд

```
make -i
```

6 - Утилита make. Назначение, переменные, шаблонные правила

Для чего make, какие у нее входные данные, какая идея лежит в основе ее работы ++ какие разновидности утилиты make существуют. О переменных (бывают обычные, неявные, автоматические переменные - рассказать о каждой, для чего нужна и как использовать). Про шаблонные правила

Утилита make

Автоматизирует процесс преобразования файлов из одной формы в другую.
Разновидности:

- GNU Make (наша радость)
- BSD Make
- Microsoft Make

Для утилиты `make` входным данным является `makefile` - текстовый файл определенного формата, описывающий

- отношения между файлами программы
- команды для обновления каждого файла

Также `make` использует время последнего изменения файла чтобы решить, какие файлы надо обновить

Переменные в make

Бывают явные, неявные, автоматические.

Явные:

```
CFLAGS := -std=c99 -Wall -Werror
```

Неявные:

```
CFLAGS := -std=c99 -Wall -Werror
app.exe: main.c
    $(CC) $(CFLAGS) -o app.exe main.c
```

Неявная потому что она преднаписана уже в самой утилите. `$(CC)` будет равно `cc`.

Автоматические:

- `$^` - список зависимостей
- `$@` - имя цели
- `$<` - первая зависимость

Было:

```
CFLAGS := -std=c99 -Wall -Werror
app.exe: main.c
    $(CC) $(CFLAGS) -o app.exe main.c
```

Стало:

```
CFLAGS := -std=c99 -Wall -Werror
app.exe: main.c
    $(CC) $(CFLAGS) -o $@ $^
```

Шаблонные правила

```
CFLAGS := -std=c99 -Wall -Werror
CFILES := list.c kchaow.c
OFILES := $(CFILES:.c=.o)
app.exe: $(OFILES)
    $(CC) $(CFLAGS) -o $@ $^

%.o: %.c %.h
    $(CC) $(CFLAGS) -c $<
```

Шаблонное правило это `%.o` - утилита понимает, что ей нужно выполнить какую-то цель, и если название цели подходит под шаблон `%.o`, то она его выполняет, подставляя вместо знака `%` то, что стоит перед `.o` (в данном случае если правило `list.o`, то вместо `%` везде подставится `list`).

7 - Утилита make. Назначение, условные конструкции, анализ зависимостей

Для чего make, какие у нее входные данные, какая идея лежит в основе ее работы ++ какие разновидности утилиты make существуют. 2 подхода к реализации условных конструкций: директивы условные и переменные, которые зависят от целей. Рассмотреть анализ зависимостей в утилите make, 3 подхода (ручной анализ, подход когда любой си файл зависит от всех заголовочных файлов, автоматическая генерация зависимостей)

8 - Динамические матрицы.

Представление в виде одномерного массива и в виде массива указателей на строки. Анализ преимуществ и недостатков

Рассказать про 2 представления, сравнить между собой. Как динамическая матрица в том или ином представлении представлена в памяти компьютера + схема. Алгоритм выделения, освобождения памяти для матрицы. После анализ 2 представления с точки зрения + и - (таблица).

Почему-то с 8 по 13 вопросы одно и то же...

14 - Чтение сложных объявлений

О правилах чтения сложных объявлений в Си. Есть словарь, правила. Примеры чтения данных объявлений. Остановиться на ситуациях, которые в процессе чтения возникнуть не должны

15 - Строки в динамической памяти, функции POSIX, расширение GNU

3 функции, которые относятся к POSIX и GNU: strdup, getline, sprintf. Нужно не только рассказать про функции и особенности их работы, но и реализовать их функционал самостоятельно (подготовить getline). Про Feature Test Macro: что это такое, для чего нужно.

16 - Особенности использования структур с полями-указателями

Нужны примеры. Фраза о том, что операция присваивание в Си для структурных переменных по сути выполняет побитовое переменное 1 структурной переменной в область другой -> выводы о том, к чему приводит такое копирование в случае, если одно из полей - указатель. Рассказать о поверхностном и глубоком копировании. Про рекурсивное освобождение памяти из под структурных переменных (память под структуру динамическая и внутри нее есть динамические поля)
++Поверхностное - не всегда плохо

17 - Структуры переменного размера

Про поле типа flexible array member (появился в Си 99, но до этого тоже что-то было, КАК): все его особенности. Пример работы с подобной структурой (пример есть в лекции). Сравнить поле flexible array member с обычным указателем: + и -.

18 - Динамически расширяемый массив

Определение массива. Чем динамический массив отличается от динамически расширяемого. Описание типа, функции добавления и удаления. Особенности использования. ++Почему при перевыделении памяти эту память следует перевыделять крупными блоками.

19 - Линейный односвязный список. Добавление удаление элемента

Определение узла, списка, линейного односвязного списка. Чем отличаются массивы от списков. Реализовать функции и сопроводить ее схемой-картинкой. Как описывается и освобождается эта структура в Си.

20 - Линейный односвязный список. Вставка, удаление элемента

Определение узла, списка, линейного односвязного списка. Чем отличаются массивы от списков. Реализовать функции и сопроводить ее схемой-картинкой. Как описывается и освобождается эта структура в Си.

21 - Линейный односвязный список. Обход

Определение узла, списка, линейного односвязного списка. Чем отличаются массивы от списков. Реализовать функции и сопроводить ее схемой-картинкой. Как описывается и освобождается эта структура в Си.

22 - Бинарное дерево поиска. Добавление элемента

Определение

`Дерево` - связный ациклический граф.
`Двоичным деревом поиска` называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (левого и правого) и все вершины, кроме корня, имеют родителя. Основное свойство бинарного дерева: все левые потомки узла **меньше** его, все правые - **больше**.

Про узлы

Узел - единица хранения данных, несущая в себе ссылки на связанные с ней узлы.
Узел обычно состоит из двух частей: информационной и ссылочной

```
typedef struct node
{
    type_t data;
    struct node *next;
    // struct node *prev;
    // struct node *end;
} node_t;
```

Базовые операции

- добавление узла
- поиск узла
- удаление узла
- обход дерева

Описание ДДП в Си:

```
typedef struct tree_node
{
    int data;
    struct tree_node *left;
    struct tree_node *right;
} tree_node_t;
```

Опишем основные действия в дереве

- создание/очистка узла

```
tree_node_t *create_node(int data)
{
    tree_node_t *new = malloc(sizeof(tree_node_t));
    if (!new)
        return NULL;

    new->data = data;
    new->left = NULL;
    new->right = NULL;

    return new;
}

void free_node(tree_node_t *node)
{
    // free(data); // в случае, если наша data была бы указателем на
```

```
    динамические данные
    free(node);
}

int main()
{
    tree_node_t *node = create_node(5);
    printf("%d\n", node->data);
    free_node(node);
    node = NULL;
    return 0;
}
```

- добавление узла в дерево

```
int compare_nodes(const void *l, const void *r)
{
    ...
}

tree_node_t *insert(tree_node_t *tree, tree_node_t *node)
{
    if (tree == NULL)
        return node;
    int cmp = compare_nodes(tree->data, node->data);
    if (cmp < 0)
        tree->left = insert(tree->left, node);
    else
        tree->right = insert(tree->right, node);

    return tree;
}
```

- очистка дерева

```
void free_tree(tree_node_t *tree)
{
    if (tree)
    {
        free_tree(tree->left);
        free_tree(tree->right);
        free_node(tree);
        tree = NULL;
    }
}
```

23 - Бинарное дерево поиска. Поиск элемента

Определение

`Дерево` - связный ациклический граф.
`Двоичным деревом поиска` называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (левого и правого) и все вершины, кроме корня, имеют родителя.
Основное свойство бинарного дерева: все левые потомки узла **меньше** его, все правые - **больше**.

Про узлы

Узел - единица хранения данных, несущая в себе ссылки на связанные с ней узлы.
Узел обычно состоит из двух частей: информационной и ссылочной

```
typedef struct node
{
    type_t data;
    struct node *next;
    // struct node *prev;
    // struct node *end;
} node_t;
```

Базовые операции

- добавление узла
- поиск узла
- удаление узла
- обход дерева

Описание ДДП в Си:

```
struct tree_node
{
    int data;
    struct tree_node *left;
    struct tree_node *right;
} tree_node_t;
```

Опишем поиск в дереве

```
int compare_nodes(const void *l, const void *r)
{
    ...
}
```c
tree_node_t *find(tree_node_t *tree, int data)
{
    int cmp;
```
```

```
    while (tree)
    {
        cmp = compare_nodes(data, tree->data);
        if (cmp == 0)
            return tree;
        if (cmp > 0)
            tree = tree->right;
        else
            tree = tree->left;
    }
    return NULL;
}
tree_node_t *find2(tree_node_t *tree, int data)
{
    int cmp;
    if (tree == NULL)
        return NULL;

    cmp = compare_nodes(data, tree->data);
    if (cmp == 0)
        return tree;
    if (cmp < 0)
        return find2(tree->left, data);
    else
        return find2(tree->right, data);
}
```

Опишем очистку дерева

```
void free_tree(tree_node_t *tree)
{
    if (tree)
    {
        free_tree(tree->left);
        free_tree(tree->right);
        free_node(tree);
        tree = NULL;
    }
}
```

24 - Бинарное дерево поиска. Обход

Определение

`Дерево` - связный ациклический граф.
`Двоичным деревом поиска` называют дерево, все вершины которого упорядочены, каждая вершина

имеет не более двух потомков (левого и правого) и все вершины, кроме корня, имеют родителя.
Основное свойство бинарного дерева: все левые потомки узла **меньше** его, все правые - **больше**.

Про узлы

Узел - единица хранения данных, несущая в себе ссылки на связанные с ней узлы.
Узел обычно состоит из двух частей: информационной и ссылочной

```
typedef struct node
{
    type_t data;
    struct node *next;
    // struct node *prev;
    // struct node *end;
} node_t;
```

Базовые операции

- добавление узла
- поиск узла
- удаление узла
- обход дерева

Описание ДДП в Си:

```
struct tree_node
{
    int data;
    struct tree_node *left;
    struct tree_node *right;
} tree_node_t;
```

Опишем обходы в дереве

```
int compare_nodes(const void *l, const void *r)
{
    ...
}
```c
void inorder(tree_node_t *tree, void (*f)(tree_node_t, void *), void *arg)
{
    if (tree = NULL)
        return;
    inorder(tree->left, f, arg);
    f(tree, arg);
    inorder(tree->right, f, arg);
```
```

```
}

void preorder(tree_node_t *tree, void (*f)(tree_node_t, void *), void *arg)
{
    if (tree = NULL)
        return;
    f(tree, arg);
    preorder(tree->left, f, arg);
    preorder(tree->right, f, arg);
}

void postorder(tree_node_t *tree, void (*f)(tree_node_t, void *), void *arg)
{
    if (tree = NULL)
        return;
    postorder(tree->left, f, arg);
    postorder(tree->right, f, arg);
    f(tree, arg);
}
```

Инфиксный нужен чтобы получить отсортированные данные из дерева.
Префиксный нужен чтобы копировать дерево или префиксное выражение.
Постфиксный нужен чтобы удалять дерево, освобождать все ресурсы верно.

Опишем очистку дерева

```
void free_tree(tree_node_t *tree)
{
    if (tree)
    {
        free_tree(tree->left);
        free_tree(tree->right);
        free_node(tree);
        tree = NULL;
    }
}
```

25 - Бинарное дерево поиска. Удаление элемента

Определение

`Дерево` - связный ациклический граф.
`Двоичным деревом поиска` называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (левого и правого) и все вершины, кроме корня, имеют родителя.
Основное свойство бинарного дерева: все левые потомки узла **меньше** его, все правые - **больше**.

Про узлы

Узел - единица хранения данных, несущая в себе ссылки на связанные с ней узлы.
Узел обычно состоит из двух частей: информационной и ссылочной

```
typedef struct node
{
    type_t data;
    struct node *next;
    // struct node *prev;
    // struct node *end;
} node_t;
```

Базовые операции

- добавление узла
- поиск узла
- удаление узла
- обход дерева

Описание ДДП в Си:

```
struct tree_node
{
    int data;
    struct tree_node *left;
    struct tree_node *right;
} tree_node_t;
```

Опишем удаление в дереве

```
tree_node_t* delete_node(tree_node_t *root, int key) {
    if (root == NULL) return NULL;

    if (key < root->data)
    {
        root->left = delete_node(root->left, key);
    }

    else if (key > root->data)
    {
        root->right = delete_node(root->right, key);
    }

    else
    {
        if (root->left == NULL && root->right == NULL) {
            free_node(root);
            return NULL;
```

```
        }

        if (root->left == NULL) {
            tree_node_t *temp = root->right;
            free_node(root);
            return temp;
        }

        if (root->right == NULL) {
            tree_node_t *temp = root->left;
            free_node(root);
            return temp;
        }

        tree_node_t *current = root->right;
        while (current != NULL && current->left != NULL) {
            current = current->left;
        }
        root->data = current->data;
        root->right = delete_node(root->right, current->data);
    }

    return root;
}
```

Опишем очистку дерева

```
void free_tree(tree_node_t *tree)
{
    if (tree)
    {
        free_tree(tree->left);
        free_tree(tree->right);
        free_node(tree);
        tree = NULL;
    }
}
```

26 - Куча в программе на Си. Алгоритм работы функций malloc free

//Скорее всего будет разделено на 3 вопроса: 1) malloc, 2) free, 3) выравнивание Когда описывается алгоритм работы malloc или free нужно не только словесное описание но и сама реализация соответствующей функции (как на лекции). Лучше сразу реализацию и потом как комментарии писать общий алгоритм.

27 - Variable length array. Функция alloca

Рассказать про оба(FLA|VLA ?) и таблица сравнения VLA vs alloca.

28 - Функции с переменным числом параметров

Изложить идею, которая лежит в основе реализации функции с переменным числом параметров, потом сказать что так делать нельзя. Рассказать как правильно реализовывать функции с переменным числом параметров с помощью стандартной библиотеки.

29 - Препроцессор. Общие понятия. Директивы include, простые макросы, предопределенные макросы.

Препроцессор

Это часть компилятора, которая обрабатывает исходный код перед основной компиляцией.

- 1 - Обработка директив препроцессора (команды начинаются с символа #)
- 2 - Включение заголовочных файлов (замена директив #include на содержимое этих файлов)
- 3 - Определение и замена макросов (на их значения)
- 4 - Условная компиляция (включение, исключение частей кода на основе условий (#if, #ifdef, #ifndef))
- 5 - Генерация предупреждений или ошибок (#warning, #error)
- 6 - Передача дополнительных настроек компилятору (#pragma)
- 7 - Упрощение кода (удаление комментариев и текстовые преобразования)

Директива #include ???почему директивЫ в вопросе??? ----------

- 1 - #include - включает файл, использую стандартные системные директории поиска (например для стандартных библиотек)
- 2 - #include "filename" - включает файл, сначала ищет его в текущей директории, а затем в системных папках

Когда препроцессор встречает директиву #include, он заменяет её содержимым указанного файла, что позволяет организовать повторное использование кода и разделение программ на модули.

Важно:

- препроцессор не выполняет обработку кода из включаемых файлов, а просто вставляет его в места вызова
- файла включаются рекурсивно, и если файл уже был включен, он не будет включен снова (предотвращение циклической зав-ти)

Простой макрос

Это директива, которая используется для подстановки значений или выражений на этапе препроцессинга. Это позволяет заменить текст в коде до того, как он будет передан компилятору.

Синтаксис:

```
#define NAME value

#define PI 3.14
```

Каждый раз, когда в коде встречается PI, препроцессор заменяет его на 3.14

Ограничения:

Макросы не имеют типа и проверяются только на уровне текста. Важно соблюдать осторожность при использовании, чтобы избежать неожиданных побочных эффектов. Например, если они не заключены в скобки.

```
// например
#define SQUARE(x) x * x
int result = SQUARE(1 + 2);
// проблема: 1 + 2 * 1 + 2 = 5, а не 9
```

Предопределённые макросы

Это макросы, которые автоматически определяются компилятором на этапе препроцессинга. Эти макросы могут использоваться для получения информации о текущем окружении, платформе и процессе компиляции.

- `__FILE__` - представляет строку, содержащую имя исходного файла.

```
printf("This code is in file %s\n", __FILE__);
```

- `__LINE__` - представляет номер строки в исходном файла, на которой он встрачается.

```
printf("This is line number %d\n", __LINE__);
```

- `__DATE__` - представляет строку, содержащую дату компиляции исходного файла (Месяц, дата, год).

```
printf("Compiled on %s\n", __DATE__);
```

- `__TIME__` - представляет строку, содержащую время компиляции (часы:минуты:секунды).

```
printf("Compiled at %s\n", __TIME__);
```

- другие: `__STDC__` - если компилятор поддерживает ANSI - 1, `__GNUC__`, `__clang__` и тп - версия компилятора и его особенности

30 - Препроцессор. Макросы с параметрами

Препроцессор

Выше (29)

Макросы с параметрами

Это расширенный вид макросов, которые позволяют передавать параметры. Они определяются с использованием директивы #define и принимают аргументы, как функции, но обрабатываются на этапе препроцессинга

Синтаксис:

```
#define MACRO_NAME(param_list) replacement_text
```

Пример простого макроса с параметром:

```
#define SQUARE(x) ((x) * (x))

int res = SQUARE(6);
int a = SQAURE(1 + 2);
```

Плюсы:

- нет накладных расходов функций (нет стека для функции, просто подставляем текст в код)
- универсальность. Параметры макросов могут быть любого типа, так как типы не проверяются
- упрощение кода
- динамическая подстановка. Аргументы могут быть любыми выражениями, и они динамически подставляются в текст макроса

Минусы:

- отсутсиве проверки типов. Не проверяются типы аргументов, что может приводить к трудноуловимым ошибкам.
- проеблма с приоритетом операций. Без использования скобок, возможны ошибки, связанные с неправильным порядком выполнения операций:

```c
#define SQUARE(x) x * x

int res = SQUARE(6);
int a = SQAURE(1 + 2);
// 1 + 2 * 1 + 2 = 5, а не 9
```

- сложности в отладке, так как макросы заменяются текстом до компиляции, их трудно отлаживать

31 - Препроцессор. Общие понятия, директивы условной компиляции, директивы error и pragma

Указать проблему использования директивы if и ifdef. //Не было обсуждено на лекции, но в примерах лежит файл, на основе которого нужно сделать выводы самому (?)

Препроцессор

Выше (29)

Директивы условной компиляции

Они позволяют управлять включением или исключением фрагментов кода на основе заданых условий. Основные директивы:

- #if - условная проверка с логическим выражением

```c
#define DEBUG 1
#if DEBUG
    printf("DEBUF mode is active\n");
#endif
```

- #ifdef - проверяется, **определён** ли макрос

```c
#ifdef FEATURE_X
    printf("Feature X is enabled\n");
#endif
```

- #ifndef - проверяет, **не определён** ли макрос

```c
#ifndef CONFIG_H
#define CONFIG_H
    //содержимое
#endif
```

- #else - альтернативная ветвь выполнения, если одно из условий выше не выполняется

```
#ifdef DEBUG
    printf("DEBUG\n");
#else
    printf("DEFAULT\n");
#endif
```

- #elif - *иначе если*, позволяет проверить дополнительное условие

```
#if DEBUG == 1
    printf("DEBUG 1\n");
#elif DEBUG == 2
    printf("DEBUF 2\n");
#else
    printf("NO DEBUF\n");
#endif
```

- #endif - завершает блок условной компиляции

pragma u error

Эти директивы позволяют разработчику генерировать ошибки или предупреждения на этапе препроцессинга

- #pragma - используется для передачи специальных команд (инструкций) компилятору, обычно используется для включения/отключения определённых функций.
    - 1 Отключение предупреждений

```
#pragma warning(disable: 4996) //отключить предупреждение 4996
```

    - 2 Выравнивание данных

```
#pragma pack(oush, 1) //включить выравнивание данных по 1-му байту
struct Example {
    char c;
    int i;
}
#pragma pack(pop) // вернуть выравнивание по умолчанию
```

    - 3 Управление оптимизацией

```

```

```
#pragma optimize("", off) // отключить оптимизацию
#pragma optimize("", on) // включить оптимизацию
```

- 4 Указание процессора или архитектуры

```
#pragma GCC target("avx2") // использовать инструкции AVX2 (GCC)
```

- 5 Отключение определённых функций

```
#pragma GCC poison malloc // запрет использовать malloc
```

- #error - выводит сообщение об ошибке и завершает компиляцию

```
#ifndef CONFIG
#error "CONFIG must be defined!"
#endif
```

- #warning - выводит предупреждение, но не прерывает компиляцию

```
#ifdef DEPRICATED_FEATURE
#warning  "depricated_feature is being used!"
#endif
```

Проблема использования директивы if и ifdef

Хз насколько верно:

- 1 - компилятор не может проверить на ошибки директивы, т.е. где-то `#define CL_O_WN`, а в другом файле `ifdef CL_A_WN`. Т.е. синтаксические ошибки мы можем не заметить и часть кода никогда не исполнится
- 2 - проблемы с переносимостью, если условная компиляция зависит от платформоспецифичных макросов (_WIN32, __linux и тд)
- 3 - сложность отладки. Различные ветви кода активируются в зав-ти от компилятора, платформы или флагов.
- 4 - возможная путаница в больших проектах при наличии сложных связей
- 5 - неявные зав-ти между модулями

32 - Препроцессор. Общие понятия, операция # и

Препроцессор

Выше (29)

Операция #

`Операция #` конвертирует аргумент макроса в строковый литерал.

```
#define PRINT_INT(n) printf(#n " = %d\n", (n))
Где-то в программе
PRINT_INT(i / j);
// printf("i/j" " = %d", i/j);
```

Операция #

`Операция ##` объединяет две лексемы в одну.

```
#define MK_ID(n)
Где-то в программе
i##n
int MK_ID(1), MK_ID(2);
// int i1, i2;
```

33 - Встраиваемые функции.

Почему в Си появилось ключевое слово inline. Какие особенности есть у встраиваемых функций с точки зрения стандарта к чему они приводят. Как бороться с проблемой unresolved symbol при использовании inline (было рассмотрено 3-4 подхода).

Почему появилось слово inline

- Уменьшение накладных расходов на вызов функции, особенно небольших, часто вызыываемых функций
- предоставление более гибкого контроля над тем, где функция должна быть встраивваемой

Особенности с точки зрения стандарта

- `inline` – пожелание компилятору заменить вызовы функции последовательной вставкой кода самой функции. Компилятор может проигнорировать `inline` и не встраивать функцию.
- Не создаёт внешнее определение функции, крирпре мрднр использовать в других модулях, она становится локальной для текущего модуля
- Для использования в нескольких модулях нужно использовать комбинацию `extern inline`, после `inline и определения`
- `inline` не гарантирует отсутствие определения: если не ключевого слова extern, компилятор может сгенерировать отдельную реализацию функции на случай, если она вызвана по адресу.

```
inline double average(double a, double b)
{
    return (a + b) / 2;
}
```

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.

Способы исправления проблемы unresolved reference

- 1 - Использовать ключевое слово static (Такая функция доступна только в текущей единице трансляции.)

```
static inline int add(int a, int b) {return a + b;}
```

- 2 - Использовать ключевое слово extern (Такая функция доступна из других единиц трансляции.)

```
extern inline int add(int a, int b) {return a + b;}
```

- 3 - Добавить еще одно ***такое же не-inline*** определение функции ***где-нибудь*** в программе. Самый **плохой** способ решения проблемы, потому что реализации могут не совпасть.
- 4 - Убрать ключевое слово inline из определения функции. (Цитата: **Компилятор «умный»** 😀**, сам разберется.**)

```
int add(int a, int b) {return a + b;}
```

34 - Библиотеки. Статические библиотеки.

// Текст далее относится в целом про библиотеки: Особенности работы компоновщика, проблема видимости функций, что такое position independent code и как он устроен в Linux в ELF, LD_LIBRARY_PATH и R PATH (2 подхода), LD_PRELOAD и все что было на лекции.

// Может быть отдельным вопросом (если нет, то тоже указываем): Про 2 подхода к функциям, которые выделяют динамическую память: либо выделяем в библиотеке и пишем там функцию для очистки, либо все вопросы с выделением и освобождением памяти перекладываем на вызывающую сторону.

35 - Библиотеки. Динамические библиотеки. Динамическая компоновка.

36 - Библиотеки. Динамические библиотеки. Динамическая загрузка.

37 - Библиотеки. Динамические библиотеки на Си, приложение на Питоне

37.1 - ctypes

Про ctypes подробно. Как модулем ctypes пользоваться на примерах функции целочисленного сложения и целочисленного деления.

37.2 - ctypes

Про ctypes немного. Как с его помощью реализовать функции, которые работают с массивами.

37.3 - функции модули расширения. Разработка модуля расширения

Какие шаги нужно выполнить, чтобы реализовать модуль расширения. Подключить Python.h, все функции имеют один и тот же заголовок, как из аргументов, переданных императором, достать переменные в Си, как потом сформировать результат, что есть за метаинформация и т.д.

37.4 - все из 37.1-37.3 + разработка функции модуля расширени, которая обрабатывает последовательность.

38 - Абстрактный тип данных. Понятие модуль. Разновидности модулей. Абстрактный объект стек.

Определение модуля. Какие разновидности модулей бывают. Какие есть средства для реализации модулей в Си. Далее про абстрактный объект с примером реализации. // Если не успеваете, то начинаете с примера: реализация, а потом уже туда докинуть что-то из теории.

39 - Абстрактный тип данных. Понятие модуль. Разновидности модулей. Абстрактный тип данных стек целых чисел

Определение модуля. Какие разновидности модулей бывают. Какие есть средства для реализации модулей в Си. Далее про абстрактный тип данных с примером реализации. // Если не успеваете, то начинаете с примера: реализация, а потом уже туда докинуть что-то из теории.

40 - Списки ядра Линкус (горите в 9 кругах ада). Идеи, основные моменты использования.

Реализовать приложение по примеру того, что было в лекции (простейший список целых чисел с добавлением элемента, обходом списка, удалением элемента и освобождением памяти). // Должна быть предоставлена шпаргалка с названиями макросов. Если ее нет, напомнить

41 - Списки ядра Линкус. Идея, основные моменты реализации.

Сосредоточиться на реализации макроса container_of. (как по указателю на поле структуры можно получить указатель на саму структуру). // Должна быть предоставлена шпаргалка с названиями макросов. Если ее нет, напомнить

Надеюсь, конец 😃