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

???

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

```c
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

```c
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

```c
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

- 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 вспомагательный 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

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

- Изменение указателя, который вернула функция выделения памяти.
- Двойное освобождение памяти.
- Освобождение невыделенной или нединамической памяти.
- Выход за границы динамического массива.
- И многое другое

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

???

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");
}

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

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

Инициализация

```
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 возвращаемой 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.