Адресная арифметика и динамическое выделение памяти в примерах

Разница между const * и * const

```
#include <stdio.h>

int main()
{
    int a = 3;

    int const * a_ptr = &a;  //Указатель на константное целое число.
    //const int * a_ptr = &a;//Идентичная запись. const стоит справа от типа
    *a_ptr = 43;  //ОШИБКА! Изменение константного значения.
    a_ptr++;      //ОК! Увеличение неконстантного указателя.

    int * const b_ptr = &a;   //Константный указатель на целое число.
    *b_ptr = 45;  //ОК! Изменение неконстантого значения.
    b_ptr++;      //ОШИБКА! Изменение констного указателя.

    return 0;
}
```

Практика:

Попробуйте использовать int const * const. Что это будет означать? Какие операции допустимы, какие нет?

Копирование строк

```
#include <stdio.h>

#define MAX 100   //Максимальный размер строки.

//Копирование строки с использованием массивов.
void copy_str1(char str_dest[], char str_src[])
{
    int i = 0;
    for(; str_src[i] != '\0'; i++) //Пока в исходной строке не встретим '\0'
        str_dest[i] = str_src[i];  //Поэлементно копируем символы.

    str_dest[i] = str_src[i]; //Добавляем символ '\0' в результирующую строку.

/*    Сокращенный вариант реализации в одну строку.  */
//  for(int i = 0; (str_dest[i] = str_src[i]) != '\0'; i++); //Без скобок не работает!
}

//Копирование строки с использованием указателей.
void copy_str2(char *str_dst, char *str_src)
{
    int i = 0;
    //str_src изначально содержит указатель на первый элемент.
    //с помощью i будет осуществляться смещение от первого элемента строки.
    //для получения значения остальных элементов.
```

```
    while(*(str_src + i ) != '\0')
    {
        *(str_dst + i) = *(str_src + i);
        i++; //Увеличиваем смещение.
    }

  /*  Сокращенный вариант реализации в одну строку.   */
  /*  Здесь увеличивается не счетчик, а сами указатели. */
  //for(; (*str_dst = *str_src) != '\0'; str_src++, str_dst++);
}


int main()
{
    char str1[MAX] = "Hello world!";
    char str2[MAX] = "";

    //copy_str1(str2, str1);
    copy_str2(str2, str1);

    printf("str1 = %s\n", str1);
    printf("str2 = %s\n", str2);

    return 0;
}
```

Практика:

В реализации copy_str2 попрубуйте использовать квалификатор const в описании аргументов функции. В каком варианте реализации какие квалификаторы можно использовать?

Смещение по массиву структур

```
#include <stdio.h>
#define MAX 100

//Структура описывающая информацию о студенте.
struct student {
    char name[MAX];
    int age;
};

//Функция печати информации об одном студенте.
void print_student(struct student stud)
{
    printf("Name : %s\n", stud.name);
    printf("Age : %d\n", stud.age);
}

//Функция печати информации о нескольких студентах.
//Реализация с использованием массивов.
void print_all_students_arr(struct student students[], int count)
{
    for(int i = 0; i < count; i++)
        print_student(students[i]);

}

//Функция печати информации о нескольких студентах.
//Реализация с использованием указателей.
//Входной параметр – указатель на первый элемент массива.
```

```
void print_all_students(struct student *stud_start, int count)
{
    for(int i = 0; i < count; i++)
        print_student(*(stud_start + i)); //Смещаемся по счётчику от адреса
первого элемента.
}

int main()
{
    struct student students[MAX] = { { .name = "ivan", .age = 20}, { .name =
"petr", .age = 21 } };
    print_all_students_arr(students, 2);
    //print_all_students(students, 2);
    return 0;
}
```

Практика:

В функцию печати передается копия объекта на структуру. Можно ли использовать указатель? константный указатель? Указатель на константу? Экономится ли при этом память?

Динамические массивы

Часто длина массива неизвестна до выполнения программы, закладывать заранее большой размер непрактично и занимает много памяти. Необходимо средство позволяющее указывать размер во время программы.

Нижеприведенный код будет компилироваться только компилятором gcc с использованием стандарта c99 (опция: `-std=c99`). Подробно можно почиать тут: https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html

```
#include <stdio.h>

int main()
{
    int size;
    printf("Enter size: ");
    scanf("%d", &size);

    //int array[size] = {0};
    int array[size];

    for(int i = 0; i < size; i++)
    {
        printf("Enter array[%d]: ", i);
        scanf("%d", &array[i]);
    }

    for(int i = 0; i < size; i++)
        printf("%d ", array[i]);

    return 0;
}
```

Выделение памяти в «куче»

Во всех наших предыдущих программах мы пользовались автоматической памятью, также называемой «стеком». Удобство использования автоматической памяти заключается в том, что необходимая память для переменных выделяется и очищается автоматически. Однако на стек накладываются ограничения не позволяющие хранить большие данные или данные переменной длины (компилятор не имеет возможности расчитать требуемый им объём).

Чтобы обойти эти ограничения в процессе работы мы можем запрашивать данные из динамической памяти, именуемой «кучей». Под «кучей» можно понимать всю оперативную память компьютера. Тут это описано немного подробнее http://www.programbeginner.ru/?p=323.

Рассмотрим на примере как выделить память под массив, длина которого вводиться с клавиатуры.

```
#include <stdio.h>
#include <stdlib.h> //Содержит описание функции malloc().

int main()
{
    int size = 0;

    //Ввод размера массива.
    printf("Enter size : ");
    scanf("%d", &size);

/* Для выделения памяти из «кучи» используется функция malloc(). */
/* В качестве аргумента ей передается количество необходимых байт. */
/* Чтобы правильно расчитать удобно использовать функцию sizeof() */
/* malloc возвращает бестиповый указатель void*, который хорошо привести к int*
*/

    int* array = (int*)malloc(sizeof(int) * size);

/* Ввод данных массива. */
    for(int i = 0; i < size; i++)
        scanf("%d", (array + i));

/* Подсчёт среднего арифметического. */
    int sum = 0, i = 0;
    for(; i < size; i++)
        sum += *(array + i);

   printf("Average = %f\n", (double)sum / (double)i);

   free(array); //Обязательная очистка памяти!

  return 0;
}
```

Статическая память

Переменные находящиеся в статической памяти находятся в одном том же месте и сохраняют своё значение в течении всего времени работы программы. Значения статических

переменных инициализируются до функции `main()` константными значениями не требующими вычисления. Если значение не задано, переменные инициализируются нулём.

```
#include <stdio.h>

void foo()
{
    static int counter = 0; //Статическая переменная в функции.

    counter++;
    printf("Counter = %d\n", counter);
}


int main()
{
    for(int i = 0; i < 10; i++)
        foo();
    return 0;
}
```

Практика:

Попробуйте добавить функцию нестатическую переменную. Будет ли она сохранять своё значение?

Структуры, ссылающиеся на себя. Связанные списки

С помощью структур и указателей на языке Си можно реализовывать большое количество разнообразных структур данных. Рассмотрим самую простую — односвязанный список.

Теорию можно почитать на [википедии](#).

Более полная реализация с картинками http://learnc.info/adt/linked_list.html

```
#include <stdio.h>
#include <stdlib.h> //malloc();

//Структура, описывающая элемент списка.
struct list
{
    int data;
    struct list *next;
};

//Функция печати списка.
void print_list(struct list *head)
{
    struct list *ptr = head; //Создаём временный указатель-итератор.
    int counter = 0;         //Счётчик для нумерования элементов.
    while(ptr != NULL)  //Пока указатель-итератор не будет указывать «в никуда»
    {
        printf("list[%d] = %d", counter++, ptr→data); //Выводим информацию
        ptr = ptr->next;     //Переход к следующему элементу.
    }
    printf("\n");
}
```

```
//Функция создания элемента
//Возвращает указатель на созданный элемент.
struct list *create_new_list_elem(int data)
{
    struct list *res_ptr = NULL; //Создаём временный указатель.

    /* Выделяем память под один элемент структуры элемента списка. */
    res_ptr = (struct list*)malloc(sizeof(struct list));
    res_ptr->data = data; //Записываем значение.
    res_ptr->next = NULL; //Указатель на следующий элемент указывает «в никуда».

    return res_ptr;
}

//Функция добавления элемента в список.
//На вход передаётся указатель на первый элемент и на добавляемый элемент.
void add_elem_to_list(struct list *head, struct list *new_elem)
{
    struct list *ptr = head;
    while(ptr->next != NULL) //Пока мы не найдем последний элемент
        ptr = ptr->next;      //Переходим к следующему элементу.

    ptr->next = new_elem;     //Последний элемент ссылается на добавленный
элемент.
}

//Очистка памяти, удаление списка.
void clear_list(struct list *head)
{
    struct list *ptr = head;
    while(ptr != NULL)
    {
        struct list *elem_to_remove = ptr; //Делаем копию «итератора»
        ptr = ptr->next;      //Переходим к следующему элементу.
        free(elem_to_remove); //Очищаем элемент на который указывает копия.
    }
}

int main()
{
    struct list *test_elem = create_new_list_elem(42);
    add_elem_to_list(&new_list, test_elem);
    print_list(&new_list);

    test_elem = create_new_list_elem(43);
    add_elem_to_list(&new_list, test_elem);
    print_list(&new_list);

    clear_list(&new_list);

    return 0;
}
```

Практика:

1) Почему следующий фрагмент кода приведет к ошибке во время выполнения?

```
int main()
{
    struct list new_list = { .data = 1, .next = NULL };
    print_list(&new_list);
```

```
    struct list *test_elem = create_new_list_elem(42);
    print_list(test_elem);

    add_elem_to_list(&new_list, test_elem);
    print_list(&new_list);

    clear_list(&new_list);

    return 0;
}
```

2) Попробуйте добавить квалификаторы const к указателям в аргументах функций. Определите какие входные параметры константные, а какие нет?

3) Определите элемент списка как отдельный тип с помощью typedef. Познакомиться можно тут http://www.c-cpp.ru/books/typedef . Сделает ли это код более понятным?

4) Напишите дополнительные функции для работы со списком. Например, поиск элемента, удаление элемента, сортировка списка, создание списка из массива и прочее.