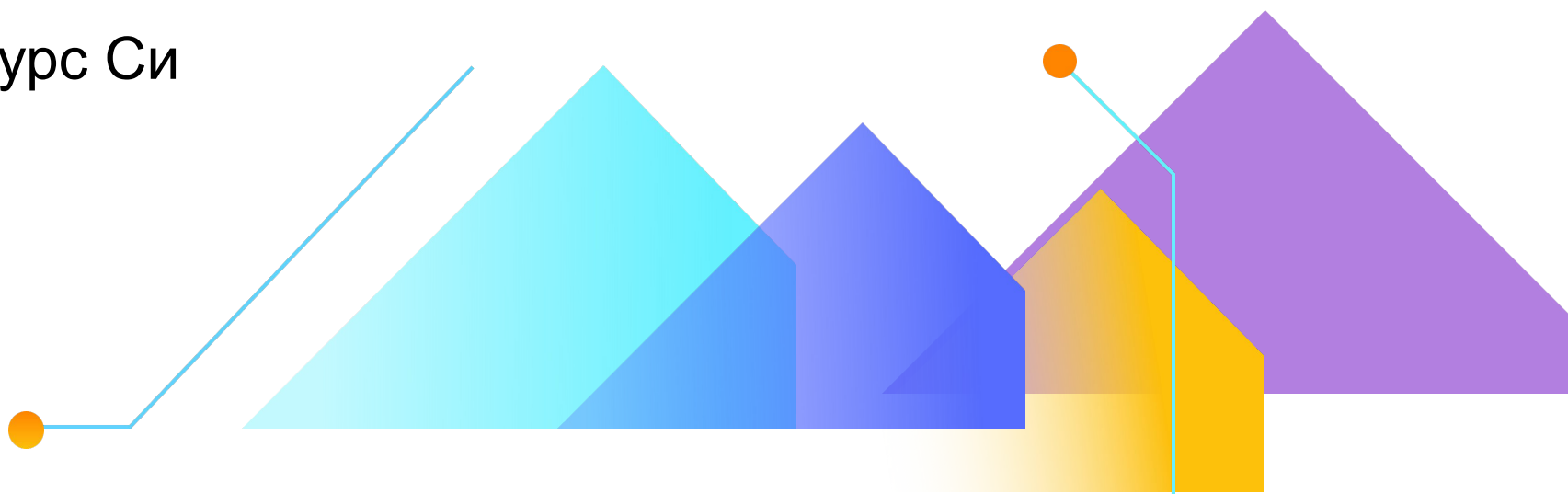




Лекция №2

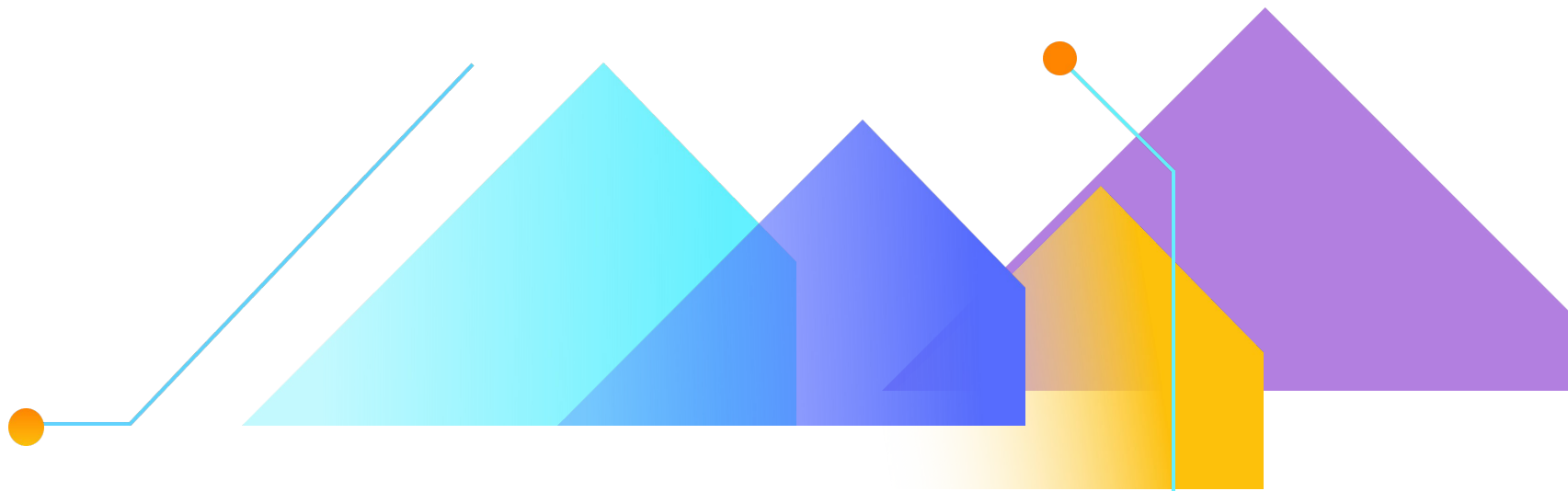
Структуры Динамические типы

Продвинутый курс Си






План курса

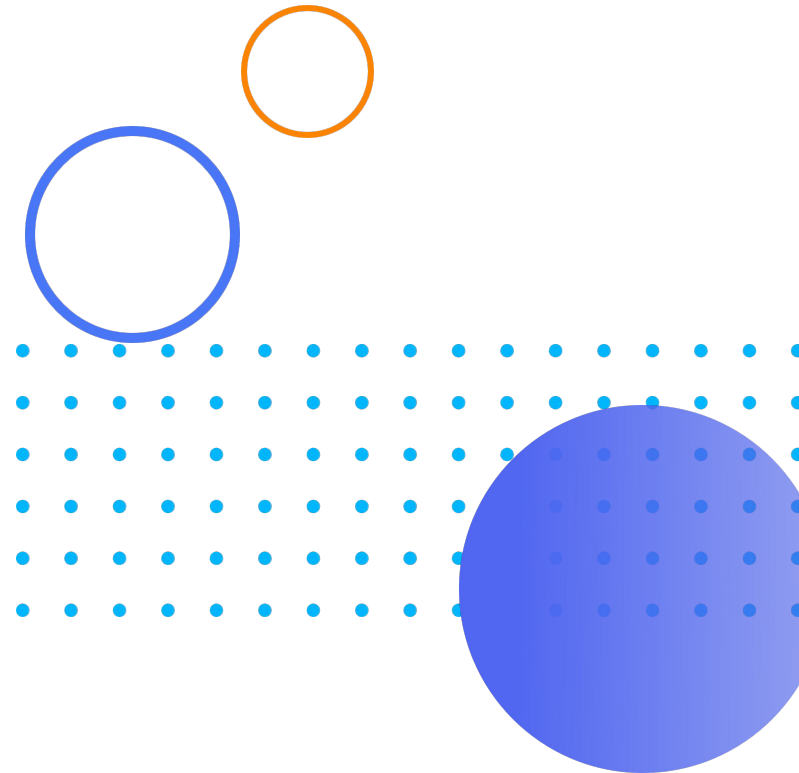
- Вводный урок
- Структуры. Динамические типы
- Библиотеки языка C
- Оптимизация кода
- Алгоритмы
- Компиляция и компиляторы
- Динамические структуры данных
- Курсовая работа



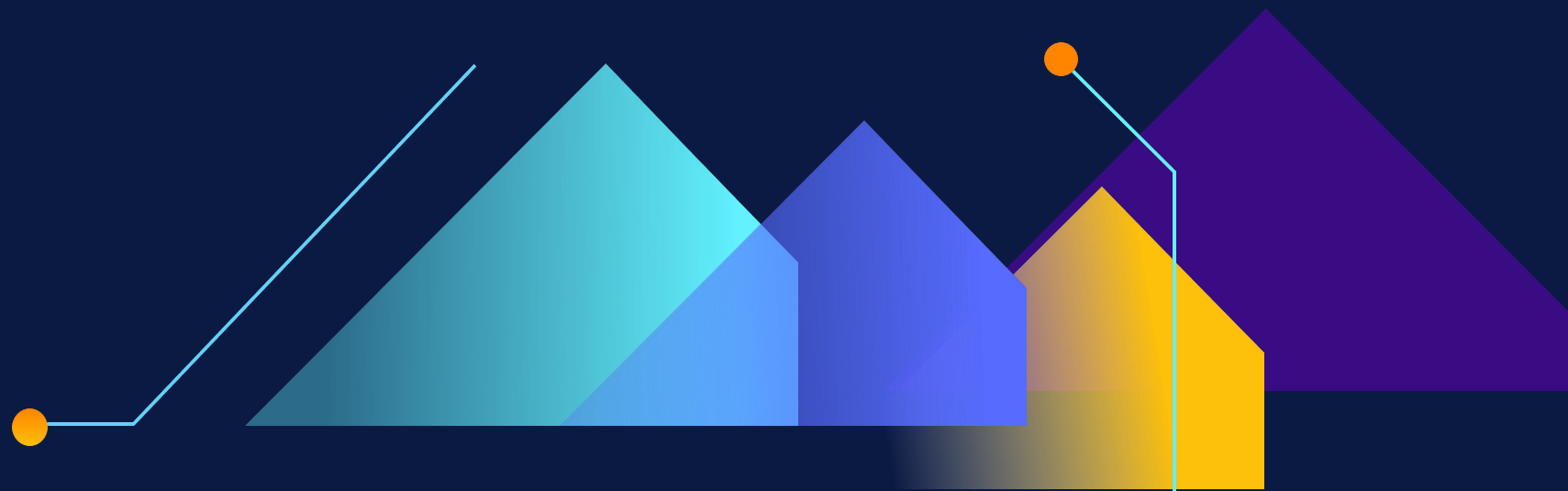
Маршрут

Структуры. Динамические типы

-  Повторим указатели
-  Рассмотрим особенности работы с динамической памятью
-  Разберём ошибки при работе с указателями



Указатели



Указатели

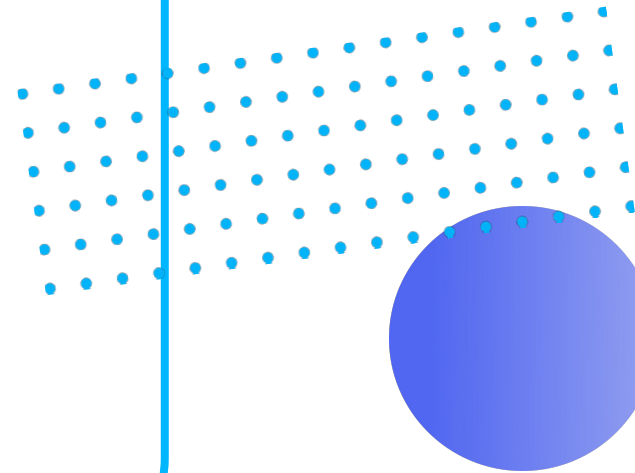
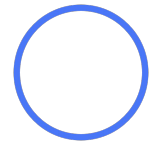


Указатель — это переменная, значением которой является адрес некоторого объекта. При задании указателя необходимо определить базовый тип на который данный указатель будет ссылаться. Определено также специальное значение NULL (0), которое не соответствует никакому допустимому адресу объекта.



Определение указателя

- **int *x;** // Указатель на тип int
- **char * func(int *ptr);** // Функция, которая возвращает указатель на char и принимает на вход указатель на тип int.
- **int *arr[10];** // Массив указателей
- **void *pv;** // Указатель на пустой тип
- **int (*ars)[5];** // Указатель на строку из 5 элементов int.
- **char (* pfunc)(int, float);** // Указатель на функцию, которая возвращает char и принимает два аргумента int и float
- **int **p;** // Указатель на указатель на тип int.



Переменные с типом указатель

Указательный тип может иметь в качестве своего базового типа любой тип, в частности сам указатель, тип структура, массив и т. д.

В переменных с типом указатель можно хранить адреса объектов любого типа в том случае, если типы самого объекта и указателя согласованы.

Так как адрес объекта имеет размер, не зависящий от типа объекта, то возможно приведение одних типов указателей к другим.

Используя это можно посмотреть как выглядит вещественное число в памяти компьютера в шестнадцатеричном виде.

Пример

```
float f = 2.0;  
int *p;  
p = &f;  
printf("*p = %x\n", *p);  
*p = (127+3)<<23; // кладем в f 2^3  
printf("f = %f\n", f);
```

*p = 40000000

f = 8.000000

Приведение типов

Внимание! Если указатель не инициализирован, то он указывает на случайную область памяти.

Внимание! В реальной жизни не стоит злоупотреблять явным приведением значений одних указательных типов к другим, это может привести к появлению трудных для обнаружения ошибок.

Определена унарная операция разыменования `*` и унарная операция взятия адреса `&`. Причём `&(*ptr)` всегда тождественно равно `ptr`, даже если `ptr` при этом равен `NULL`.

Пример приведения типов указателей

```
int *p = NULL; // 0 - не указывает ни на что
/*ОШИБКА! попытка разыменовать нулевой
указатель */
if( *p )
    printf("True\n");
```

Segmentation
fault

```
/* Нет ошибки */
int *p = NULL;
if( p == &(*p) )
    printf("True\n");
```

True

Условные операторы в циклах

Выражения с указателями можно использовать в условных операторах и циклах.

```
for (p = head; p; p = p->next)  
    // работает, пока p не станет 0 (NULL)
```

Специальный тип void

Существует специальный тип обобщенного указателя `void *`.

Такому указателю можно присваивать значения указателей любого типа, и такой указатель можно присваивать указателю любого типа. Обычно он используется для передачи объекта в функцию, когда тип передаваемого объекта заранее не известен.

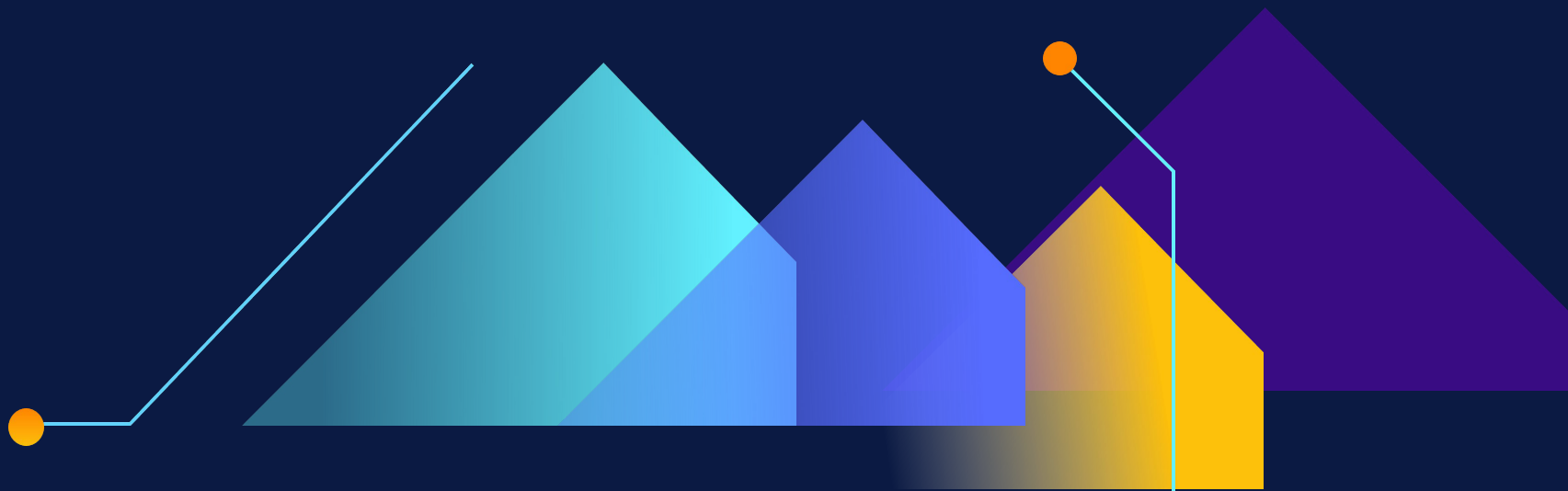
Внимание! Указатель `void *` нельзя разыменовывать, но можно сравнивать с нулём.

Пример со специальным типом void

```
_Bool is_same(void *a, void *b) {  
    // *a = *b; // ОШИБКА! void * нельзя  
    разыменовывать  
    return a == b; // Можно только  
    сравнивать  
}  
int main()  
{  
    int a=5;  
    int *pa = &a;  
    is_same(&a, pa) ? printf("Same\n") :  
    printf("Different\n");  
    return 0;  
}
```

Same

Указатели и массивы



Имя массива

Имя массива — это константный указатель на его первый элемент.

```
int a[5] = {1,2,3,4,5}; // в памяти выделено 20 байт
int * const pa = &a[0]; // в памяти выделено 8 байт
int *pb = &a[0]; // в памяти выделено 8 байт
//a = &a[2]; // ОШИБКА!
//pa = &a[1]; // Нельзя менять константный указатель
pb = &a[1];
pb[2] = 100;
```

Объявление массива

При объявлении массива, в памяти будет выделено достаточное место для хранения заявленного числа элементов, а при объявлении указателя — для хранения указателя (обычно 8 байт для 64 разрядной ОС).

Рассмотрим пример реализации библиотечной функции `strcpy`, которая копирует строку из второго аргумента в первый и возвращает адрес новой строки.

```
int main()  
{  
    char a[]="Hello!", b[7];  
    strcpy(b,a);  
    printf("b = %s\n",b);  
    return 0;  
}
```


Пример реализации функции strcpy

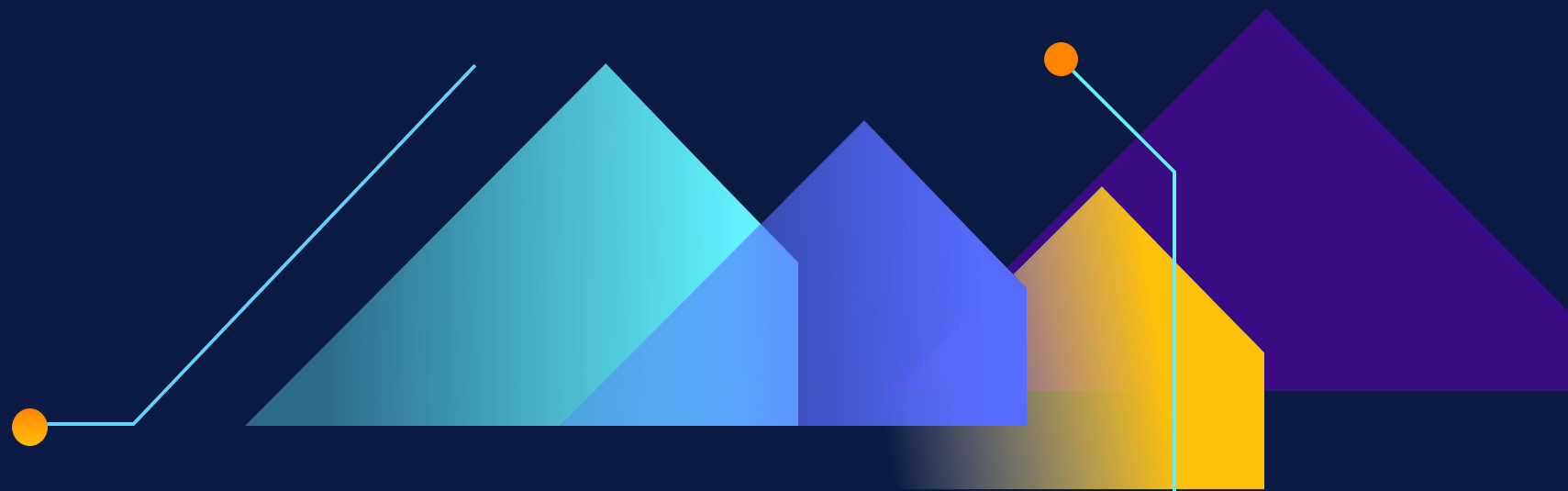
```
char *strcpy(char *dst, const char *src ) {  
    char *d = dst;  
    while ((*dst++ = *src++));  
    return d;  
}
```

Hello

Внимание! Функция strcpy никак не контролирует возможность переполнения массива.

Порядок выполнения операций при вычислении условия цикла while. Операции * и ++ имеют одинаковый приоритет и выполняются справа налево. Однако постфиксная операция ++ не изменяет указатель src, пока по нему не получено значение.

Динамическая память

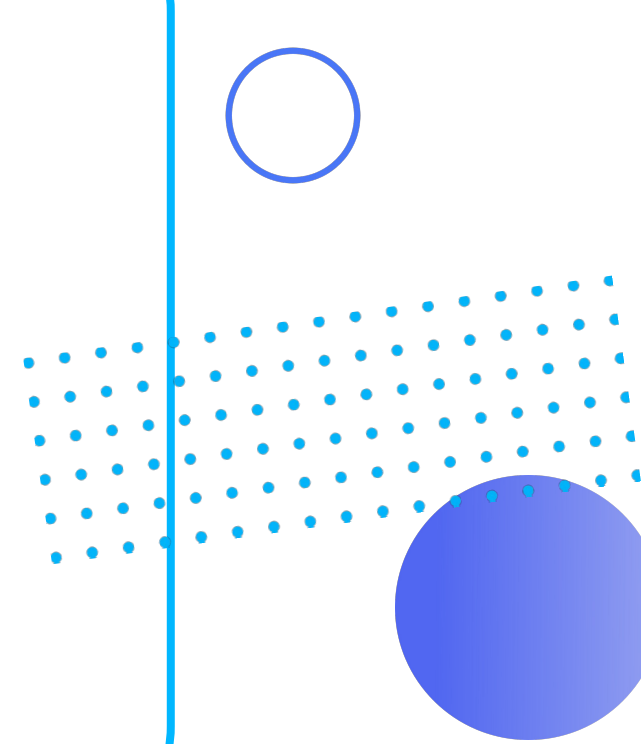




Динамическая память

Помимо области памяти, выделяемой во время компиляции программы для глобальных переменных, и области памяти в стеке, выделяемой для локальных переменных, существует область памяти (традиционно называемая «кучей»), представляющая собой нечто среднее между этими двумя типами памяти. Куча существует всё время выполнения программы, но память в ней выделяется и освобождается динамически, по требованию программиста.

Средства работы с динамической памятью не встроены в язык, а предоставляются стандартной библиотекой. Чтобы использовать функции работы с динамической памятью, необходимо подключить заголовочный файл `stdlib.h`.





Функции

Для работы с динамической памятью определены следующий функции:

- **`void *malloc(size_t size);`** — выделяет память указанного размера и возвращает указатель на начало выделенного блока
- **`void *calloc(size_t nitems, size_t nsize);`** — выделяет память под массив с последующей инициализацией элементов нулями
- **`void free(void *ptr);`** — освобождает ранее выделенную память
- **`void *realloc(void *ptr, size_t newsize);`** — изменяет размер выделенного блока памяти

Пример

Рассмотрим пример: на стандартном потоке ввода задаётся последовательность вещественных чисел. Ввод завершается концом файла. На стандартный поток вывода необходимо распечатать только числа, не превышающие среднее арифметическое всех введённых чисел. Для хранения чисел будем использовать динамический массив, созданный при помощи функции malloc.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {

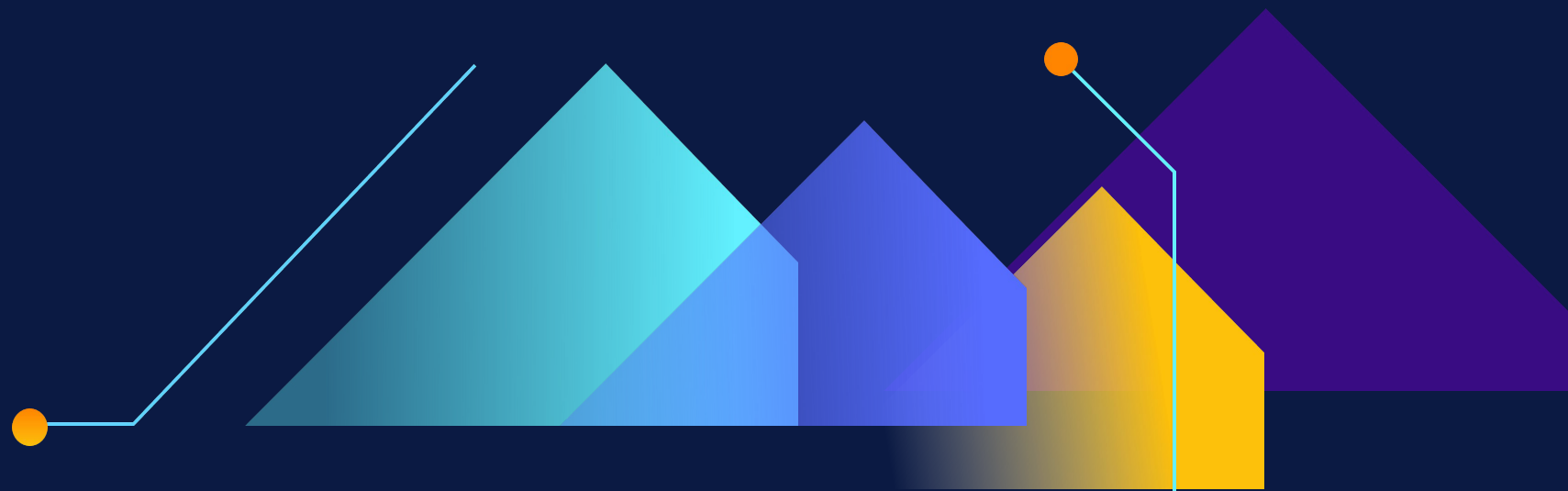
    int arr_size=0, arr_u=0, i;
    double *arr = NULL;
    double s=0,v;
    arr_size = 16;
    if (!(arr = (double*) malloc(arr_size * sizeof(arr[0])))) {
        goto out_of_mem;
    }
```

Пример

```
while (scanf("%lf", &v) == 1) {
    if (arr_u == arr_size) {
        arr_size *= 2;
        if (!(arr = (double*)
realloc(arr, arr_size *
sizeof(arr[0]))))
            goto out_of_mem;
    }
    s += v;
    arr[arr_u] = v;
    arr_u++;
}
```

```
if (!arr_u) return 0;
s /= arr_u;
for(i=0; i < arr_u; i++) {
    if (arr[i] <= s)
printf("%lf\n", arr[i]);
}
free(arr);
return 0;
out_of_mem:
    fprintf(stderr, "Can't
allocate memory\n");
    return 1;
}
```

Ошибки при работе с указателями



Ошибка при множественном объявлении переменных

Знак * относится к имени переменной p1, а не к типу int. В данном случае происходит определение указателя p1 и целочисленной переменной p2.

```
int main()  
{  
    int *p1, p2;  
    int n = 30;  
    p1 = &n;  
    p2 = &n; // ОШИБКА  
}
```

```
int *p1, *p2;
```


Использование неинициализированного указателя

```
int *p1; // указывает на случайную
область памяти
int n = *p1; // ОШИБКА разыменования
printf("%d", n);
```

```
int *p1;
int n = *p1;
int m = 10;
p = &m;
printf("%d", n);
```

```
int* p1;
int m;
p1 = &m; // указатель ссылается на
          // неинициализированную
          // переменную

int n = *p1;
printf("%d", n);
```

```
int m = 100;
```

Присвоение неверного значения в указатель

```
int *p1;  
int m = 100;  
p1 = m; // ОШИБКА
```

```
p1 = &m;
```

Некорректный инкремент при работе с указателем

```
int *p1;  
int m = 100;  
p1 = &m; // присваиваем адрес m  
*p1++; // ОШИБКА! увеличение  
        //адреса, а не значения
```

```
(*p1)++;
```

Неверное освобождение памяти

Функция `free` может освобождать только память, выделенную ранее при помощи функции `malloc`.

```
int* p1;  
int m = 100;  
p1 = &m;  
free(p1); //ОШИБКА! Попытка освободить  
стековую память
```

Обращение к неопределённой памяти

```
int* p1;  
p1 = malloc(sizeof(int));  
*p1 = 99;  
free(p1);  
*p1 = 100; // ОШИБКА! память не  
определена
```

```
int* p1;  
p1 =  
malloc(sizeof(int));  
*p1 = 99;  
*p1 = 100;  
free(p1);
```

Повторное использование free

```
char *str1 = malloc(strlen("Hello world")
+ 1);
strcpy(str1, strlen("Hello world") + 1,
"Hello world");
//...
free(str1); // Первый раз free
//...
free(str1); // ОШИБКА! второй раз free
```

```
str1 = NULL;
```

Использование sizeof и malloc

Размер типа данных может отличаться на разных архитектурах. Если вы передадите в malloc какое-то константное число, то могут возникнуть проблемы при переносе на другие архитектуры.

```
int *p;  
p = malloc(4); // ОШИБКА!  
           // нарушение переносимости
```

```
p = malloc( sizeof (int) );
```

Утечка памяти

Не забывайте освобождать выделенную память. Не оставляйте в памяти блоки, на которые никто не ссылается.

```
int* p = malloc(sizeof(int));
*p = 5;
p = malloc(sizeof(int));
//ОШИБКА!
    // Предыдущий блок все еще
    // занимает память
```

```
int* p = malloc(sizeof(int));
*p = 5;
free(p);
p = malloc(sizeof(int));
```


Копирование структур и указатели

При присваивании структур происходит побайтовое копирование всех полей одной структуры в другую.

В данном примере указатели в переменных t1.s и t2.s указывают на одну и ту же область памяти.

```
struct st { char *s; } t1, t2;  
t1.s = malloc(10);  
strcpy (t1.s, "Hello ");  
t2 = t1;  
printf ("%s", t2.s);  
strcpy (t1.s, "world!");  
printf ("%s", t2.s);
```

Hello world!

Копирование структур и указатели

В этом примере в полях t1.s и t2.s хранятся два разных массива из 10 СИМВОЛОВ.

```
struct st { char s[10]; } t1, t2;  
strcpy (t1.s, "Hello ");  
t2 = t1;  
printf ("%s", t2.s);  
strcpy (t1.s, "world!");  
printf ("%s", t2.s);
```

Hello Hello

Работа с двумя указателями, ссылающимися на одну область памяти

В приведенном ниже примере. `str1` и `str2` указывают на один и тот же блок памяти. Поэтому, когда `str1` освобождается, по существу освобождается блок памяти, на который указывает `str2`. Любая попытка использовать `str2` после освобождения `str1` приведёт к неопределённому поведению.

Для выявления такого рода ошибок имеет смысл использовать статические анализаторы кода или подключить дополнительную библиотеку, например `SmartPointer`.

```
int len = strlen("Hello");
char *str1 = malloc(len + 1);
strcpy(str1, "Hello");
char *str2 = str1;
printf("%s\n", str1);
// ... много кода
free(str1);
// .. много кода
printf("%s\n", str2); // ОШИБКА! Эта память уже не определена
```

Обращение к последнему элементу в массиве

```
size_t const SIZE = 10;  
int *arr;  
arr = malloc(sizeof(int) * SIZE);  
arr[SIZE] = 100; // ОШИБКА! выход за  
                // границу массива
```

```
arr[SIZE-1] = 100;
```

Работа с двумя указателями, ссылающимися на одну область памяти

В данном примере происходит обращение к 4-х байтовой переменной, которая лежит в памяти в формате big-endian. Из-за того, что вместо типа `int` используется указатель на двухбайтовый тип `short`, то из памяти прочитаны младшие 2 байта переменной `num`.

```
int  num = 2147483647; // 0x7fffffff
int  *pi = &num;
short *ps = (short*)pi;
printf("pi: %p  Value(16): %x  Value(10): %d\n", pi, *pi, *pi);
printf("ps: %p  Value(16): %hx  Value(10): %hd\n", ps, *ps,
*ps);
```

```
pi: 0x7ffeecb34a58  Value(16): 7fffffff  Value(10): 2147483647
ps: 0x7ffeecb34a58  Value(16): ffff  Value(10): -1
```

Сравнение указателей вместо значений

В этом примере сравниваются адреса двух разных строк, одинаковых по содержанию. Несмотря на то, что содержимое строк одинаково, они расположены в разных блоках памяти. Для сравнения строк необходимо использовать функцию `strcmp` из библиотеки `string.h`.

```
int len = strlen("Hello");
char* str1 = malloc(len + 1);
strcpy(str1, "Hello");
char* str2 = malloc(len + 1);
strcpy(str2, "Hello");
if (str1 == str2) {
    printf("Strings are equal\n");
} else {
    printf("Strings are NOT equal\n");
}
```

Strings are
NOT equal

Сравнение указателей вместо значений

```
if (strcmp(str1, str2) == 0) {  
    printf("Strings are equal\n");  
}
```

Передача массива в функцию

При передаче массива в функцию происходит копирование адреса, т.к. в языке Си возможна передача только по значению. При копировании теряется размер массива, поэтому необходимо передавать его отдельным аргументом.

```
#include <stdio.h>

void print_array(int a[5]) {
    /* ОШИБКА!    Размер массива теряется */
    for(int i=0; i<sizeof(a)/sizeof(int); i++)
        printf("a[%d]=%d ",i,a[i]);
}

int main(void) {
    int a[5] = {1,2,3,4,5};
    print_array(a);
    return 0;
}
```

```
#include <stdio.h>

void print_array(int a[], int size) {
    /* Так можно */
    for( int i=0; i<size; i++ )
        printf("a[%d]=%d ",i,a[i]);
}

int main(void) {
    int a[5] = {1,2,3,4,5};
    print_array(a, 5);
    /* Так тоже можно */
    for(int i=0; i<sizeof(a)/sizeof(int); i++) {
        printf("a[%d]=%d ",i,a[i]);
    }
    return 0;
}
```


Передача массива в функцию

Если в функцию передаётся двумерный массив(матрица), то внутри можно задать переменную: указатель на строку матрицы, это позволит работать с данным массивом, используя двойной индекс.

```
void print_matrix(int n, int m, int
a[]) {
    int (*pa)[m] = a;
    // Указатель на строку из
    // m элементов типа int
    for( int i=0; i<n; i++ ) {
        for( int j=0; j<m; j++ ) {
            printf("%4d ", pa[i][j]);
        }
        printf("\n");
    }
}
```

```
int main(void) {
    int a[2][3] = {
        {1,2,3},
        {4,5,6}
    };

    print_matrix(2,3,a);
    return 0;
}
```

1	2	3
4	5	6

Передача массива в функцию

Еще один вариант функции: передать размер строки при объявлении типа аргумента. Описание аргументов *n* и *m* должно идти перед описанием указателя на начало матрицы.

```
void print_matrix(int n, int m, int a[][m]) { // Важен  
    только размер строки  
    for( int i=0; i<n; i++ ) {  
        for( int j=0; j<m; j++ ) {  
            printf("%4d ", a[i][j]);  
        }  
        printf("\n");  
    }  
}
```

Отличие массива от указателя

Массивы и указатели в Си — это разные типы, хотя иногда складывается впечатление, что они эквивалентны.

```
#include <stdio.h>

int ar[] = {1,2,3,4,5};
int *par = ar;

int main(void) {
    par[3] = 123;
    ar[2] = 321;
    return 0;
}
```

Отличие массива от указателя

В этом примере обращение к элементу массива происходит одинаково, однако на самом деле отличие все же есть. Рассмотрим пример многомодульной программы, состоящей из двух файлов.

`main.c`

```
extern int *ar;  
  
int main(void) {  
    ar[3] = 123;  
    return 0;  
}
```

`arr.c`

```
int ar[5];
```

Отличие массива от указателя

Однако после сборки, мы получим ошибку: Segmentation fault.

```
:~$ gcc -c -o main.o main.c  
:~$ gcc -c -o arr.o arr.c  
:~$ gcc -o prog main.o arr.o  
:~$ ./prog  
Segmentation fault: 11
```

Создать объектный файл из main.c
Создать объектный файл из arr.c
Линкуем исполняемый файл prog

Основные отличия указателей от массивов:

Указатель	Массив
Содержит адрес, ссылающийся на данные	Содержит данные
Для доступа к данным сначала надо запросить содержащийся в указателе адрес, а потом запросить данные по этому адресу	Прямой доступ к данным. К адресу прибавляется смещение индекса
В основном используется для динамических данных	В основном используется для данных с известным числом элементов
В основном используется с <code>malloc()</code> и <code>free()</code>	Неявное выделение памяти
Часто ссылается на безымянные данные	Именованная переменная

Отличие массива от указателя

Массивы и указатели можно инициализировать литерными строками.

Во втором случае константная строка расположена в read-only памяти и в указателе, сохранён адрес её первого элемента, поэтому изменение такой строки невозможно.

```
char s[] = "Hello world";  
s[0] = 'A'; // Так можно
```

```
char *s = "Hello world";  
s[0] = 'A'; // ОШИБКА!  
read-only string
```

Обнаружение утечек

Рассмотрим пример, в котором мы обнаруживаем утечку своими силами. В данном примере используется макрос, который замкнѐвает стандартный вызов функции `malloc` на вызов функции `my_malloc` и аналогичный макрос для вызова функции `free`. При каждом вызове `malloc` заносится информация в однонаправленный список о новом блоке выделенной памяти. При каждом вызове `free` происходит удаление информации из списка. Если в конце программы в списке остались элементы, то это сигнализирует об утечке памяти.

Обнаружение утечек

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

typedef struct list {
    void *address;
    size_t size;
    char comment[64];
    struct list *next;
} list;

list *memlist = NULL;
```

```
void insert(list **head, void
*address, size_t size, char
*comment) {
    list *tmp =
malloc(sizeof(list));
    tmp->next = *head;
    tmp->address = address;
    tmp->size = size;

    sprintf(tmp->comment, "%s", comm
ent);
    *head = tmp;
}
```

Обнаружение утечек

```
_Bool delete(list **head, void
*address) {
    if(*head == NULL)
        return 0;
    list *del = NULL;
    if( (*head)->address ==
address) {
        del = *head;
        *head = (*head)->next;
        free(del);
        return 1;
    }
```

```
list *tmp = *head;
while( tmp->next ) {
    if( tmp->next->address
== address ) {
        del = tmp->next;
        tmp->next =
del->next;
        free(del);
        return 1;
    }
    tmp=tmp->next;
}
return 0;
}
```

Обнаружение утечек

```
void printList(list *head) {
    if(head == NULL) {
        printf("No memory leak
detect\n");
    }
    while(head) {
        printf("%s\n",head->comment);
        head = head->next;
    }
}
```

```
void* my_malloc(size_t size,
const char *file, int line,
const char *func){
    void *ptr = malloc(size);
    char coment[64] = {0};
    sprintf (coment,"Allocated =
%s, %i, %s, %p[%li]", file,
line, func, ptr, size);

    insert(&memlist,ptr,size,coment)
;

    return ptr;
}
```

Обнаружение утечек

```
void my_free(void *ptr, const
char *file, int line, const char
*func)
{
    free(ptr);
    delete(&memlist, ptr);
}

#define malloc(X) my_malloc( (X),
__FILE__, __LINE__, __FUNCTION__)
#define free(X) my_free( (X),
__FILE__, __LINE__, __FUNCTION__)
```

```
int main(void) {
    int *p = malloc(
sizeof(int) );
    int *ar =
malloc(sizeof(int)*10);
    *p = 5;
    free(p);
    p = malloc(sizeof(int));

    free(p);
    printList(memlist);
    return 0;
}
```