

# Основы программирования

ФИСТ 1 курс

Власенко

Олег

Федосович

Лекция 12

**Память и работа с памятью в Си:**

Оперативная память. Загрузка и выполнение программы. Код программы.

Разделы памяти: автоматическая, статическая, динамическая память.

Адрес. Указатель. Операции взятия адреса & и разыменования \*.

[ Ассемблер и машинный код. Трансляция программы из ЯВУ в ассемблер. ]

Классы памяти: auto, static, register, extern.

Выделение и освобождение динамической памяти.

# Оперативная память

[https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F\\_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C)



## *Схемотехника ЭВМ*

С точки зрения программиста, оперативная память — это одномерный массив байт:

```
char memory[0x100000000];
```

# Загрузка и выполнение программы

[https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D1%87%D0%B8%D0%BA\\_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC)

«При запуске новой программы загрузчик должен:

- Считать данные из запускаемого файла.
- Если необходимо — загрузить в память недостающие динамические библиотеки.
- Заменить в коде новой программы относительные адреса и символические ссылки на точные, с учётом текущего размещения в памяти, то есть выполнить связывание адресов (англ. *Relocation*).
- Создать в памяти образ нового процесса и запланировать его к исполнению.»

**Операционные системы**

**Системное программное обеспечение**

# Код программы

Машинный код:

<https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9%D0%BA%D0%BE%D0%B4>

*Организация ЭВМ и систем*

# Разделы памяти: автоматическая, статическая, динамическая память

«Размещение объектов в оперативной памяти. Понятие указателя»

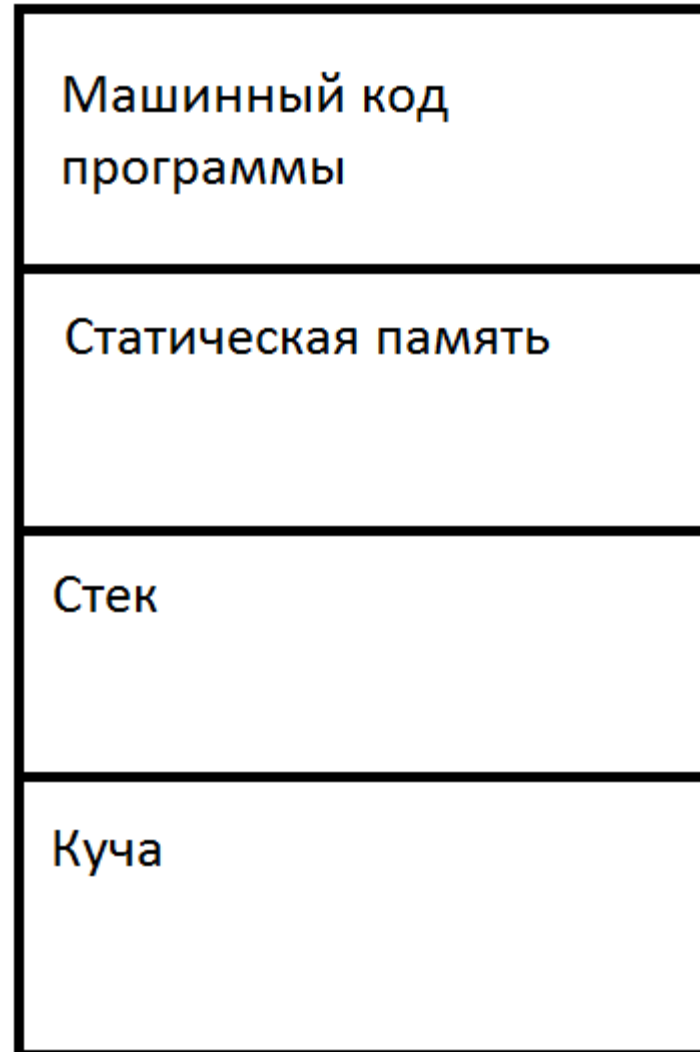
<https://rstdn.org/article/cpp/ObjectsAndPointers.xml?print>

**Статическая память** — это область памяти, *выделяемая при запуске программы до вызова функции `main`* из свободной оперативной памяти для размещения глобальных и статических объектов, а также объектов, определённых в пространствах имён.

**Автоматическая память** — это специальный регион памяти, *резервируемый при запуске программы до вызова функции `main`* из свободной оперативной памяти и *используемый в дальнейшем* для размещения локальных объектов: объектов, определяемых в теле функций и получаемых функциями через параметры в момент вызова. Автоматическую память часто называют **стеком**.

**Динамическая память** — это совокупность блоков памяти, выделяемых из доступной свободной оперативной памяти непосредственно *во время выполнения программы* под размещение конкретных объектов.

# Структура памяти программы во время выполнения



# Адрес. Указатель. Операции взятия адреса & и разыменования \*.

```
void main() {  
    // char memory[0x100000000];  
    int a;  
    int b;  
    int * p1;  
    int * p2;  
    a = 10;  
    b = 20;  
    printf("%d %d\n", a, b);  
    p1 = &a;  
    p2 = p1;  
    *p1 = 30;  
    printf("%d %d\n", a, b);  
}
```

# Трансляция программы из ЯВУ в ассемблер.

```
0002f  c7 45 e8 14 00
          00 00          mov     DWORD PTR _b$[ebp], 20; 00000014H
```

```
0004b  8d 45 f4  lea      eax, DWORD PTR _a$[ebp]
0004e  89 45 dc  mov      DWORD PTR _p1$[ebp], eax
```

>> Добрый день! Скажите, пожалуйста, можно ли в VS 2012 pro посмотреть ассемблерный код написанной программки? ...

**> Project Properties > Configuration Properties > C/C++ > Output Files > Assembler output**



# Классы памяти: auto, static, register, extern

Спецификаторы классов памяти extern, auto, register, static используются для изменения способа создания памяти для переменных в языках С (и С++).

Эти спецификаторы ставятся перед именем типа, который они модифицируют.

- **auto**
- **static**
- **register**
- **extern**

[http://ic.asf.ru/~docs/cpp/cppd\\_qualifier.htm](http://ic.asf.ru/~docs/cpp/cppd_qualifier.htm)

# auto и static

- Спецификатор auto уведомляет компилятор о том, что локальная переменная, перед именем которой он стоит, создается при входе в блок и разрушается при выходе из блока. Все переменные, определённые внутри функции, являются автоматическими по умолчанию, и поэтому ключевое слово auto используется крайне редко.
- Спецификатор static указывает компилятору на хранение локальной переменной во время всего жизненного цикла программы вместо ее создания и разрушения при каждом входе в область действия и выходе из неё. Следовательно, возведение локальных переменных в ранг статистических позволяет поддерживать их значения между вызовами функций.

```
void f() {  
    static int a = 3;  
    auto int b = 3;  
    a++;  
    b++;  
}
```

# auto и static (2)

Что Вы увидите на экране?

```
void f() {  
    static int a = 3;  
    auto int b = 3;  
    a++;  
    b++;  
    printf("%d %d\n", a, b);  
}
```

```
void main() {  
    f(); // Вывод: 4 4 \n  
    f(); // ????  
    f(); // ????  
}
```

# register

- Когда язык C был только изобретён, спецификатор `register` можно было использовать лишь для локальных целых или символьных переменных, поскольку он заставлял компилятор пытаться сохранить эту переменную в регистре центрального процессора вместо того, чтобы её просто разместить в памяти. В таком случае все ссылки на переменную работали исключительно быстро. С тех пор определение спецификатора расширилось. Теперь любую переменную можно определить как `register` и тем самым возложить заботу об оптимизации доступа к ней на компилятор. Для символов и целых это по-прежнему означает их хранение в регистре процессора, но для других типов данных, это может означать, например, использование кеш-памяти. Следует иметь в виду, что использование спецификатора `register` — это всего лишь заявка, которая может быть и не удовлетворена. Компилятор волен её проигнорировать. Причина этого состоит в том, что только ограниченное число переменных можно оптимизировать ради ускорения обработки данных. При превышении этого предела компилятор будет просто игнорировать дальнейшие `register`-"заявки".

## register (2)

```
double get_average_even(int arr[NUM]) {
```

```
    register int s = 0;
```

```
    register int cnt_even = 0;
```

```
    register int i = 0;
```

```
    while (i < NUM) {
```

```
        if (arr[i] % 2 == 0) {
```

```
            s += arr[i];
```

```
            cnt_even++;
```

```
        }
```

```
        i++;
```

```
    }
```

```
    return s / cnt_even;
```

```
}
```

# extern

- Если спецификатор `extern` размещается перед именем переменной, компилятор будет "знать", что переменная имеет внешнюю привязку, т.е. что память для этой переменной выделена где-то в другом месте программы. Внешняя "привязка" означает, что данный объект виден вне его собственного файла. По сути, спецификатор `extern` сообщает компилятору лишь тип переменной, но не выделяет для неё области памяти. Чаще всего спецификатор `extern` используется в тех случаях, когда одни и те же глобальные переменные используются в двух или более файлах.

```
extern int cnt_call;
```

# extern (2)

// Файл main.c

```
#define _CRT_SECURE_NO_WARNINGS
```

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

```
extern int cnt_call;
```

```
void call();
```

```
void main() {  
    printf("cnt_call = %d\n", cnt_call);  
    call();  
    printf("cnt_call = %d\n", cnt_call);  
    call();  
    printf("cnt_call = %d\n", cnt_call);  
}
```

# extern (3)

```
// Файл call.c
```

```
#define _CRT_SECURE_NO_WARNINGS
```

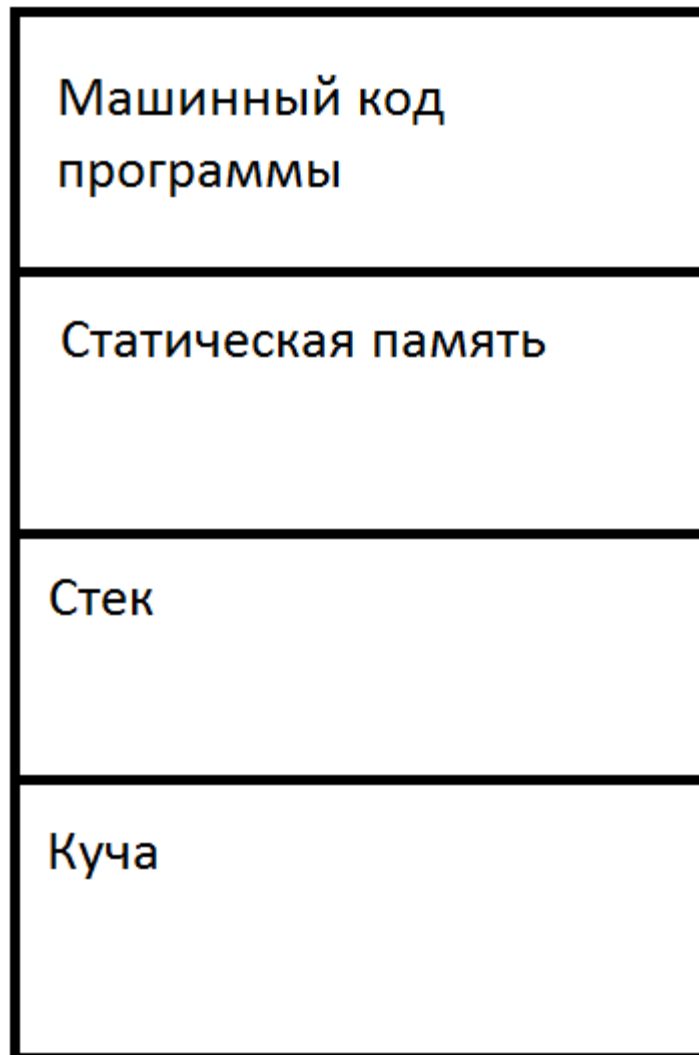
```
#include <stdio.h>
```

```
int cnt_call = 10;
```

```
void call() {  
    printf("call!!!\n");  
    cnt_call++;  
}
```



# Куда попадут переменные?



# В какой раздел памяти попадут переменные?

```
int s;
```

```
int f(int *arr, int n) {  
    static int cnt = 0;  
    int s = 0, i;  
    for (i = 0; i < n; i++) s += arr[i];  
    cnt++;  
    return s;  
}
```

```
void main() {  
    int *a = (int*)malloc(5 * sizeof(int));  
    a[0] = 10; a[1] = 15; a[2] = 25; a[3] = 30; a[4] = 40;  
    s = f(a, 5);  
    printf("s = %d", s);  
    free(a);  
}
```

# Работа с динамической памятью

# Указатель void \*

```
#include <stdio.h>

void main() {
    int a = 20;
    int * pa = &a;
    void * p = pa;
    int * p2 = (int *)p;
    printf("a = %d\n", a);
    *pa = 25; // a = 25;
    printf("a = %d\n", a);
    // *p = 30;
    *p2 = 30; // a = 30;
    printf("a = %d\n", a);
}
```

# Выделение и освобождение динамической памяти

[http://learnc.info/c/memory\\_allocation.html](http://learnc.info/c/memory_allocation.html)

Для выделения памяти на куче в си используется функция malloc (memory allocation)

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

После того, как мы поработали с памятью, необходимо освободить память функцией free.

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

# Задача: Обработка оценок по контрольной для произвольной группы

## Задача:

Количество студентов в группе задается перед началом обработки.

Нужно подсчитать среднее арифметическое оценки по контрольной для группы. И отдельно вывести оценки ниже и выше средних с указанием индексов.

## Пример ввода:

5

2 3 4 5 1

## Вывод:

3

2(0) 1(4)

4(2) 5(3)

# Задача (1)

```
#define _CRT_SECURE_NO_WARNINGS
```

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

```
void main() {
```

```
    int * a;
```

```
    int n;
```

```
    int i;
```

```
    int s;
```

```
    double average;
```

```
    printf("n=");
```

```
    scanf("%d", &n);
```

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

## Задача (2)

```
printf("Please input %d numbers:", n);
```

```
i = 0;
```

```
while (i < n) {
```

```
    scanf("%d", &a[i]);
```

```
    i++;
```

```
}
```

```
s = 0;
```

```
i = 0;
```

```
while (i < n) {
```

```
    s += a[i];
```

```
    i++;
```

```
}
```

```
average = s / (double)n;
```

```
printf("%4.1lf\n", average);
```



## Задача (3)

```
i = 0;
while (i < n) {
    if (a[i] < average) {
        printf("%d(%d) ", a[i], i);
    }
    i++;
}
printf("\n");
i = 0;
while (i < n) {
    if (a[i] > average) {
        printf("%d(%d) ", a[i], i);
    }
    i++;
}
printf("\n");
```

## Задача (4)

`free(a);`

`}`

## Задача 2

Из файла in4.txt загрузить массив из N целых чисел.

Во входном файле есть 2 строки. 1 строка содержит число N. Вторая строка содержит N чисел.

Все элементы, стоящие после первого четного увеличить в 10 раз.

Получившийся массив вывести в файл out4.txt в формате, аналогичном входному файлу.

Использовать динамический массив.

**Пример входа:**

**3**

**2 3 4**

**Пример выхода:**

**3**

**2 3 40**

## Задача 2 (1)

```
void main() {  
    int n;  
    int *elements = read_array(&n);  
    if (elements == NULL) {  
        return;  
    }  
    int i = find_first_odd(elements, n) + 1;  
    while (i < n) {  
        elements[i] *= 10;  
        i++;  
    }  
    write_array(elements, n);  
    free(elements);  
}
```

## Задача 2 (2)

```
int find_first_odd(int a[], int n) {  
    int i = 0;  
    while (i < n) {  
        if (a[i] % 2 == 1) {  
            return i;  
        }  
        i++;  
    }  
    return i;  
}
```

## Задача 2 (3)

```
int* read_array(int *pn) {  
    FILE *fin = fopen("c:\\Temp\\Files\\in4.txt", "rt");  
    if (fin == NULL) {  
        printf("File in4.txt is not found");  
        return NULL;  
    }  
    fscanf(fin, "%d", pn);  
    int *pa = (int*)malloc(*pn * sizeof(int));  
    int i = 0;  
    while (i < *pn) {  
        fscanf(fin, "%d", &pa[i]);  
        i++;  
    }  
    fclose(fin);  
    return pa;  
}
```

## Задача 2 (4)

```
void write_array(int a[], int n) {  
    FILE * fout = fopen("c:\\Temp\\Files\\out4.txt", "wt");  
    if (fout == NULL) {  
        printf("File out4.txt cannot be created");  
        return;  
    }  
    fprintf(fout, "%d\\n", n);  
    int i = 0;  
    while (i < n) {  
        fprintf(fout, "%d ", a[i]);  
        i++;  
    }  
    fclose(fout);  
}
```

# Домашнее задание

1. Изучить информацию по Выделению и освобождению динамической памяти:  
[http://learnc.info/c/memory\\_allocation.html](http://learnc.info/c/memory_allocation.html)