



## Лекция №4

# Оптимизация кода

Продвинутый курс Си



# План курса

---

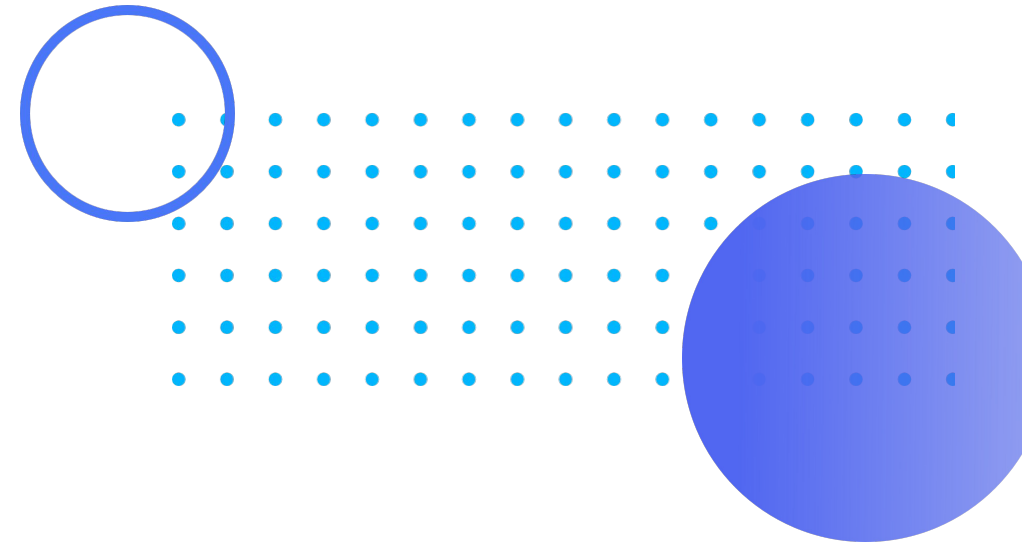
- Вводный урок
- Структуры. Динамические типы
- Библиотеки языка C
- **Оптимизация кода**
- Алгоритмы
- Компиляция и компиляторы
- Динамические структуры данных
- Курсовая работа



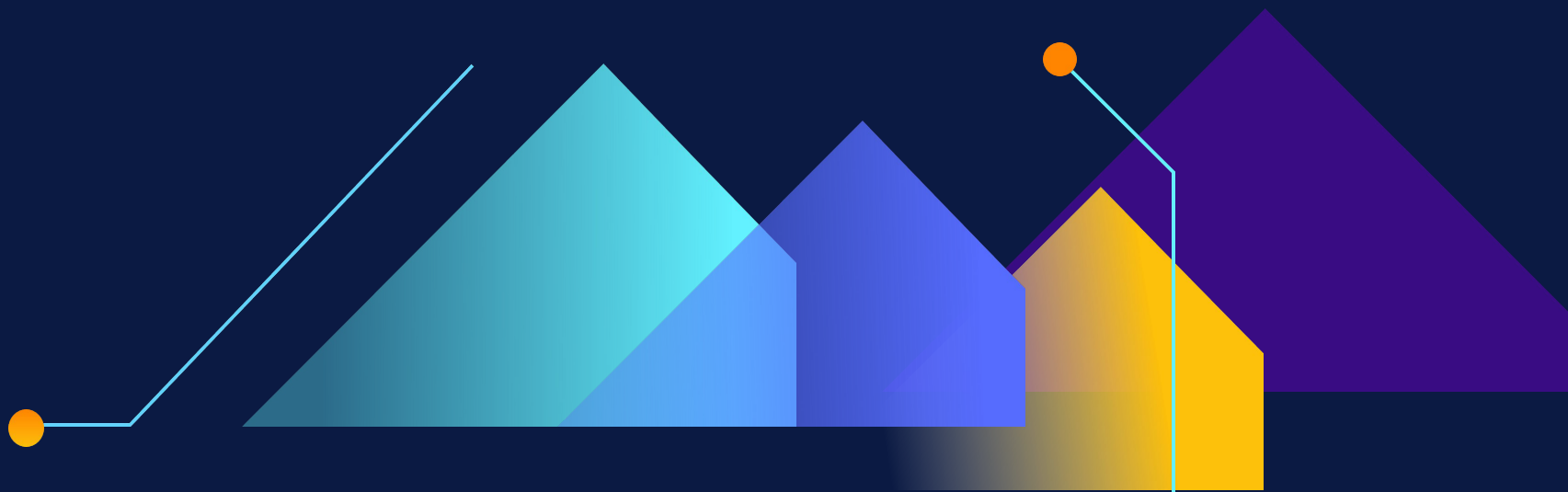
# Маршрут

## Оптимизация кода

- Узнаем, какие есть способы оптимизации алгоритмов, и что они делают
- Разберём инструмент оптимизации кода — профайлер
- Обсудим, когда стоит оптимизировать код, а когда нет



# Теория оптимизации



# Оптимизация

---

**Оптимизация программ** – процесс улучшения программы в части конкретных критериев. А не стремление к поиску идеального варианта, т.е. варианта, который нельзя улучшить.

Цель оптимизации программы – получение из работающего варианта программы другого работающего варианта, обладающего желаемыми критериями

Основными критериями при оптимизации программ являются, например:

- Скорость работы
- Объем используемой памяти
- Объем места, занимаемого на диске

# Когда не нужна оптимизация?

---

Следует помнить, что оптимизация — это просто пустая трата времени программиста, если любое из этих утверждений верно:

- Часть программы еще не написана
- Программа не полностью протестирована и отлажена
- Кажется, что программа уже работает достаточно быстро

# Когда не нужна оптимизация?

---

Необходимо принять во внимание, как программа будет использоваться:

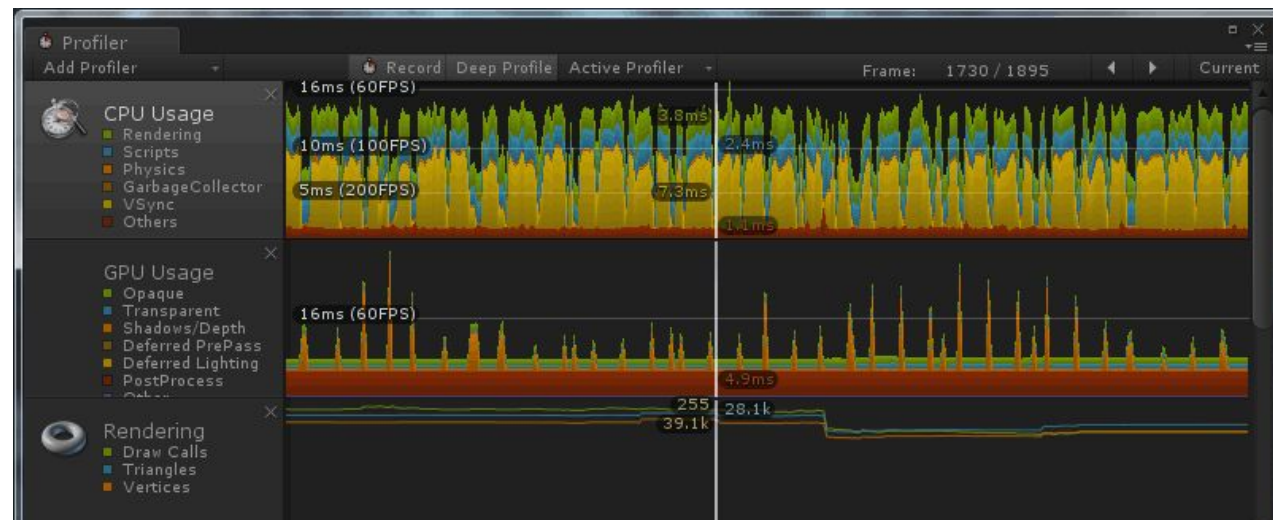
- Программа запускается редко и в фоновом режиме.
- Программа вызывается из более медленной программы.
- Если программа работает в режиме реального времени - оптимизация нужна

# Оптимизация

---

**Профайлер** — это специализированный программный инструмент, который собирает характеристики работы какой-то программы.

**Самый эффективный метод оптимизации** — это использование профайлера для выявления узких мест в программе. Бывает трудно угадать, какая часть вашей программы потребляет больше всего ресурсов, и если вы основываетесь на предположениях, а не на реальных данных, то потратите много времени на ускорение тех частей вашей программы, которые уже и так работали быстро.

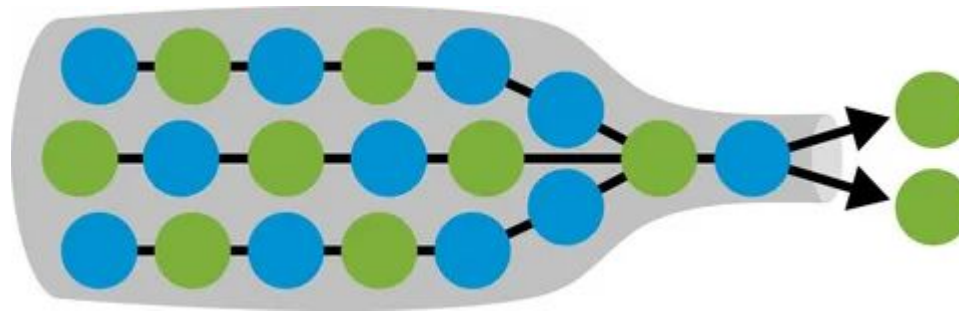




# Bottleneck

---

Как только вы определили узкое место **bottleneck** (например, цикл, который выполняется тысячу раз), помните, что лучше всего перепроектировать программу так, чтобы ей не приходилось выполнять цикл тысячу раз. Это более эффективно, чем заставлять цикл работать на 10% быстрее.



# Когда нужна оптимизация?

---

Алгоритм оптимизации:

- Определите “узкое место” (например, цикл, который выполняется тысячу раз)
- Лучше всего алгоритмически перепроектировать программу так, чтобы ей не приходилось выполнять цикл тысячу раз
- Это более эффективно, чем заставлять цикл работать на 10% быстрее

Сначала напишите программу, потом оптимизируйте!

# Рекомендации

---

Также необходимо принять во внимание, как программа будет использоваться. Если это программа-генератор отчётов, которую нужно запускать только один раз в день, пользователь может запустить её перед тем, как отправится на обед, и в таком случае действительно нет смысла завершать её до того, как обед закончится, и пользователь вернётся.

Если он вызывается из другой программы, которая даже медленнее, чем ваша, то пользователь снова не заметит разницы. Но если программа обрабатывает события отслеживания мыши для графического интерфейса, то пользователи будут жаловаться на любую заметную задержку.

Учитывая, что оптимизация является разумной, скомпилируйте в режиме полной оптимизации и запустите вашу программу на «реальных» входных данных. Если у вас нет доступа к реальным входным данным, то постарайтесь предусмотреть тестовые входные данные, которые могли бы покрыть как можно больше узких мест.

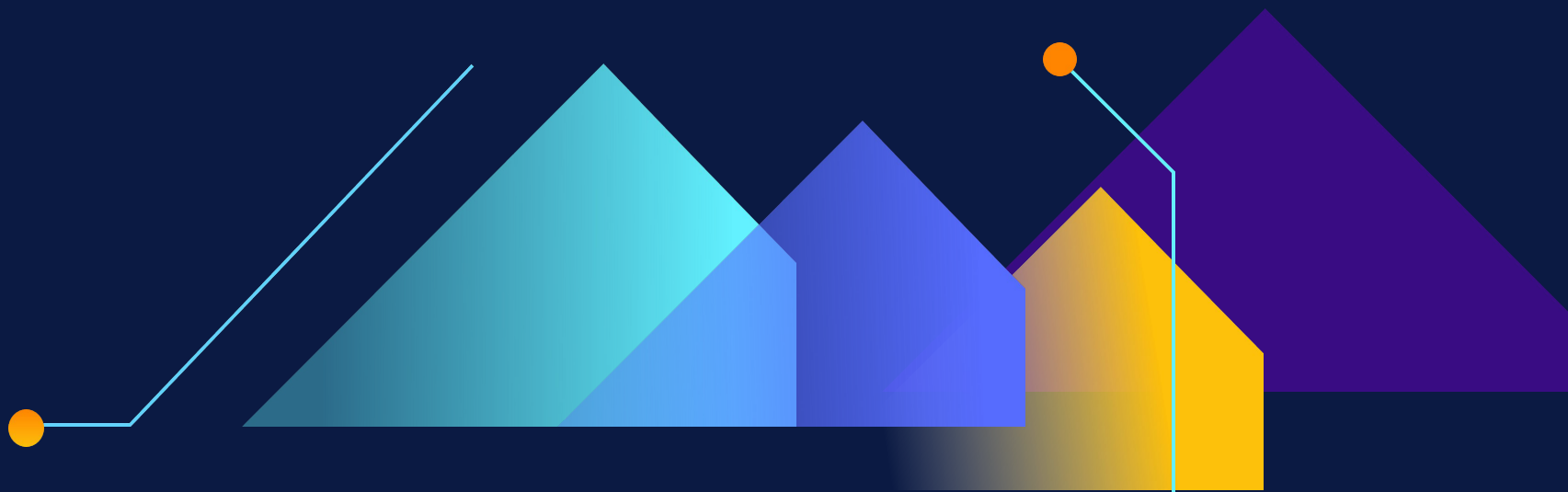
# Анализ производительности

---

- Используйте функции **time()** и **clock()** чтобы узнать, время работы критического кода. Даже если программа «кажется» занимает много времени, это может быть только доли секунд реального времени.
- Задержки могут быть вызваны: вычислениями, памятью или вводам-выводом.
- В вашей ОС могут быть команды для отслеживания времени работы **time**. Иногда они встроены в оболочку (например, **csch**) и имеют множество изящных опций.
- Вы также можете получить информацию о производительности из **getrusage()**, если он у вас есть, и, конечно, из программ профилирования, таких как **gprof**, **prof** и **tcov**.

# Измерение времени

## Windows



# Измерение времени

Функция `GetProcessTimes()` заполняет структуру `FILETIME` процессорным временем.

Функция `FileTimeToSystemTime()` конвертирует структуру `FILETIME` в структуру `SYSTEMTIME`, содержащую значение времени. [C/C++: как измерять процессорное время](#)

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

```
FILETIME createTime, exitTime, kernelTime,
userTime;
if (GetProcessTimes(GetCurrentProcess(), &createTime
, &exitTime, &kernelTime, &userTime) != -1)
{
    SYSTEMTIME userSystemTime;
    if (FileTimeToSystemTime(&userTime,
                             &userSystemTime) != -1)
        return (double)userSystemTime.wHour * 3600.0;
}
```

# Пример использования

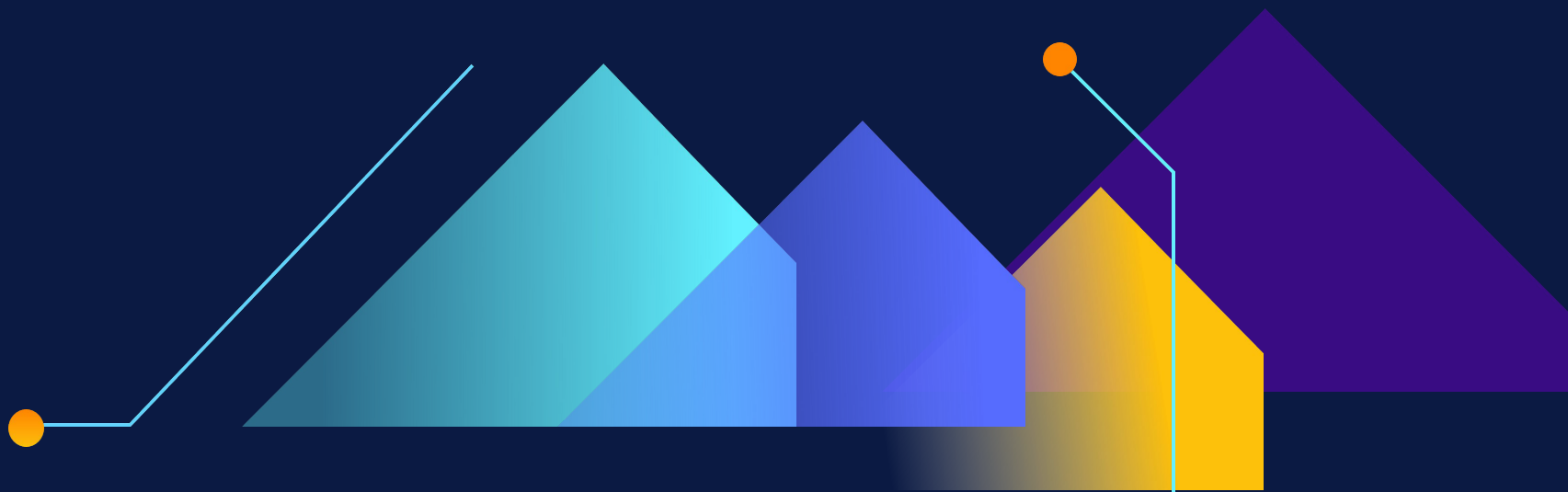
```
#include <Windows.h>

double getCPUtime(void)
{ /* Windows */
    FILETIME createTime, exitTime, kernelTime,
    userTime;
    if(GetProcessTimes(GetCurrentProcess(),
        &createTime, &exitTime, &kernelTime,
        &userTime) != -1 )
    {
        SYSTEMTIME userSystemTime;
        if(FileTimeToSystemTime(&userTime,
            &userSystemTime) != -1 )
            return (double)userSystemTime.wHour * 3600.0 +
                (double)userSystemTime.wMinute * 60.0 +
                (double)userSystemTime.wSecond +
                (double)userSystemTime.wMilliseconds
                    / 1000.0;
    }
    return -1; /* Failed. */
}
```

```
int main(int argc, char **argv)
{
    double startTime, endTime;
    startTime = getCPUtime( );
    for(int i=0;i<10000000;i++)
    {}
    endTime = getCPUtime( );
    printf( "CPU time used =
%lf\n",
                (endTime-startTime)
    );
    return 0;
}
```

# Измерение времени

Linux





# Оптимизация

---

Рассмотрим пример использования `getrusage()`. Данная функция возвращает измерения в структуре `usage`, которая имеет различные поля. Вот некоторые из них:

- `ru_utime` Общее количество времени, проведенное в режиме пользователя, выражается структурой `timeval` (секунды и микросекунды).
- `ru_stime` Общее количество времени, проведенное в режиме ядра, выражается структурой `timeval` (секунды и микросекунды).
- `ru_maxrss` Максимальный используемый размер постоянно занимаемый в памяти (в байтах).

# Пример getrusage()

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    struct rusage usage;
    struct timeval startu, endu, starts, ends;
    int i, j, k = 0;
    getrusage(RUSAGE_SELF, &usage);
    startu = usage.ru_utime; // измерение в режиме пользователя
    starts = usage.ru_stime; // измерение в режиме системы
    //int arr[100000]={0}; // Попробуйте раскомментировать и сравнить
    for (i = 0; i < 10000; i++) {
        for (j = 0; j < 10000; j++) {
            k += 20;
        }
    }
}
```

# Пример getrusage()

```
getrusage(RUSAGE_SELF, &usage);  
endu = usage.ru_utime; // измерение в режиме пользователя  
ends  = usage.ru_stime; // измерение в режиме системы  
  
printf("Started at user mode: %ld.%d\n", startu.tv_sec,  
startu.tv_usec);  
printf("Ended at user mode: %ld.%d\n", endu.tv_sec,  
endu.tv_usec);  
printf("Started at system mode: %ld.%d\n", starts.tv_sec,  
starts.tv_usec);  
printf("Ended at systme mode: %ld.%d\n", ends.tv_sec,  
ends.tv_usec);  
printf("Total memory usage: %ld bytes\n", usage.ru_maxrss);  
return 0;  
}
```

# Пример

Рассмотрим пример: дан массив из целых чисел. Каждое число встречается ровно два раза, но есть одно, которое встречается только один раз. Необходимо найти его. Напишем решение в «лоб» и замерим время:

```
enum {SIZE=2001};
struct timespec tstart, tstop;
int a[SIZE] = {1,2,... 1000,12345,
               1,2, ..., 1000};

int main() {
    struct rusage usage;
    struct timeval startu, endu,
starts, ends;
    int same = 0;
    getrusage(RUSAGE_SELF, &usage);
    // измерение в режиме пользователя
```

```
    startu = usage.ru_utime;
    // измерение в режиме системы
    starts = usage.ru_stime;
    _Bool is_same=0;
    for (size_t i=0; i<SIZE; i++){
        is_same=0;
        for (size_t j=0; j<SIZE; j++){
            if(i!=j && a[i]==a[j])
                is_same=1;
        }
        if(!is_same)
            same=a[i];
    }
```

# Пример

```
getrusage(RUSAGE_SELF, &usage);
endu = usage.ru_utime;
ends    = usage.ru_stime;
printf("Started at user mode: %ld.%d\n",
startu.tv_sec, startu.tv_usec);
printf("Ended at user mode: %ld.%d\n",
endu.tv_sec, endu.tv_usec);
printf("Started at system mode:
%ld.%d\n", starts.tv_sec, starts.tv_usec);
printf("Ended at system mode: %ld.%d\n",
ends.tv_sec, ends.tv_usec);
return 0;
}
```

Started at user mode: 0.2087  
Ended at user mode: 0.11931  
Started at system mode: 0.3093  
Ended at system mode: 0.3162

# Пример

Более элегантное решение с использованием побитовой операции XOR:

```
same = 0;  
for (size_t i=0; i<SIZE; i++)  
    same ^= a[i];
```

Started at user mode: 0.2180

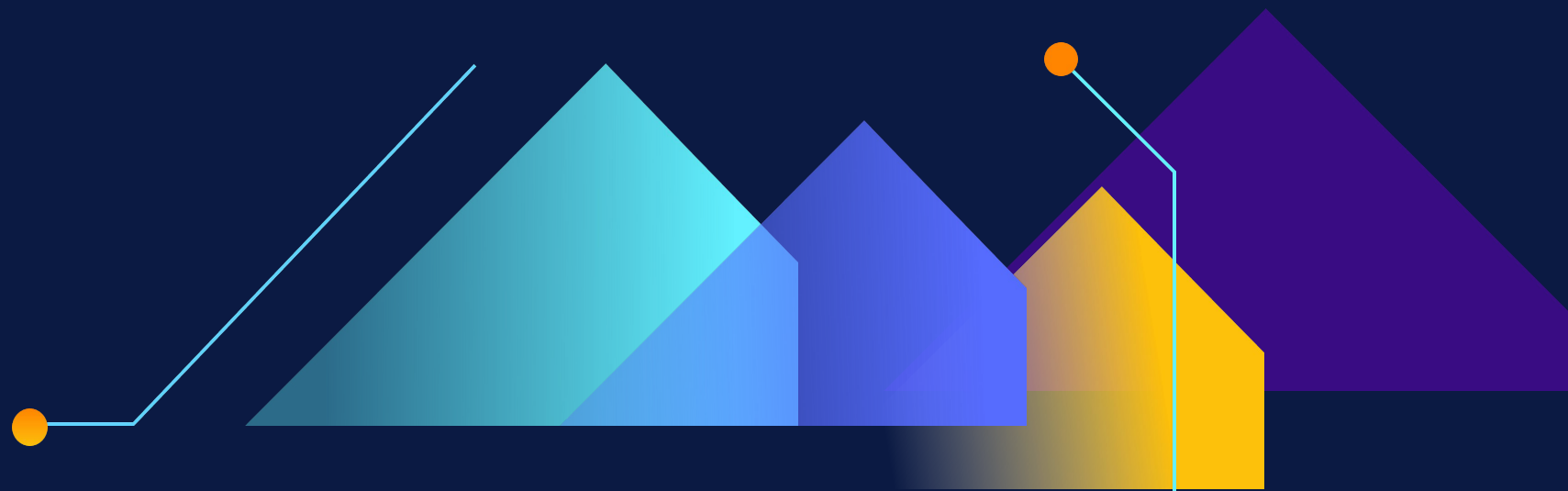
Ended at user mode: 0.2192

Started at system mode: 0.2648

Ended at system mode: 0.2653

Ускорение превышает 100 раз.

# Макросы



# Вызов функций

---

Хотя функции и модульность — это хорошо, вызов функции внутри часто выполняемого цикла является возможным узким местом. Помимо затрат на выполнение инструкций в другой функции:

- Вызов функций радикально прерывает ход мыслей оптимизатора. Любые ссылки через указатели или на глобальные переменные теперь «грязные» и должны быть сохранены / восстановлены во время вызова функции. Локальные переменные, адрес которых был взят и передан за пределы функции, теперь также загрязнены, как отмечалось выше.
- Сам вызов функции связан с некоторыми накладными расходами, поскольку необходимо управлять стеком и изменять счётчик программы с помощью любого механизма, используемого ЦП. Сохранять и восстанавливать регистры на стеке.



# Макросы

---

При вызове функции происходит переход по адресу функции и под неё выделяется новый **стековый кадр** — передача аргументов и выделение временной памяти с использованием системного стека.

- Если мы делаем макрос, то будет выполнена макроподстановка и никакого вызова функции не произойдет
- Если необходимо, то можно переписать небольшие функции в виде макросов. Однако это стоит делать только после того, как вся программа отлажена и проверена (т.к. отладчик не сможет обработать макросы).

**Внимание!** Отладчик не сможет обработать макросы!

# Пример

```
// До
int foo(int32_t a, int32_t b)
{
    a = a - b;
    b++;
    a = a * b;
    return a;
}
```

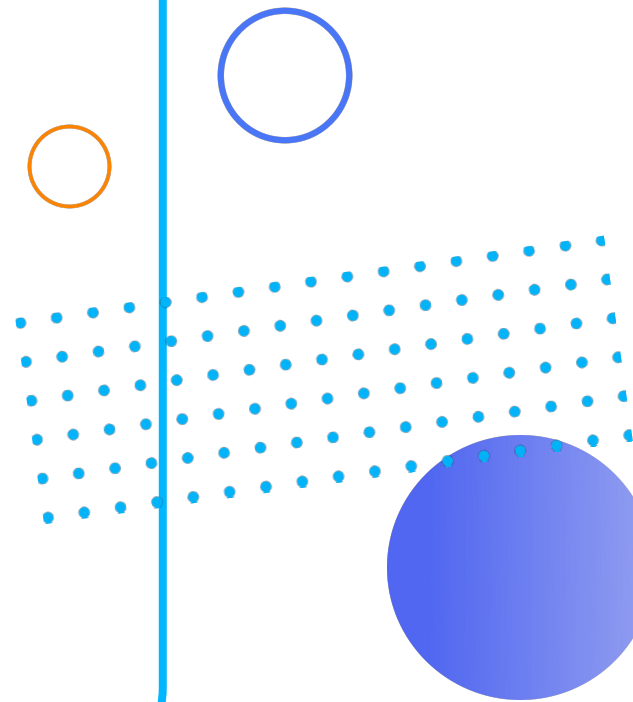
```
// После
#define foo(a, b) (((a) - (b)) *  
((b) + 1))
```



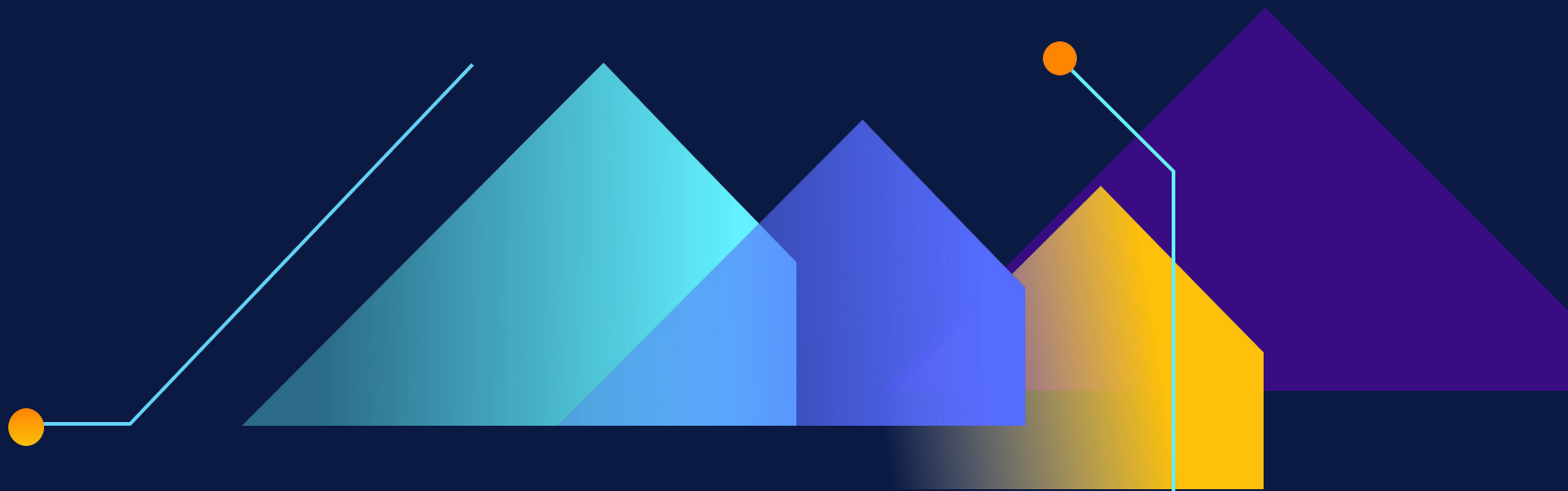
# Особенности

При составлении таких макросов необходимо помнить об особенностях, связанных с их написанием.

- Используйте круглые скобки ( ), выделяя ими как само тело макроса, так и все его аргументы
- Избегайте написания макросов, в телах которых аргументы вычисляются более одного раза, а также передачи в макросы в качестве фактических аргументов выражений с побочным эффектом
- Необоснованное превращение каждой функции в макрос приводит к огромному раздутию кода и может значительно увеличить объём памяти, необходимый программе
- Если макрос содержит сложные операторы, то оптимизатору будет трудно их понять
- Обычно существует ограничение на количество символов в макросе
- Профайлер не сможет проанализировать макрос



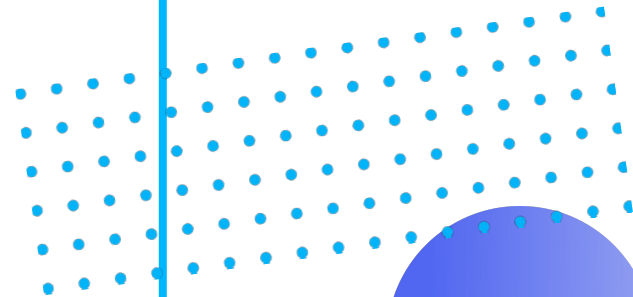
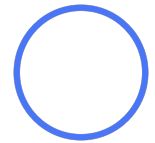
# Inline функции





## inline-функции

В языке Си есть возможность объявлять **встраиваемые** функции. При компиляции вызов функции будет заменён её телом.



# Пример inline-функции

```
#include <stdio.h>

inline int foo() {
    return 2;
}

int main() {
    int ret;

    ret = foo();

    printf("Result: %d\n", ret);
    return 0;
}
```

# Ошибка на этапе линковки

```
$ gcc -o prog main.c
Undefined symbols for architecture x86_64:
  "_foo", referenced from:
      _main in main-05d05d.o
ld: symbol(s) not found for architecture x86_64
error: linker command failed with exit code 1 (use -v to
see invocation)
```

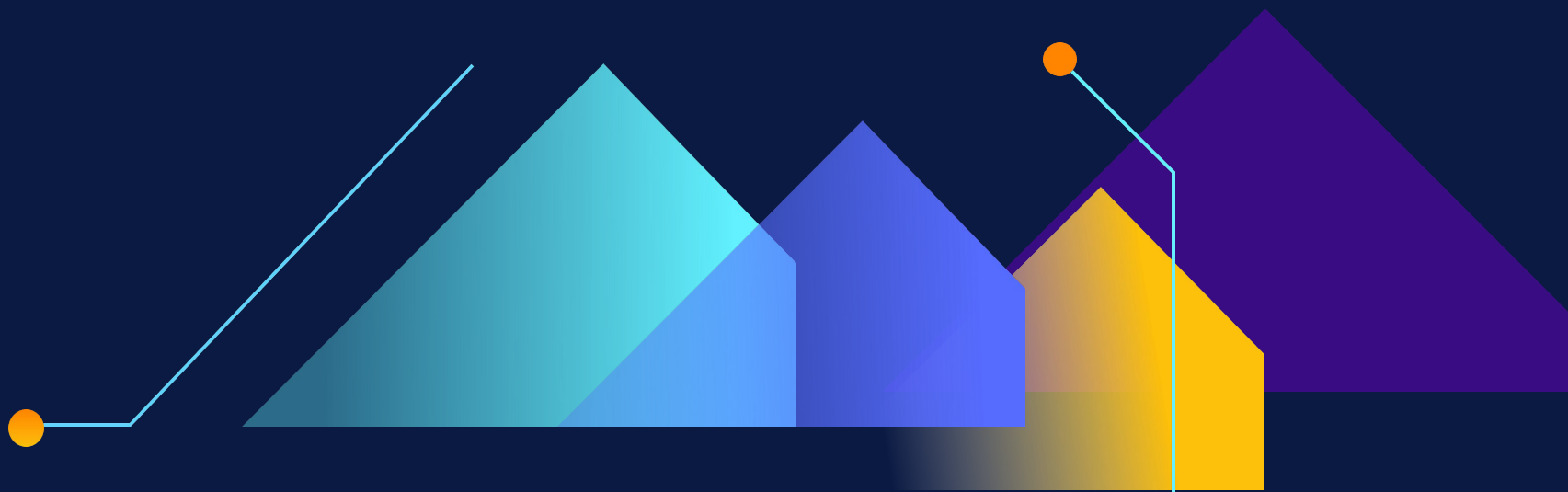
# Что исправить?

```
...  
static inline __attribute__((always_inline)) inline int  
foo() {  
    return 2;  
}  
...
```

Когда функция определена как inline, то все вызовы функции в виде её тела интегрированы в вызывающий код. Адрес функции не используются. Нет ссылок к собственному ассемблерному коду функции.



# Оптимизация циклов



# Развёртывание циклов. Unrolling

---

Развертывание цикла используется для уменьшения количества инструкций перехода, которые потенциально могут ускорить цикл, но это также может увеличить размер двоичного файла.

```
for (size_t i = 0; i < 100; i++)
```

При развертывании цикла происходит уменьшение или исключение инструкций, управляющих циклом, таких как:

- арифметические операции с указателями и операция проверки «конца цикла» на каждой итерации;
- уменьшаются накладные расходы при условных переходах;
- сокращаются задержки, включая задержку чтения данных из памяти.

<https://chipenable.ru/index.php/programming-avr/item/179>

## Пример развертывания цикла

Развёрнутый цикл больше, чем «свернутая» версия, и поэтому может не помещаться в кэш инструкций (на машинах, на которых они есть). Это в итоге замедлит работу развёрнутой версии. Кроме того, в этом примере вызов `func()` затмевает стоимость цикла, поэтому любая экономия от развёртывания цикла незначительна по сравнению с тем, что вы могли бы получить от `inline`.

```
for (size_t i = 0; i < 100; i++)
{
    func(i);
}
```

[illegible]

	Loops	Unrolling loops
C source code	<p>Обычный цикл</p> <pre> #include &lt;avr/io.h&gt;  int main(void) {     uint8_t loop_cnt = 10;     do {         PORTB ^= 0x01;     } while (--loop_cnt); } </pre>	<p>Развернутый цикл</p> <pre> #include &lt;avr/io.h&gt;  int main(void) {     PORTB ^= 0x01;     PORTB ^= 0x01;     PORTB ^= 0x01;     PORTB ^= 0x01;     PORTB ^= 0x01;     PORTB ^= 0x01;     PORTB ^= 0x01;     PORTB ^= 0x01;     PORTB ^= 0x01;     PORTB ^= 0x01; } </pre>
AVR Memory Usage	Program: 94 bytes (1.5% full) (.text + .data + .bootloader) Data: 0 bytes (0.1% full) (.data + .bss + .noinit)	Program: 142 bytes (1.7% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Cycle counter	80	50
Compiler optimization level	-O2	-O2

Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers

# Объединение циклов

Идея состоит в том, чтобы объединить соседние циклы, которые работают в одном и том же диапазоне с одной и той же переменной. Предполагая, что во втором цикле нет использование следующих элементов (например,  $a[i + 3]$ ):

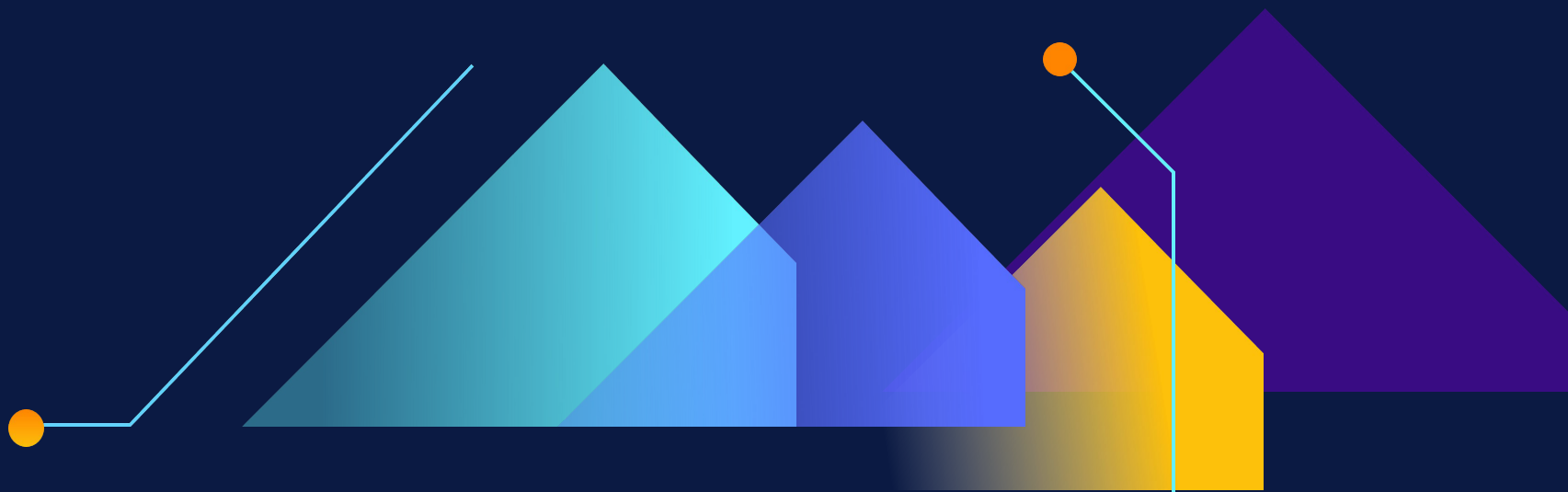
```
for (size_t i = 0; i < MAX; i++) {  
    for (j = 0; j < MAX; j++) {  
        a[i][j] = 0.0;  
    }  
}  
for (size_t i = 0; i < MAX; i++) {  
    a[i][i] = 1.0;  
}
```

```
for (i = 0; i < MAX; i++) {  
    for (j = 0; j < MAX; j++)  
        a[i][j] = 0.0;  
    a[i][i] = 1.0;  
}
```

	Separate loops	Loop jamming
C source code	<pre> #include &lt;avr/io.h&gt;  int main(void) {     uint8_t i, total = 0;     uint8_t tmp[10] = {0};      for (i=0; i&lt;10; i++) {         tmp [i] = ADCH;     }     for (i=0; i&lt;10; i++) {         total += tmp[i];     }     UDR0 = total; } </pre> <p>Обычный ЦИКЛ</p>	<pre> #include &lt;avr/io.h&gt;  int main(void) {     uint8_t i, total = 0;     uint8_t tmp[10] = {0};      for (i=0; i&lt;10; i++) {         tmp [i] = ADCH;         total += tmp[i];     }     UDR0 = total; } </pre> <p>Объединенный ЦИКЛ</p>
AVR Memory Usage	Program: 164 bytes (2.0% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)	Program: 98 bytes (1.2% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers

# Оптимизация переменных



# Оптимизация переменных

Используйте как можно меньший применимый тип данных.

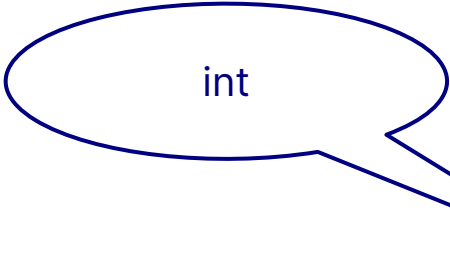

Для чтения 8-битного (байтового) значения из регистра требуется переменная размером в один байт, а не двухбайтовая переменная, это позволяет экономить память

Data type		Size
signed char / unsigned char	int8_t / uint8_t	8-bit
signed int / unsigned int	int16_t / uint16_t	16-bit
signed long / unsigned long	int32_t / uint32_t	32-bit
signed long long / unsigned long long	int64_t / uint64_t	64-bit



# Пример реализации на микроконтроллерах AVR

В левом примере мы пользуемся 2-х байтным типом данных для временной переменной и возвращаемого значения. В правом примере вместо этого используется однобайтный тип char.

	Unsigned int (16-bit)	Unsigned char (8-bit)
C source code	<pre>#include &lt;avr/io.h&gt;  unsigned int readADC() {     return ADCH; };  int main(void) {     unsigned int mAdc = readADC(); }</pre> 	<pre>#include &lt;avr/io.h&gt;  unsigned char readADC() {     return ADCH; };  int main(void) {     unsigned char mAdc = readADC(); }</pre> 
AVR Memory Usage	Program: 92 bytes (1.1% full)	Program: 90 bytes (1.1% full)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

# Глобальные и локальные переменные

---

В большинстве случаев не рекомендуется использовать глобальные переменные. Применяйте локальные переменные везде, где возможно.

- Если переменная используется только в функции, ее следует объявлять внутри функции как локальную переменную.
- Если переменная объявлена как глобальная, в оперативной памяти для нее выделяется уникальный адрес. Также для доступа к глобальной переменной, как правило, используются дополнительные байты (по 2 на 16-и разрядный адрес), чтобы получить ее адрес.
- Локальные переменные обычно размещаются в регистрах или в стеке. Когда вызывается функция, локальные переменные задействуются. Когда функция завершает свою работу, локальные переменные могут быть удалены.

# Пример реализации на микроконтроллерах AVR

	Global variables	Local variables
C source code	<pre>#include &lt;avr/io.h&gt;  uint8_t global_1;  int main(void) {     global_1 = 0xAA;     PORTB = global_1; }</pre>	<pre>#include &lt;avr/io.h&gt;  int main(void) {     uint8_t local_1;      local_1 = 0xAA;     PORTB = local_1; }</pre>
AVR Memory Usage	Program: 104 bytes (1.3% full) (.text + .data + .bootloader) Data: 1 byte (0.1% full) (.data + .bss + .noinit)	Program: 84 bytes (1.0% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

# Переменные

---

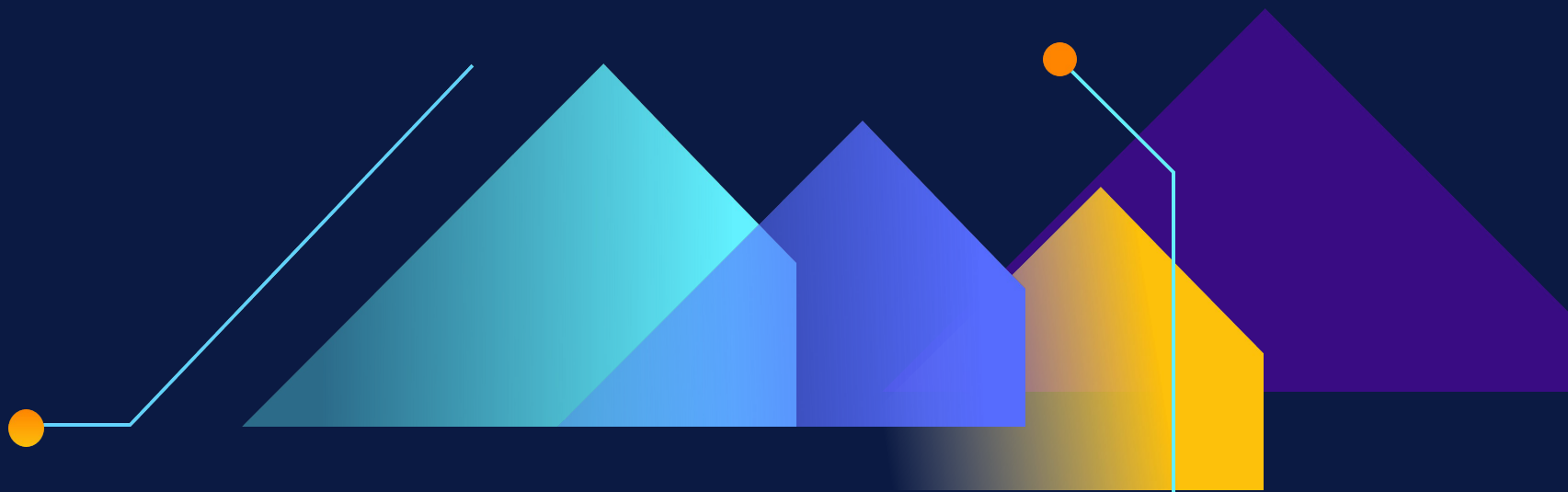
Избегайте ссылок на статические и глобальные переменные внутри циклов. Не используйте `volatile` квалификаторов (переменная может меняться извне), если вы точно не уверены в этом.

Компилятор предполагает, что в любом месте программы к переменной `volatile` может обратиться неизвестный процесс, который использует или изменяет её значение.

Независимо от оптимизаций, указанных в командной строке, нужно создать код для каждого назначения переменной `volatile` или ссылки на неё, даже если кажется, что он ничего не делает.

По возможности избегайте передачи адреса переменной в функцию: компилятор считает, что такие переменные могут быть изменены в любой момент, и не оптимизирует код в этом месте.

# Работа со строкама



# Работа со строками

При работе с функциями из библиотеки `string.h` следует придерживаться следующих рекомендаций. Избегайте вызова `strlen()` во время цикла, включающего саму строку.

```
// Плохо. Вызов strlen в
цикле
char s[]="Hello world";
for(size_t i=0; i<strlen(s);
i++)
    putchar(s[i]);
```

```
// Хорошо. Вызов strlen 1 раз
char s[]="Hello world";
size_t len = strlen(s);
for(size_t i=0; i<len; i++)
    putchar(s[i]);
```

# Работа со строками

Вы можете сэкономить немного времени, проверив первые символы сравниваемых строк перед вызовом. Если первые символы отличаются, нет причин вызывать `strcmp` для проверки остальных. Это сработает из-за неравномерного распределения букв в естественных языках, и выигрыш составит не 26 к 1, а скорее 15 к 1 для данных в верхнем регистре.

```
#define QUICKIE_STRCMP(a, b)  (*(a) != *(b) ? \  
    (int) ((unsigned char) *(a) - \  
           (unsigned char) *(b)) : \  
    strcmp((a), (b)))
```

# Работа со строками

Не используйте функцию `strlen` для определения пустой строки. В данном случае вызов `strlen` на большой строке просканирует всю строку до символа конец строки — `'\0'`.

```
strlen(s) == 0
```

```
*s == '\0' ;
```

Аналогичная ситуация и с функцией `strcpy`.

```
strcpy(s, "");
```

```
*s == '\0' ;
```



# Работа со строками

При использовании функции `strncpy`, необходимо учитывать, что строка приёмник заполняется избыточными нулями. Если строка приёмник имеет большой размер, то это может привести к незначительной задержке.

```
char s[]="hello world";  
printf("s[9]=%c\n",s[9]);  
strncpy(s,"erase", 10);  
printf("%s\n", s);  
printf("s[9]=%c\n",s[9]);
```

```
s[9]=  
erase  
s[9]=
```

# Работа со строками

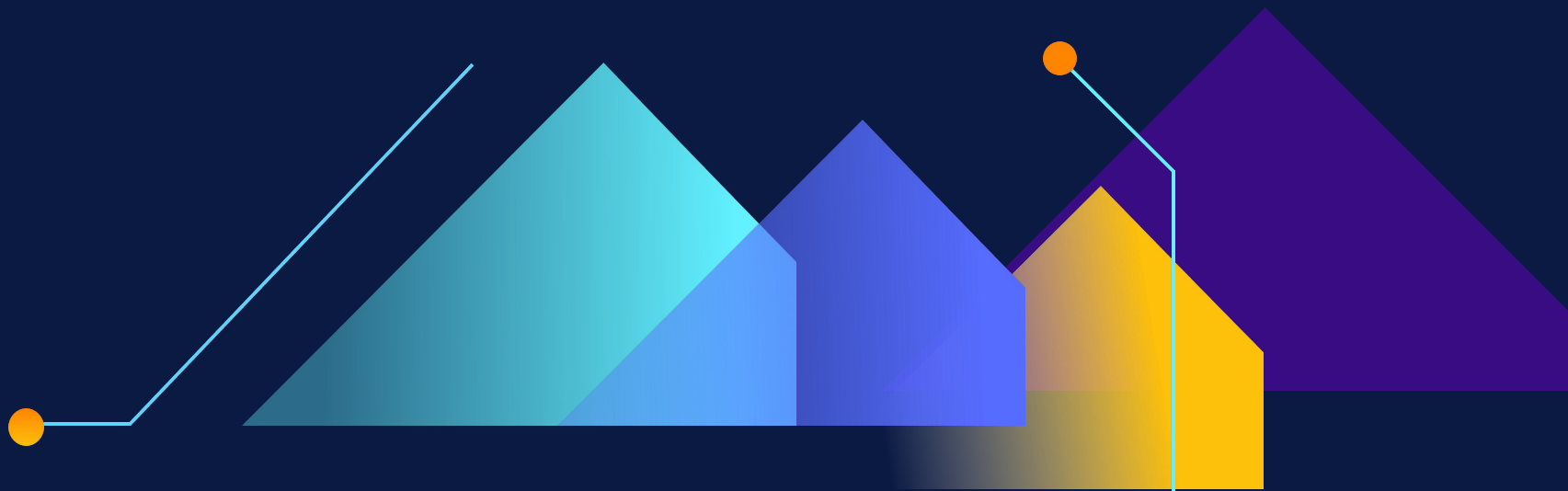
Обычно `memcpy` работает быстрее, чем `memmove`, потому предполагается, что его аргументы не перекрываются друг другом.

```
int s[10000]={1,2,3};
int d[10000]={0};
timespec_get(&tstart, TIME_UTC);
memmove(d,s,10000);
timespec_get(&tstop, TIME_UTC);
printf("%ld nanoseconds\n",
        tstop.tv_nsec - tstart.tv_nsec);
timespec_get(&tstart, TIME_UTC);
memcpy(d,s,10000);
timespec_get(&tstop, TIME_UTC);
printf(" %ld nanoseconds\n",
        tstop.tv_nsec - tstart.tv_nsec);
```

8000 nanoseconds

7000 nanoseconds

# Использование ассемблера



# Использование ассемблера

Написание ассемблерного кода достаточно сложно.

Компилятор создает достаточно эффективный код, особенно если у разработчика мало опыта в написании кода на ассемблере.

Два подхода:

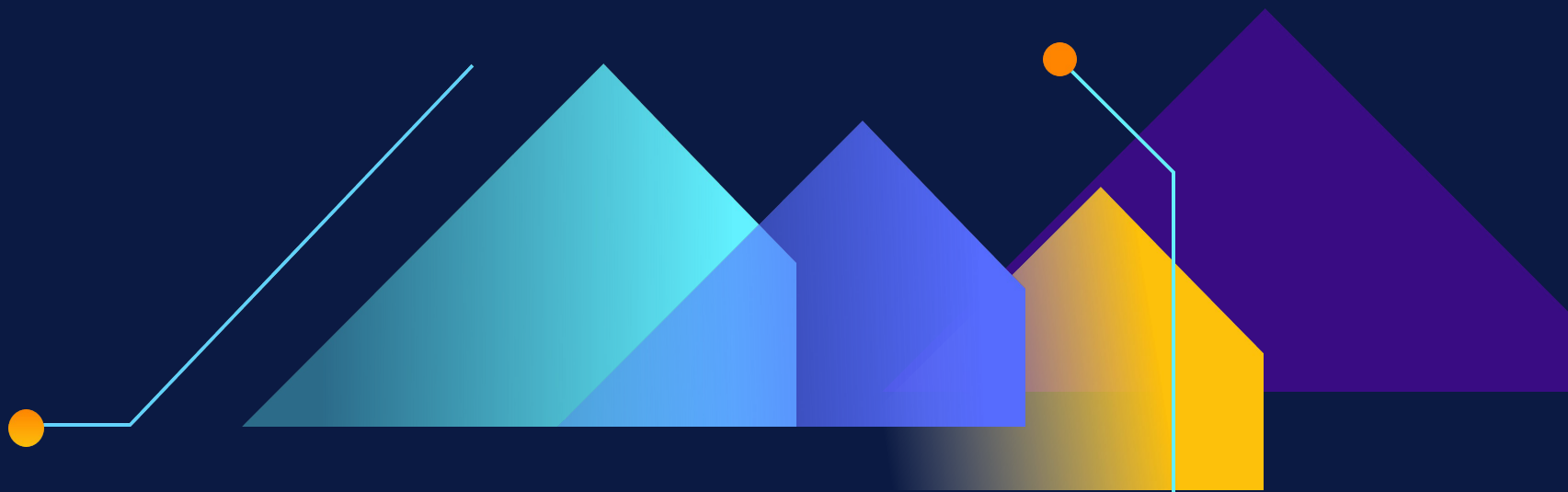
- написать функции на ассемблере с нуля
- взять версию компилятора в качестве отправной точки и просто настроить его

# Пример реализации на микроконтроллерах AVR

	Function	Assembly macro
Функция на C	<pre>#include &lt;avr/io.h&gt;  void enable_usart_rx(void) {     UCSR0B  = 0x80; };  int main(void) {     enable_usart_rx();     while (1){     } }</pre>	<pre>#include &lt;avr/io.h&gt;  #define enable_usart_rx() \     __asm__ __volatile__ ( \         "lds    r24,0x00C1" "\n\t" \         "ori    r24, 0x80" "\n\t" \         "sts    0x00C1, r24" \         ::)  int main(void) {     enable_usart_rx();     while (1){     } }</pre>
C source code		
AVR Memory Usage	Program: 90 bytes (1.1% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)	Program: 86 bytes (1.0% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

Ассемблер

# Оптимизация памяти



# Многомерные массивы

При обработке многомерных массивов обязательно сначала увеличивайте крайний правый индекс.

```
float array[20][100];  
int i, j;  
//Плохо  
for (j = 0; j < 100; j++)  
    for (i = 0; i < 20; i++)  
        array[i][j] = 0.0;
```

```
float array[20][100];  
int i, j;  
//Хорошо  
for (i = 0; i < 20; i++)  
    for (j = 0; j < 100;  
j++)  
        array[i][j] = 0.0;
```

# Копирование больших объектов

---

Избегайте копирования больших объектов: массивы, строки или структуры. При передаче структур в функции лучше передать по ссылке.

- Если в программе происходит интенсивное обращение к элементам в «параллельных» массивах, то лучше объединить их в массив структур, чтобы данные для данного индекса хранились в памяти вместе.

```
struct myStruct ar[1000];
```

- Если в массиве структур происходит обращение только к небольшому количеству полей в каждой структуре, то лучше разделить эти поля на отдельный массив, чтобы неиспользуемые поля не считывались в кэш без надобности.

```
struct myStruct ar[1000], ar2[1000];
```



# Выравнивание полей в структуре

Расположите поля в структуре от большего к меньшему, это позволит сэкономить место, которое она занимает.

```
// sizeof 16
```

```
struct st1{  
    int i;  
    char c;  
    int u;  
    char b;  
};
```

```
// sizeof 12
```

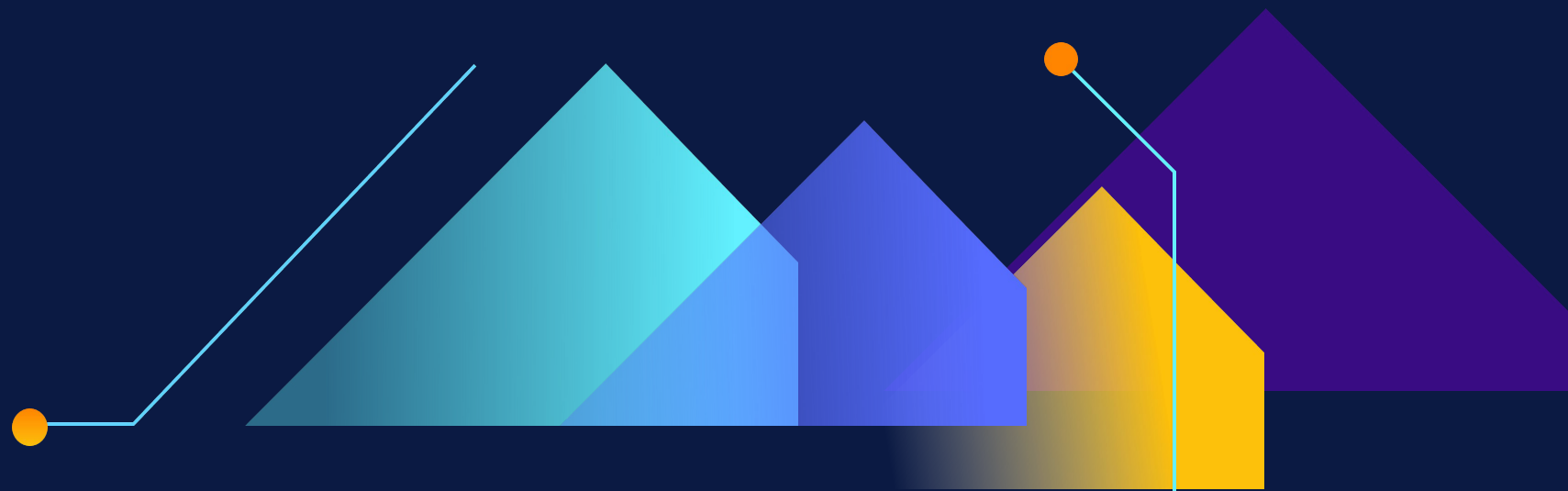
```
struct st2{  
    int i;  
    int u;  
    char c;  
    char b;  
};
```

# Выравнивание полей в структуре

Обычно для того, чтобы хранить флаг или бит режима, используют переменные типа `char` или `uint8_t`. Несколько из этих флагов можно объединить в один байт за счёт переносимости данных, используя битовые поля. Также можно упаковать данные в обычную переменную и использовать битовые маски и оператор `&`.

```
// sizeof 1
struct flags{
    uint8_t a : 2;
    uint8_t b : 2;
    uint8_t c : 2;
    uint8_t d : 2;
};
```

# Другая оптимизация



# Снижение вычислительной нагрузки

Многие компиляторы автоматически снижают вычислительную нагрузку путём замены на более легковесные выражения:

```
x = x / 8;  
y = pow(x, 2);  
z = y * 33;
```

```
x = x >> 3;  
y = x * x;  
z = (y<<5) + y;
```

# If else оптимизация

---

- При использовании множественных **if**, сначала поместите тесты для ситуаций, которые возникают чаще всего.
- Часто это принимает форму длинной очереди взаимоисключающих **if-then-else**, из которых выполняется только одно.
- Если на первое место поставить наиболее вероятный вариант, то в долгосрочной перспективе потребуется выполнять меньше «**if**».
- Но если условия просты, например `if (x==3)`, рассмотрите возможность использования оператора `switch(x){case 3:}`.
- Некоторые компиляторы довольно хорошо умеют оптимизировать операторы **switch**.

# Кеширующий массив в рекурсии

Иногда стоит использовать кеширующий массив вместо рекурсивного вызова.

```
int factorial(int i) {  
    if (i <= 0)  
        return 1;  
    else  
        return i * factorial(i - 1);  
}
```

```
static int factorial_table[] =  
    {1, 1, 2, 6, 24, 120, 720 /* и  
так далее */};  
  
int factorial(int i)  
{  
    return factorial_table[i];  
}
```

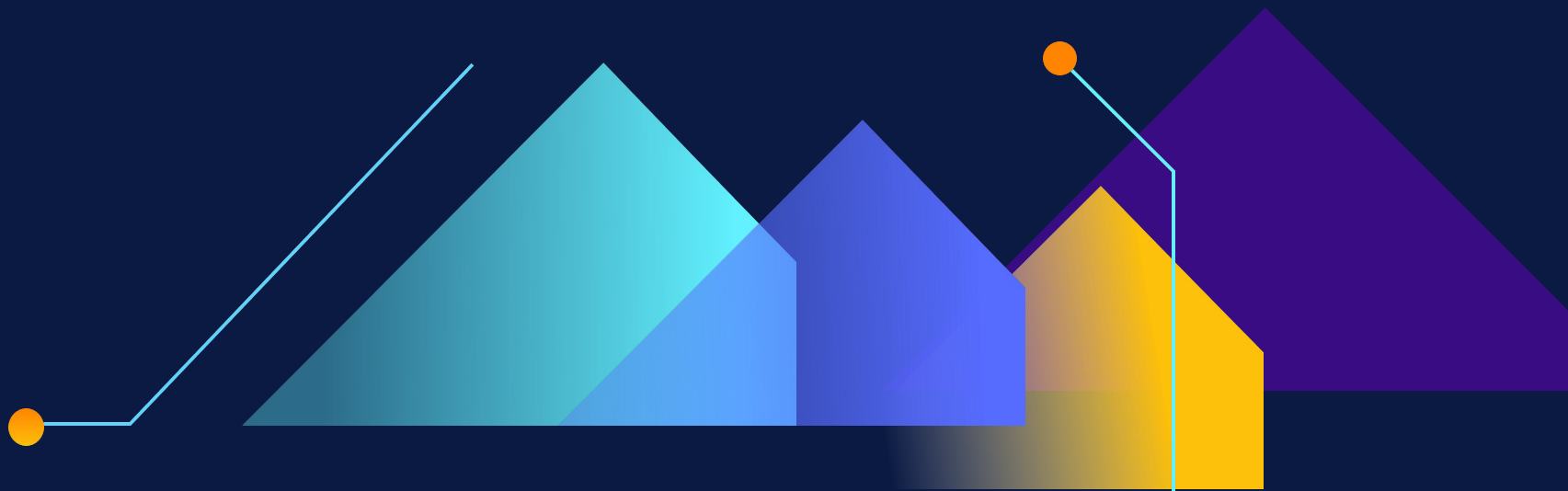
# Оптимизация рекурсии

Общее правило: существует большое количество рекурсивных алгоритмов, которые имеют простые итерационные аналоги — используйте их.

**Внимание!** Расчёты, которые занимают постоянное время, часто можно пересчитать быстрее, чем их можно извлечь из памяти, и поэтому поиск в таблице не всегда приносит пользу.

Если массив слишком велик, можно создать некоторый код инициализации и вычислить все значения при запуске. Можно также указать в таблице первые  $N$  случаев, а затем дополнять её функцией для вычисления остальных. Пока данные будут браться из массива — будет выигрыш.

# Советы новичкам





# Советы новичкам

Некоторые вещи, которые могут сбить с толку новичков:

1. Программисты склонны переоценивать полезность написанных ими программ. Примерная стоимость оптимизации:

$$\begin{aligned} &<\text{количество запусков}> \times <\text{количество пользователей}> \times \\ &<\text{экономия времени}> \times <\text{зарплата пользователя}> \quad -- \quad <\text{время,} \\ &\text{потраченное на оптимизацию}> \times <\text{зарплата программиста}> \end{aligned}$$

Даже если программа будет запускаться тысячами пользователей сотни раз, дополнительный день, потраченный на экономию 40 миллисекунд, вероятно, не поможет.

# Советы новичкам

**2.** Все компьютеры/устройства разные. То, что быстро на одной машине, может быть медленным на другой. Дополнительные интерфейсные карты, разные диски, количество зарегистрированных пользователей, дополнительная память, фоновые демоны и всё остальное могут повлиять на скорость различных частей программы, на то, какая часть программы является узким местом, и на её скорость в целом.

Проблема заключается в том, что многие программисты являются опытными пользователями и имеют компьютеры, загруженные памятью, математическим сопроцессором и тоннами дискового пространства, в то время как пользователи получают простые компьютеры и работают по сети. Программист получает искажённое представление о производительности программы и может не оптимизировать те части программы, которые отнимают время пользователя, например подпрограммы с плавающей запятой или интенсивные операции с памятью.

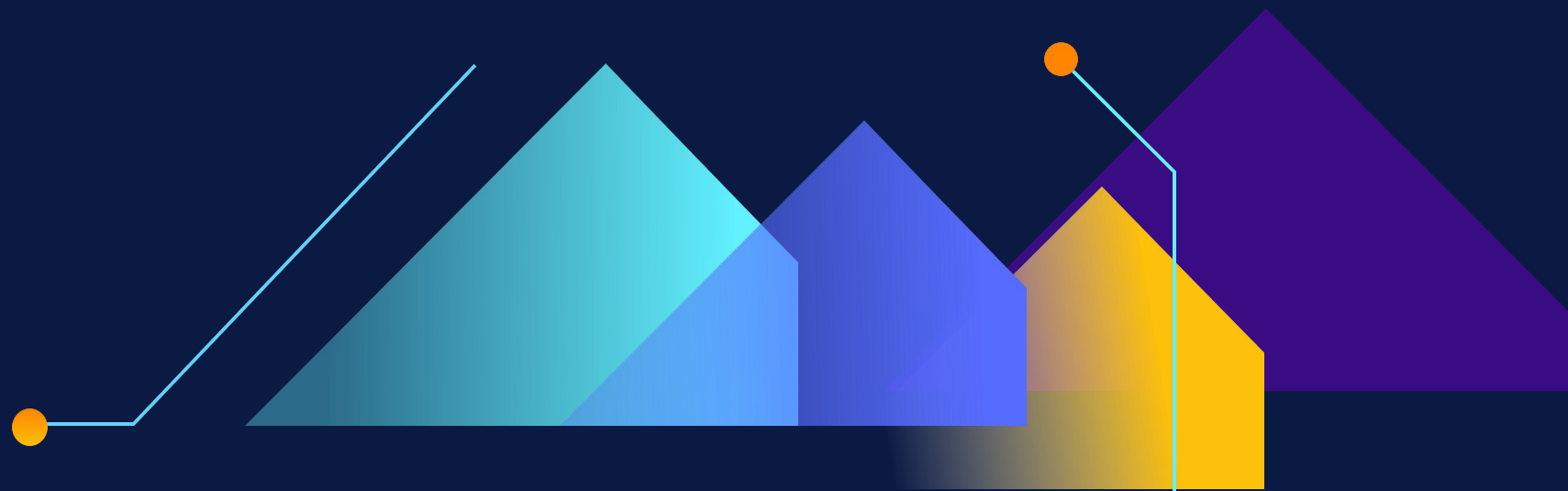
# Советы новичкам

3. Многие из этих оптимизаций могут быть уже выполнены компилятором!
4. Не берите в привычку писать код в соответствии с приведёнными выше правилами оптимизации. Применяйте их только после того, как вы точно определите, какая функция является узким местом. Некоторые правила, если их применить глобально, сделают программу еще медленнее. Практически все они делают назначение кода менее очевидным для читателя-человека, что может доставить неприятности, если вам или кому-то ещё понадобится исправить ошибку или что-то ещё в дальнейшем. **Внимание!** Обязательно прокомментируйте оптимизацию — следующий программист может просто предположить, что это уродливый код, и переписать его.

# Советы новичкам

5. Неделя, потраченная на оптимизацию программы, легко может стоить тысячи долларов рабочего времени программиста. Иногда проще просто купить более быстрый процессор, больше памяти или более быстрый диск и таким образом решить проблему.
6. Новички часто предполагают, что написание большого количества операторов в одной строке и удаление пробелов и табуляции ускорит процесс. Хотя это может быть допустимым методом для некоторых интерпретируемых языков, в С такой метод не работает. 😊

# Профилирование gprof и gcc



# Профилирование

**Профилирование** — сбор характеристик работы программы, таких как время выполнения отдельных функций, число верно предсказанных условных переходов, число кэш-промахов и т. д. Используя данный инструмент можно определить те части кода, которые занимают много времени.

# Профилирование



Используя данный инструмент можно определить те части кода, которые занимают много времени.

В больших проектах профилирование поможет сохранить время несколькими способами:

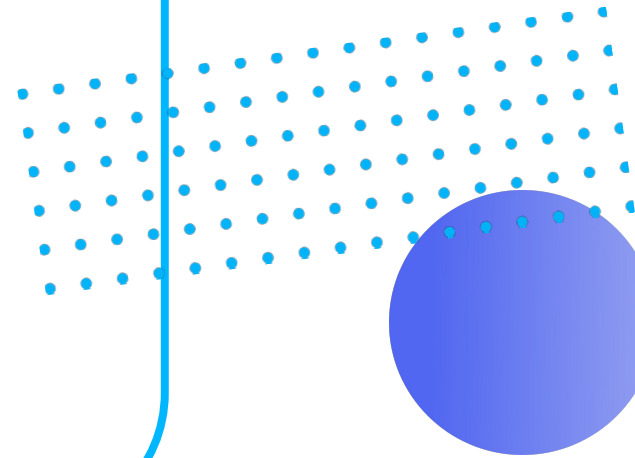
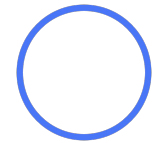
- определять части программы, которые выполняются медленнее, чем ожидалось;
- найти множество других статистических данных, с помощью которых можно обнаружить и предсказать множество потенциальных ошибок



# Используем gprof

Для того чтобы воспользоваться профайлером, необходимо выполнить три простых шага:

1. Включите профилирование при компиляции кода
2. Запустите программу для сбора профилированных данных
3. Запустите gprof для обработки собранных данных и отображения их в читаемой форме



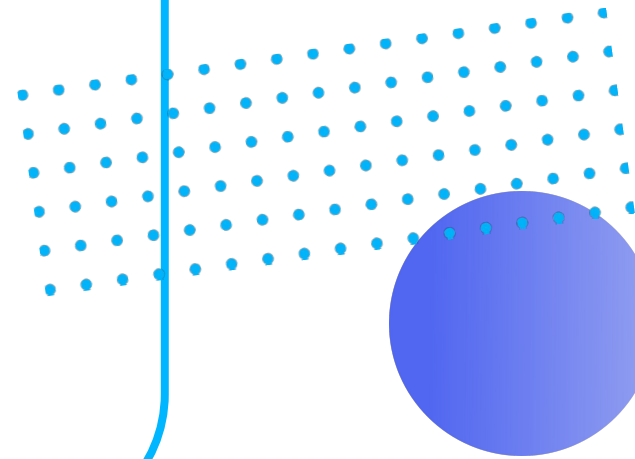
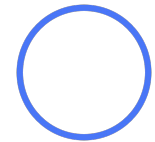




# Используем gprof

Созданный gprof содержит несколько таблиц с дополнительной информацией:

- Затраты времени на выполнение конкретной функции, сколько раз она вызывалась и т. д.
- График вызовов функций, например из какой функции функция была вызвана, все функции были вызваны из этой конкретной функции и т. д. Таким образом, можно также получить представление о времени выполнения, затраченном на отдельные функции.



# Пример из двух файлов

- **main.c** — основной файл содержит три функции(func1, func2, func3) и функцию main
- **function.c** — дополнительный файл содержит одну функцию (new\_func1)

Циклы внутри функций работают продолжительное время.

```
//main.c
#include<stdio.h>
void new_func1(void) ;
void func1(void) {
    printf("\n Inside func1 \n");
    int i = 0;
    for(;i<0xffffffff;i++);
    new_func1();
    return;
}
```

```
static void func2(void)
{
    printf("\n Inside func2
\n");
    int i = 0;

    for(;i<0xffffffffaa;i++);
    return;
}
```

# Пример из двух файлов

```
static int func3(void)
{
    printf("\n Inside func3
\n");
    int i = 0;
    int arr[10000] = {0};

    for(;i<0xffffffff00;i++)
        arr[i%10000] = i;
    return arr[0];
}
```

```
int main(void)
{
    printf("\n Inside
main() \n");
    int i = 0;

    for(;i<0xffffffff;i++);
    func1();
    func2();
    printf("func3 =
%d\n",func3());
    return 0;
}
```

# Пример из двух файлов

```
//function.c
#include<stdio.h>

void new_func1(void)
{
    printf("\n Inside new_func1() \n");
    int i = 0;

    for(;i<0xffffffff;i++);

    return;
}
```

# Шаг 1. Включаем профилирование при КОМПИЛЯЦИИ

Ключ `-pg` — генерирует дополнительный код, который соберёт профилирующую информацию, впоследствии обработанную анализатором `gprof`.

```
gcc -Wall -pg main.c function.c -o prog
```

под ОС Windows

```
gcc -Wall -pg -no-pie main.c function.c -o prog
```

## Шаг 2. Запускаем программу

После запуска программы сгенерируется дополнительный файл - gmon.out, который будет содержать профилирующую информацию.

```
$ ./prog
  Inside main()
  Inside func1
  Inside new_func1()
  Inside func2
  Inside func3
func3 = -10000
$ ls
function.c  gmon.out  main.c  prog
```

## Шаг 3. Запускаем gprof

Первый аргумент — это бинарный файл программы, второй аргумент — это файл отчёта, который был сформирован после запуска. Перенаправляем вывод в файл report.txt. Данный файл содержит хорошо описанную информацию состоящую из двух частей:

- Flat profile — информация по затраченному времени
- Graph call — граф вызова функций

```
gprof prog gmon.out > report.txt
```

под ОС Windows

```
gprof prog.exe gmon.out > report.txt
```

# Различные ключи gprof

gprof генерит много подробной информации, вывод которой можно ограничить, используя различные ключи:

- -a убрать из вывод static функции
- -b выдать отчёт в укороченном виде
- -p вывести только Flat profile

```
$ gprof -p -b prog gmon.out > report.txt
```

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
30.29	10.21	10.21	1	10.21	10.21	func3
24.87	18.60	8.39	1	8.39	15.88	func1
22.94	26.34	7.73	1	7.73	7.73	func2
22.22	33.83	7.49	1	7.49	7.49	new_func1
0.12	33.87	0.04				main



# Различные ключи gprof

- -P не выводить в отчёт Flat profile
- -p<имя функции> вывести информацию о какой-то функции

```
$ gprof -pfunc3 -b prog gmon.out > report.txt
```

```
Each sample counts as 0.01 seconds.
```

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
101.48	10.21	10.21	1	10.21	10.21	func3

# Различные ключи gprof

- -q напечатать только информацию из секции Call graph
- -q<имя функции> напечатать только информацию из секции Call graph для указанной функции

```
$ gprof -qfunc3 -b prog gmon.out > report.txt
Call graph
```

```
granularity: each sample hit covers 2 byte(s) for 0.03% of 33.87 seconds
```

index	% time	self	children	called	name
		10.21	0.00	1/1	main (1)
[3]	30.2	10.21	0.00	1	func3 [3]

-----

^L

Index by function name

(2) func1

(4) func2

[3] func3

(1) main

(5) new\_func1

# Различные ключи gprof

- -Q не выводить Call graph

Ключи -Q и -P можно совмещать с именем функции для подавления вывод информации для данной функции.

# Другие профайлеры

- Valgrind — профайлер для анализа памяти, поиска утечек памяти и не только
- gperftools от Google — набор инструментов, предназначенных для анализа и повышения производительности многопоточных приложений

Стоит отметить, что valgrind в основном ориентирован на анализ памяти, выделенной в куче (Heap). Рассмотрим небольшой пример, который использует входящую в эту библиотеку утилиту massif.

# Другие профайлеры

```
// main_array.c
#include <stdlib.h>

void g(void)
{
    malloc(4000);
}

void f(void)
{
    malloc(2000);
    g();
}
```

```
int main(void) {
    int i;
    int* a[10];
    for (i = 0; i < 10; i++) {
        a[i] = malloc(1000);
    }
    f();
    g();
    for (i = 0; i < 10; i++) {
        free(a[i]);
    }
    return 0;
}
```

# valgrind

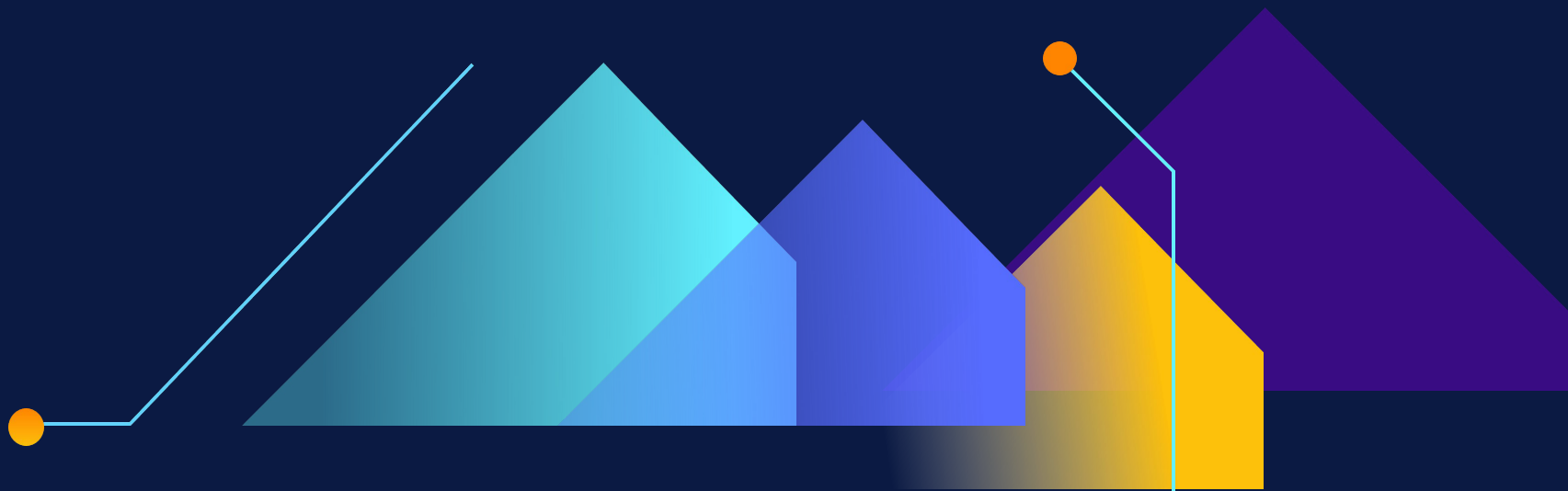
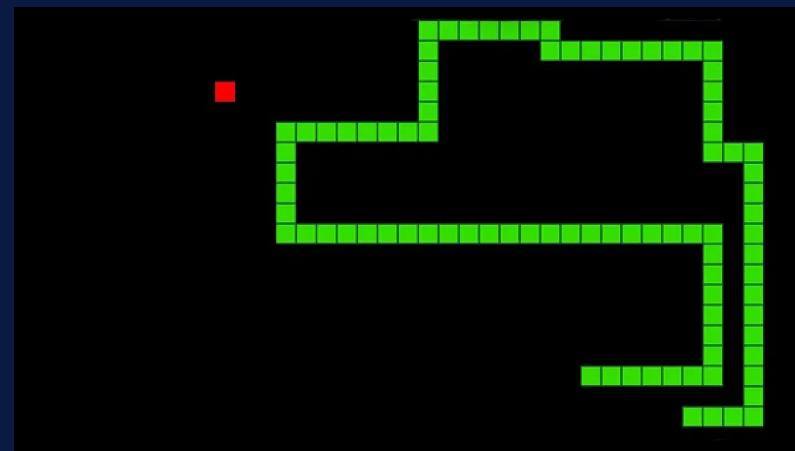
```
$ gcc -g -o prog_array main_array.c
$ valgrind --tool=massif ./prog_array
$ ms_print massif.out.10787
```

```
...
99.76% (10,000B) (heap allocation functions) malloc/new/new[],
--alloc-fns, etc.
->79.81% (8,000B) 0x1086E1: g (main_array.c:5)
| ->39.90% (4,000B) 0x1086F7: f (main_array.c:11)
| | ->39.90% (4,000B) 0x108740: main (main_array.c:23)
| |
| ->39.90% (4,000B) 0x108745: main (main_array.c:25)
|
->19.95% (2,000B) 0x1086F2: f (main_array.c:10)
| ->19.95% (2,000B) 0x108740: main (main_array.c:23)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

# Поиск утечек памяти

```
$      valgrind      --leak-check=full      -v      ./prog_array
==397542== Memcheck, a memory error detector
==397542== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward
et al.
==397542== Using Valgrind-3.15.0-608cb11914-20190413 and LibVEX;
rerun with -h for copyright info
...
==397542== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0
from 0)
```

# Игра змейка





# Логика игры

---

**Хвост** движется путём сдвига массива хвоста вправо на один элемент при каждом шаге, координаты самого первого элемента хвоста копируются из координат головы. Для перемещения хвоста реализована функция `goTail(struct snake *head)`. Для увеличения размера хвоста достаточно прибавить 1 к `snake.tsize`. За это отвечает функция `addTail(struct snake *head)`.

**Еда** — это массив точек, состоящий из координат `x, y`, времени, когда данная точка была установлена, и поля, сигнализирующего, была ли данная точка съедена. Точки расставляются случайным образом в самом начале программы — `putFood(food, SEED_NUMBER)`, `putFoodSeed(struct food *fp)`.

# Логика игры

---

**Обновление еды.** Если через какое-то время(`FOOD_EXPIRE_SECONDS`) точка устаревает, или же она была съедена(`food[i].enable==0`), то происходит её повторная отрисовка и обновление времени — `refreshFood(food, SEED_NUMBER)`

**Съесть зерно.** Такое событие возникает, когда координаты головы совпадают с координатой зерна. В этом случае зерно помечается как `enable=0`. Проверка того, является ли какое-то из зерен съеденным, происходит при помощи функции логической функции `haveEat(struct snake *head, struct food f[])`: в этом случае происходит увеличение хвоста на 1 элемент. (В следующей лекции.)

**Увеличение хвоста** — отвечает функция `addTail(&snake)`.

# Логика игры

---

**Циклическое движение змейки** по экрану терминала. Для обеспечения данной возможности необходимо сравнить координаты головы и максимально возможное значение координаты в текущем терминале. Для вычисления размера терминального окна используется макрос библиотеки `ncurses` `getmaxyx(stdscr, max_y, max_x)`. В случае, когда координата превышает максимальное значение, происходит её обнуление. Если координата достигает отрицательного значения, то ей присваивается соответствующее максимальное значение `max_y`, `max_x`. Полный текст программы можно посмотреть [тут](#).

# Задание

---

1. Добавить возможность управления змейкой клавишами WSAD (вне зависимости от регистра).

Зависимости от регистра в соответствии с таблицей.

W, w	Вверх
S, s	Вниз
A, a	Влево
D, d	Вправо



**Дедлайн:** конец курса

Советуем регулярно выполнять ДЗ  
(наверстать пропуски тяжело)

# Задание

---

Для решения предлагается сделать массив кодов управления **struct control\_buttons default\_controls[CONTROLS]**. **CONTROLS** – определяем количество элементов массива.

В необходимых функциях в цикле необходимо сравнивать с каждым типом управления в цикле

```
for (int i = 0; i < CONTROLS; i++)
```



**Дедлайн:** конец курса

Советуем регулярно выполнять ДЗ  
(наверстать пропуски тяжело)

# Задание

---

2. Написать функцию, которая будет проверять корректность выбранного направления. Змейка не может наступать на хвост, поэтому необходимо запретить

- перемещение справа-налево (при движении RIGHT нажатие стрелки влево),
- перемещение сверху-вниз (при движении UP нажатие стрелки вниз),
- перемещение слева-направо (при движении LEFT нажатие стрелки вправо),
- перемещение снизу-вверх (при движении DOWN нажатие стрелки вверх).

Функция должна иметь вид:

```
int checkDirection(snake_t* snake, int32_t key).
```



**Дедлайн:** конец курса

Советуем регулярно выполнять ДЗ  
(наверстать пропуски тяжело)