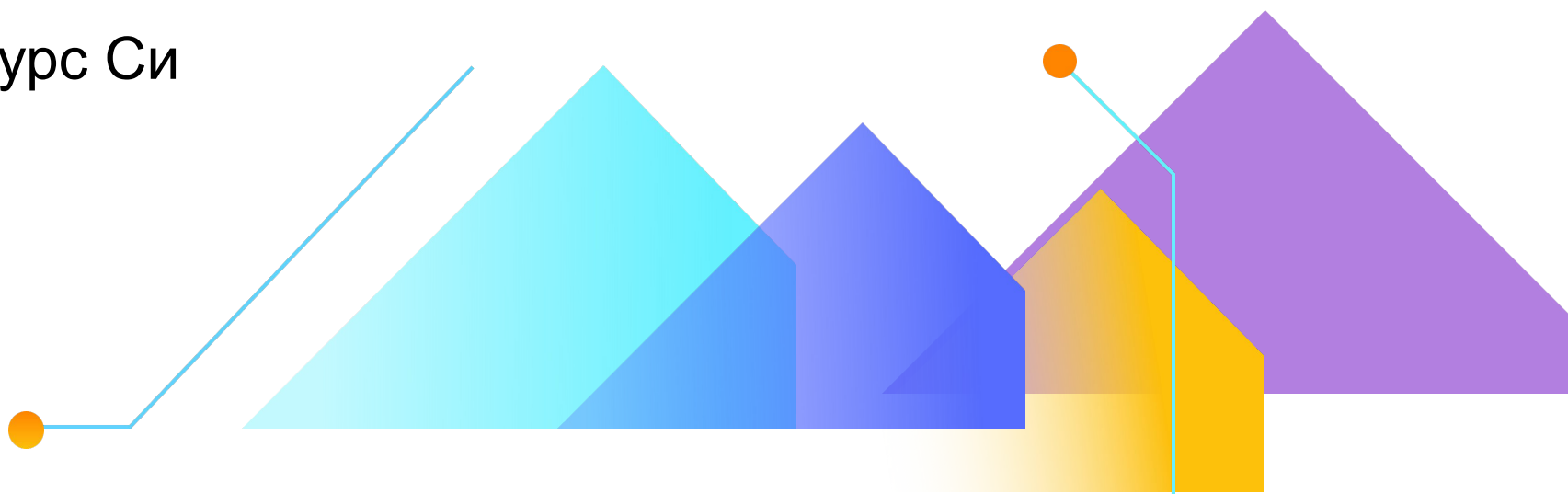




Лекция №6

Компиляция и компиляторы

Продвинутый курс Си



План курса

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

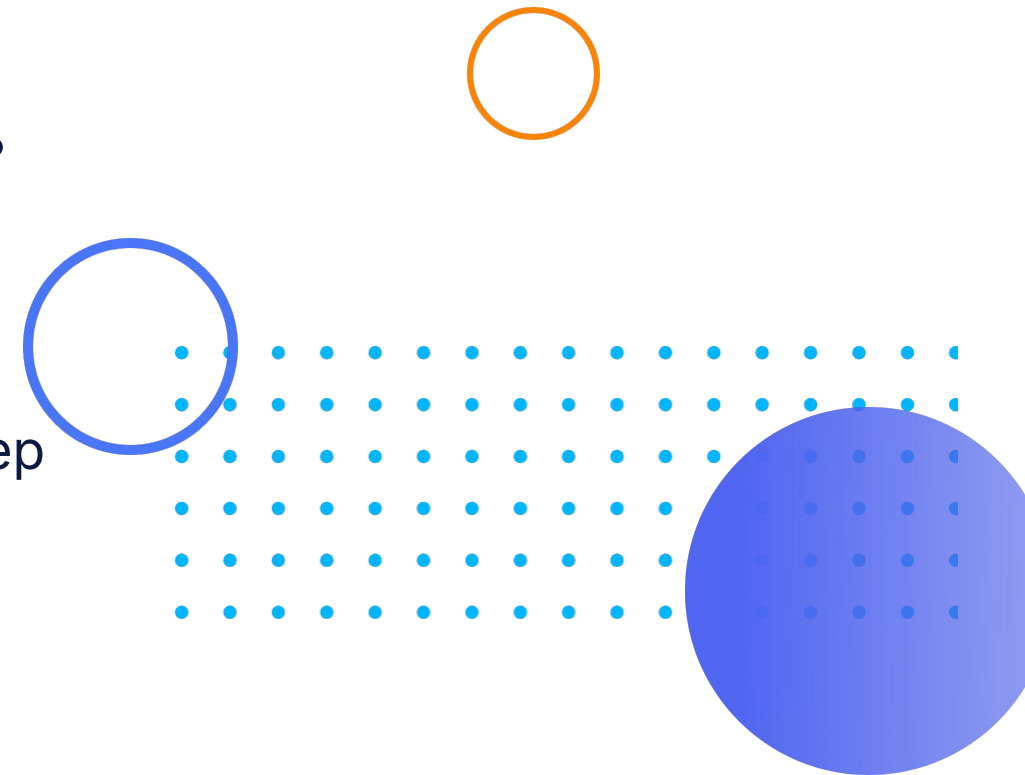


Маршрут

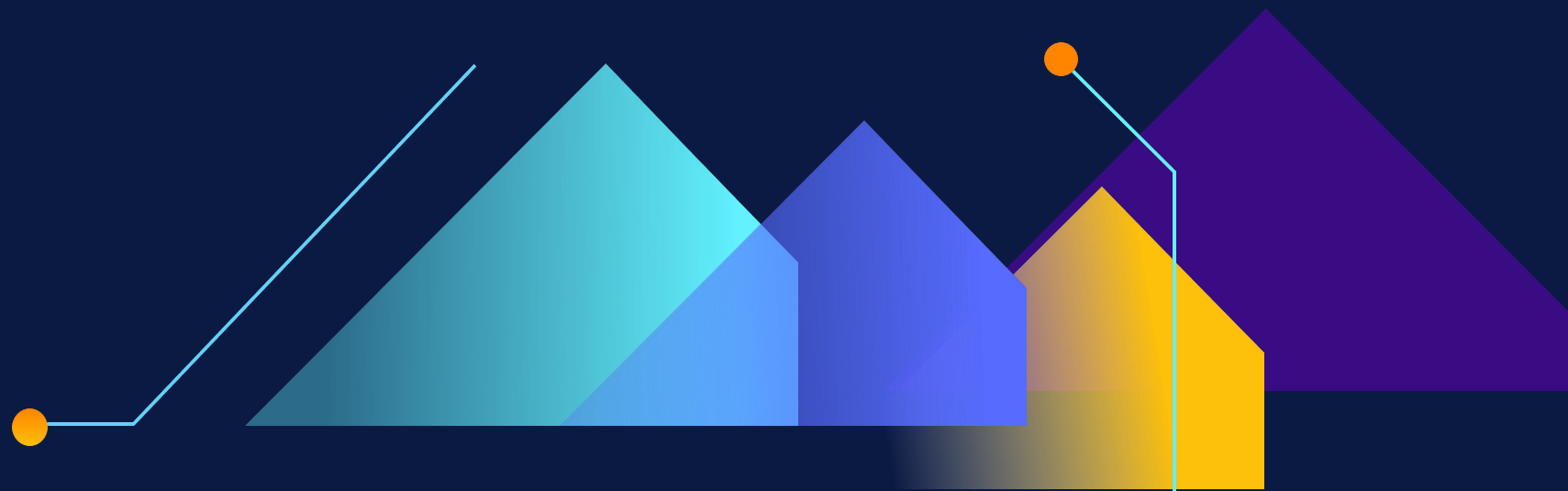
Компиляция и компиляторы

- Повторим, что такое компиляция
- Какие ключи компиляции как использовать
- Изучим, как можно исследовать исполняемый файл
- Разберем некоторые примеры на ассемблер

https://github.com/Sudar1977/MIPI_AdvancedC



Змейка (окончание)



Поедание зерна змейкой

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 `_Bool haveEat(struct snake_t *head, struct food f[])`: В ЭТОМ случае происходит увеличение хвоста на 1 элемент функцией `void addTail(struct snake_t *head)`.

В цикле `for(size_t i=0; i<MAX_FOOD_SIZE; i++)` происходит проверка наличия еды и совпадения координат `head->x == f[i].x` и `head->y == f[i].y`.

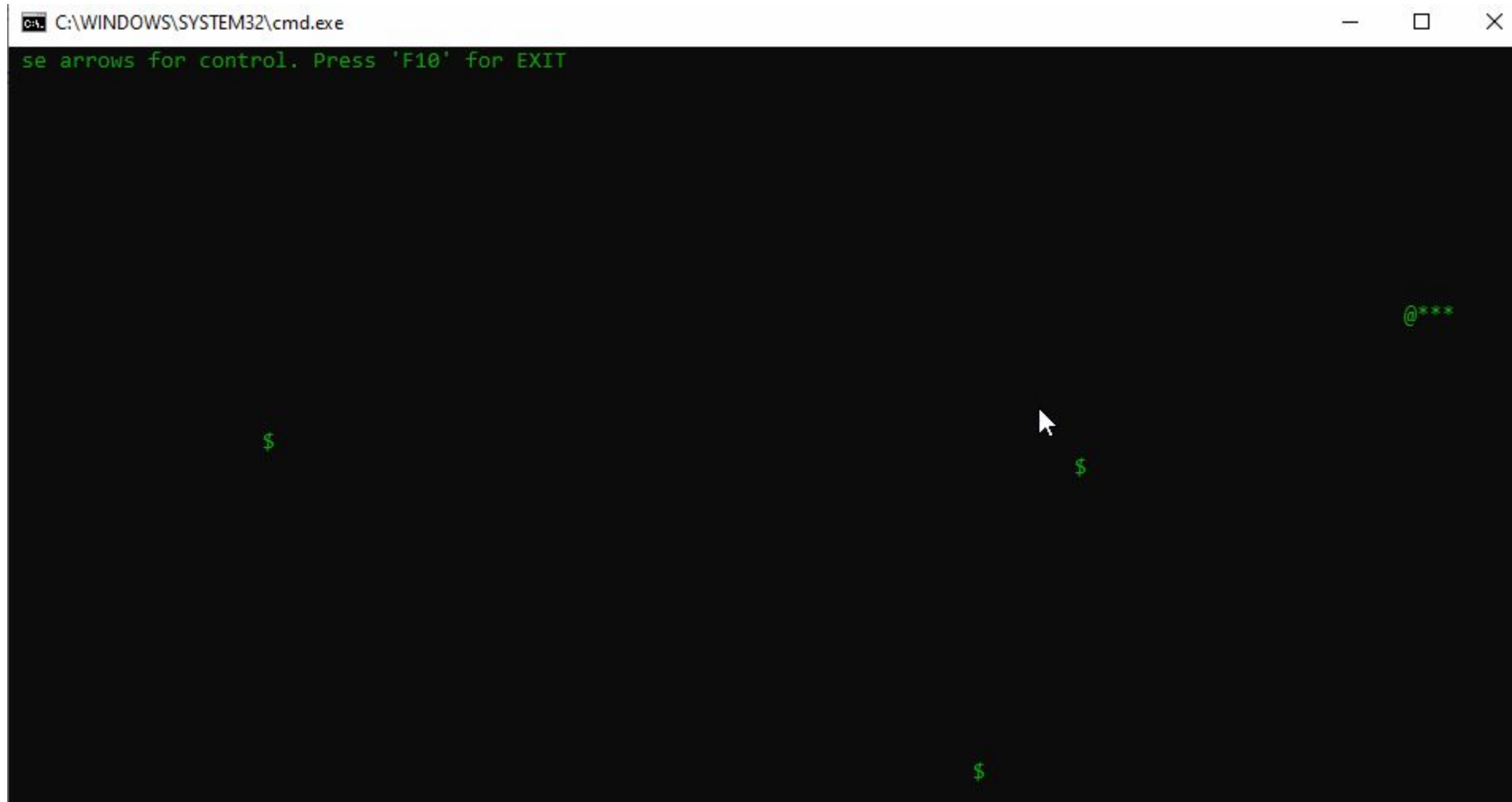
В случае выполнения условий `enable=0` и возвращается единица.

За увеличение хвоста — отвечает функция `addTail(&snake)`.

В структуре `head` параметр длины `tsize` увеличивается на единицу.

```
head->tsize++;
```

Поедание зерна змейкой (демонстрация)



Функцию update

Вынести тело цикла while из int main() в отдельную функцию update и посмотреть, как изменится профилирование.

```
void update(struct snake_t *head, struct food f[], const int32_t
key) {
    clock_t begin = clock();
    go(head);
    goTail(head);
    if (checkDirection(head, key)) {
        changeDirection(head, key);
    }
    refreshFood(food, SEED_NUMBER); // Обновляем еду
    if (haveEat(head, food)) {
        addTail(head);
    }
    while ((double)(clock() - begin)/CLOCKS_PER_SEC < DELAY)
    {}
}
```

Столкновение головы с хвостом

Добавим функцию столкновения головы змейки со своим хвостом.

Событие возникает, когда координаты головы совпадают с координатой хвоста. Проверка того, столкнулась ли голова с хвостом, происходит при помощи функции логической функции `isCrush(snake_t * snake)`.

В этом случае происходит автоматическое окончание игры.

Столкновение головы с хвостом (демонстрация)

se arrows for control. Press 'F10' for EXIT

```
*****  
*  
*  
*  
*  
*  
*  
@**
```

\$

\$

\$



Проверка корректности выставления зерна

Написать функцию, которая будет проверять корректность выставления зерна на поле

```
void repairSeed(struct food f[], size_t nfood, struct snake_t *head).
```

Она должна исключать ситуацию, когда хвост змейки совпадает с зерном, то есть

```
f[j].x == head->tail[i].x и f[j].y == head->tail[i].y
```

 в моменты, когда

```
f[i].enable.
```

А также не допускать размещения двух зерен в одной точке игрового поля:

```
i!=j && f[i].enable && f[j].enable && f[j].x == f[i].x && f[j].y ==  
f[i].y && f[i].enable
```

Добавление второй змейки

Добавим в программу вторую змею.

Вместо одной змеи передаём массив данных. PLAYERS определяем количество игроков (змей) в программе.

```
#define PLAYERS 2

int main()
{
    snake_t* snakes[PLAYERS];
    for (int i = 0; i < PLAYERS; i++)
        initSnake(snakes, START_TAIL_SIZE, 10+i*10, 10+i*10, i);
}
```

Добавление второй змейки (продолжение)

Меняем функцию `void initSnake(snake_t *head[], size_t size, int x, int y, int i)` с учётом массива.

```
void initSnake(snake_t *head[], size_t size, int x, int y, int i)
{
    head[i]          = (snake_t*)malloc(sizeof(snake_t));
    tail_t*  tail    = (tail_t*)
malloc(MAX_TAIL_SIZE*sizeof(tail_t));
    initTail(tail, MAX_TAIL_SIZE);
    initHead(head[i], x, y);
    head[i]->tail      = tail; // прикрепляем к голове хвост
    head[i]->tsize      = size+1;
    head[i]->controls = default_controls;
    //~ head->controls = default_controls[1];
}
```

Добавление второй змейки (окончание)

Добавляем циклы `for (int i = 0; i < PLAYRES; i++)` в тело `main()` программы.

```
while( key_pressed != STOP_GAME ){
    key_pressed = getch(); // Считываем клавишу
    for (int i = 0; i < PLAYERS; i++){
        update(snakes[i], food, key_pressed);
        if(isCrush(snakes[i]))
            break;//!!!!!!
        repairSeed(food, SEED_NUMBER, snakes[i]);
    }
    if (key_pressed == PAUSE_GAME) {
        pause();
    }
}
for (int i = 0; i < PLAYERS; i++){
    printExit(snakes[i]);
    free(snakes[i]->tail);
    free(snakes[i]);
}
```

Добавление второй змейки (демонстрация)



Управление для каждой змейки

Добавить отдельные клавиши управления для каждой змейки.

Для этого необходимо создать отдельные структуры управления для каждого игрока `struct pleer1_controls[CONTROLS]` и `struct pleer2_controls[CONTROLS]`.

В теле функции `main` каждой змейке нужно передать свою структуру в поле `controls`.

Добавление ИИ

Добавление ИИ – вторая змея в качестве соперника.

Для автоизменения направления напишем функцию `void autoChangeDirection(snake_t *snake, struct food food[], int foodSize)`. Она определяет ближайшую к себе еду и движется по направлению к ней.

```
int distance(const snake_t snake, const struct food food) {    // вычисляет количество ходов
до еды
    return (abs(snake.x - food.x) + abs(snake.y - food.y));
}
void autoChangeDirection(snake_t *snake, struct food food[], int foodSize)
{
    int pointer = 0;
    for (int i = 1; i < foodSize; i++) {    // ищем ближайшую еду
        pointer = (distance(*snake, food[i]) < distance(*snake, food[pointer])) ? i :
pointer;
    }
    if ((snake->direction == RIGHT || snake->direction == LEFT) &&
        (snake->y != food[pointer].y)) {    // горизонтальное движение
        snake->direction = (food[pointer].y > snake->y) ? DOWN : UP;
    } else if ((snake->direction == DOWN || snake->direction == UP) &&
        (snake->x != food[pointer].x)) {    // вертикальное движение
        snake->direction = (food[pointer].x > snake->x) ? RIGHT : LEFT;
    }
}
```


Добавление ИИ

Вносим изменения в функцию.

```
void update(snake_t *head, struct food f[], int key) {
    autoChangeDirection(head, f, SEED_NUMBER);
    go(head);
    goTail(head);
    if (checkDirection(head, key)) {
        changeDirection(head, key);
    }
    refreshFood(food, SEED_NUMBER); // Обновляем еду
    if (haveEat(head, food)) {
        addTail(head);
        printLevel(head);
        DELAY -= 0.009;
    }
}
```

Добавление ИИ (демонстрация)



Соревнование

Добавить «режим соревнования».

Преобразовать программу, чтобы одна змейка управлялась функцией

**void autoChangeDirection(struct snake *snake, struct food food[],
int foodSize)**, а вторая – вручную клавишами с клавиатуры.

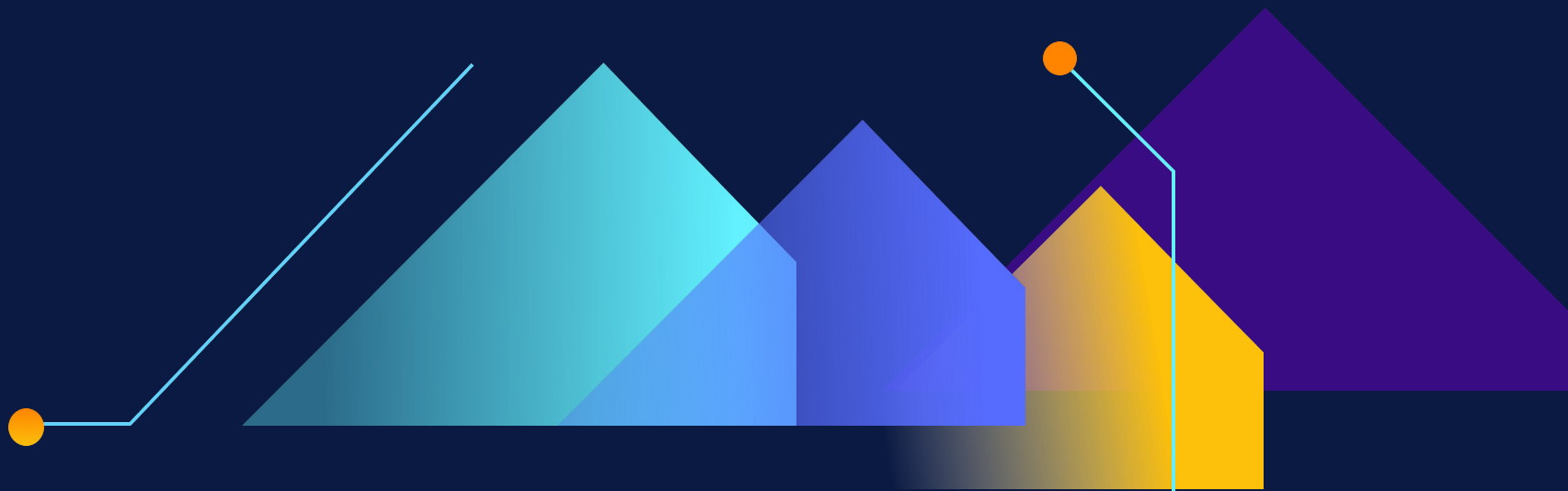
Для этого в функцию

void update(snake_t *head, struct food f[], int key, int ai)
передаём новую переменную int ai.

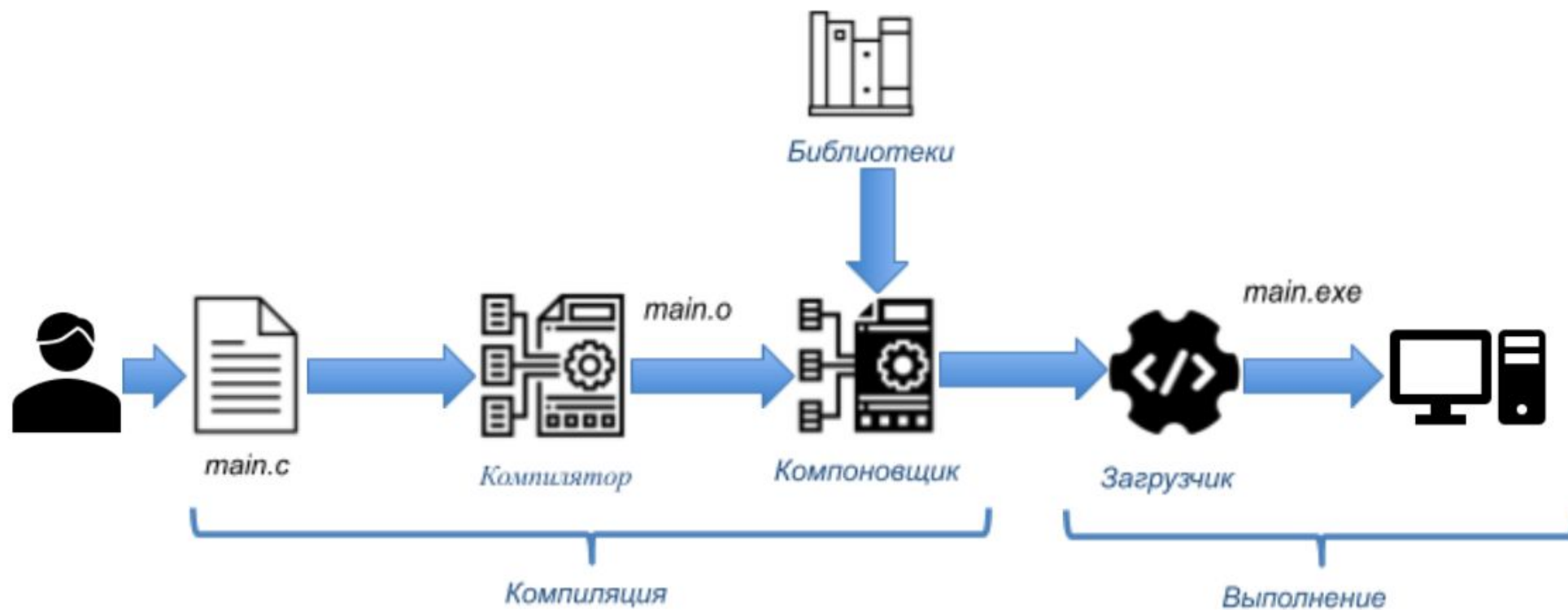
Соревнование (демонстрация)



Компиляция



Компиляция





Компиляция

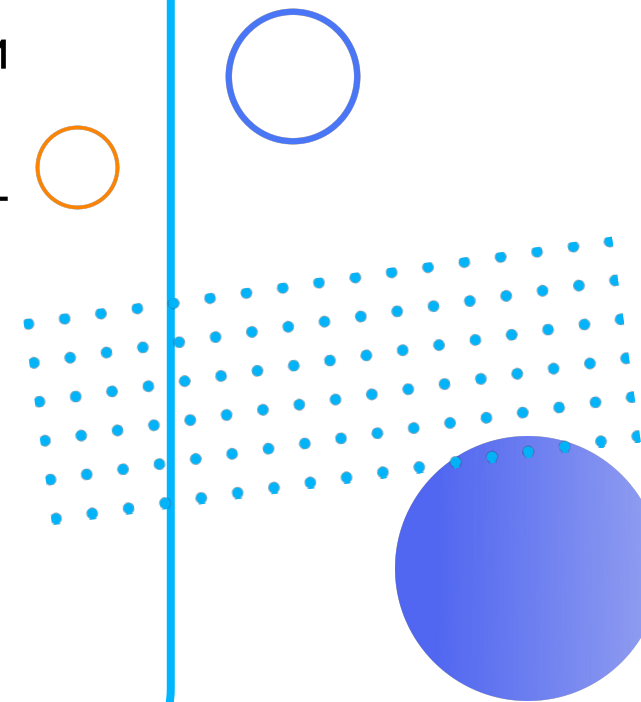
В ходе компиляции одного файла файл обрабатывается:

Препроцессором, производящим набор текстовых подстановок над файлом для получения его окончательного вида и передачи компилятору.

Компилятор получает ассемблерный код, то есть представляет программу в виде последовательности команд центрального процессора.

Ассемблер получает объектный файл (с расширением .o), в котором команды ассемблера закодированы в двоичном виде, а также описан вид статической и глобальной памяти.

Компоновщик обеспечивает слияние нескольких объектных файлов в один **исполняемый файл** программы.



Препроцессирование

Препроцессор сначала читает исходный код и подготавливает его к компиляции в виде трёх этапов.

- Сначала препроцессор удаляет все комментарии из кода, те строки, которые указаны в C как `/ * * /` или `//`.
- Далее препроцессор включает все файлы заголовков `#include «example.h»`.
- Наконец, все макропеременные, определённые в файле, заменяются их заданными значениями.

Результат работы препроцессора можно просмотреть, если запустить gcc с параметром -E.

Препроцессирование

```
$ gcc main.c -E -o main.i
$ tail main.i
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
                           const char * restrict, va_list);
# 408
"/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h"
2 3 4
# 2 "main.c" 2

int main (void) {
    printf("Hello world\n");
    return 0;
}
```

Препроцессирование

```
#include <stdio.h>

int main (void) {
    printf("Hello world\n");
    return 0;
}
```


КОМПИЛЯЦИЯ

```
## %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
subq $16, %rsp
movl $0, -4(%rbp)
leaq L_.str(%rip), %rdi
movb $0, %al
callq _printf
xorl %ecx, %ecx
movl %eax, -8(%rbp)          ## 4-byte Spill
```

КОМПИЛЯЦИЯ

```
movl    %ecx, %eax
addq    $16, %rsp
popq    %rbp
retq
.cfi_endproc

                                ## -- End function

.section __TEXT,__cstring,cstring_literals
L_.str:                          ## @.str
    .asciz "Hello world\n"

.subsections_via_symbols
```

Ассемблирование

После этапа компиляции новый ассемблерный код передаётся ассемблеру. Ассемблер собирает код в объектный код. Ассемблерный код делает соответствие между программой и машинным кодом, объектный код представляет собой машинный код в чистом виде.

Результат работы препроцессора можно просмотреть, если запустить gcc с параметром -o. Результатом работы будет файл main.o, его можно посмотреть с помощью **hexdump** или **objdump**.

Внимание! Под Windows Установите утилиты [util-linux - MSYS2 Packages](#) Не забудьте добавить в PATH c:\msys64\usr\bin\ или ваш путь.

Ассемблирование

```
$ gcc -c main.s -o main.o
```

```
$ hexdump main.o
```

```
00000000 cf fa ed fe 07 00 00 01 03 00 00 00 01 00 00 00
00000010 04 00 00 00 08 02 00 00 00 20 00 00 00 00 00 00
00000020 19 00 00 00 88 01 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 98 00 00 00 00 00 00 00 00 28 02 00 00 00 00 00
00000050 98 00 00 00 00 00 00 00 00 07 00 00 00 07 00 00
00000060 04 00 00 00 00 00 00 00 00 5f 5f 74 65 78 74 00 00
00000070 00 00 00 00 00 00 00 00 00 5f 5f 54 45 58 54 00 00
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000090 2a 00 00 00 00 00 00 00 00 28 02 00 00 04 00 00
```

```
...
```

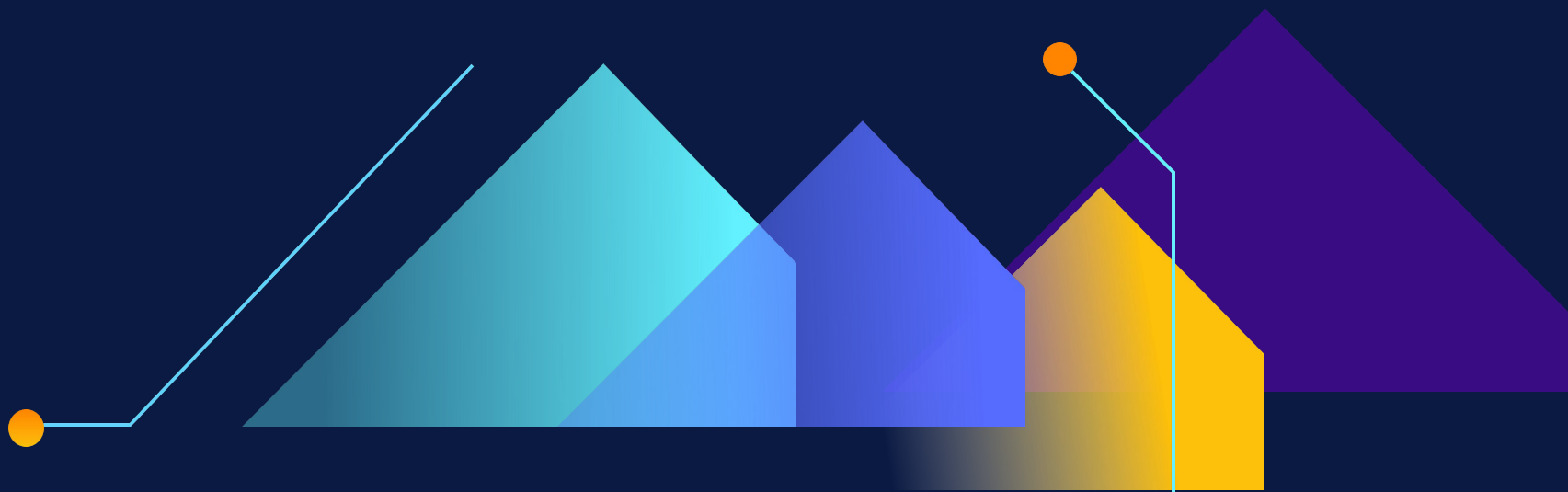
Линковка

Препроцессированный, скомпилированный и ассемблированный объектный код, наконец, готов для преобразования в исполняемый файл. Для этого компилятор делает последний шаг и отправляет код компоновщику, который берёт все переданные ему объектные коды и библиотеки и связывает их в один исполняемый файл.

Можно запустить компилятор с дополнительным флагом `-o` и явно задать имя исполняемого файла.

```
$ gcc -o program main.o
```


Сравнение gcc и clang



Сравнение gcc и clang

Компилятор GCC (GNU C Compiler) был создан в 1984 году, его основное предназначение заключалось в создании ОС Unix на языке C. С развитием компилятор стал поддерживать и другие языки программирования, такие как:

- C
- C++
- Objective-C
- Fortran
- Ada
- Go
- D



Он доступен почти во всех Unix-подобных системах, также в MinGW для Windows. Это очень ухоженный и выверенный компилятор, поддерживающий дух свободного программного обеспечения.

[clang - MSYS2 Packages](#)

Сравнение gcc и clang

Компилятор Clang — это frontend-backend компилятор, который используется для компиляции языков программирования, таких как C++, C, Objective C++ и Objective C, в машинный код. Он использует компилятор LLVM в качестве серверной части и был включён в состав LLVM.

Clang также создан как альтернатива GCC. По своей конструкции компилятор Clang был построен аналогично GCC, чтобы гарантировать максимальную переносимость. Однако разница между ними всё же есть.

[C++ | Первая программа на Windows. Компилятор Clang](#)



Сравнение gcc и clang

Процесс компиляции Clang состоит из трёх разных этапов:

1. Первый этап — это **frontend**, который используется для анализа исходного кода. Он проверяет код на наличие ошибок и строит зависящее от языка абстрактное синтаксическое дерево (AST) для работы в качестве входного кода.
2. Второй этап — это **оптимизатор**, который используется для оптимизации AST, созданного frontend-ом.
3. Третий и последний этап — это **backend**. Он отвечает за генерацию окончательного кода, который будет выполняться машиной.

Сравнение

Поддерживаемые платформы. GCC и Clang поддерживают почти все современные платформы. Clang/LLVM изначально компилируется в Windows, тогда как для GCC для работы под Windows нужна подсистема (такая, как MinGW). Изначально GCC не был задуман поддерживать Windows.

Эффективность оптимизации кода. Пространственная и временная сложность сгенерированного кода в Clang и GCC сопоставимы. Сравнение GCC и Clang в данном случае не имеет смысла.

Независимая от языка система типов. Clang/LLVM использует независимую от языка систему типов для всех поддерживаемых языков, можно определить точную семантику инструкции. GCC не преследует цель создания системы типов, не зависящей от языка, он изначально ориентирован на язык C.

Сравнение

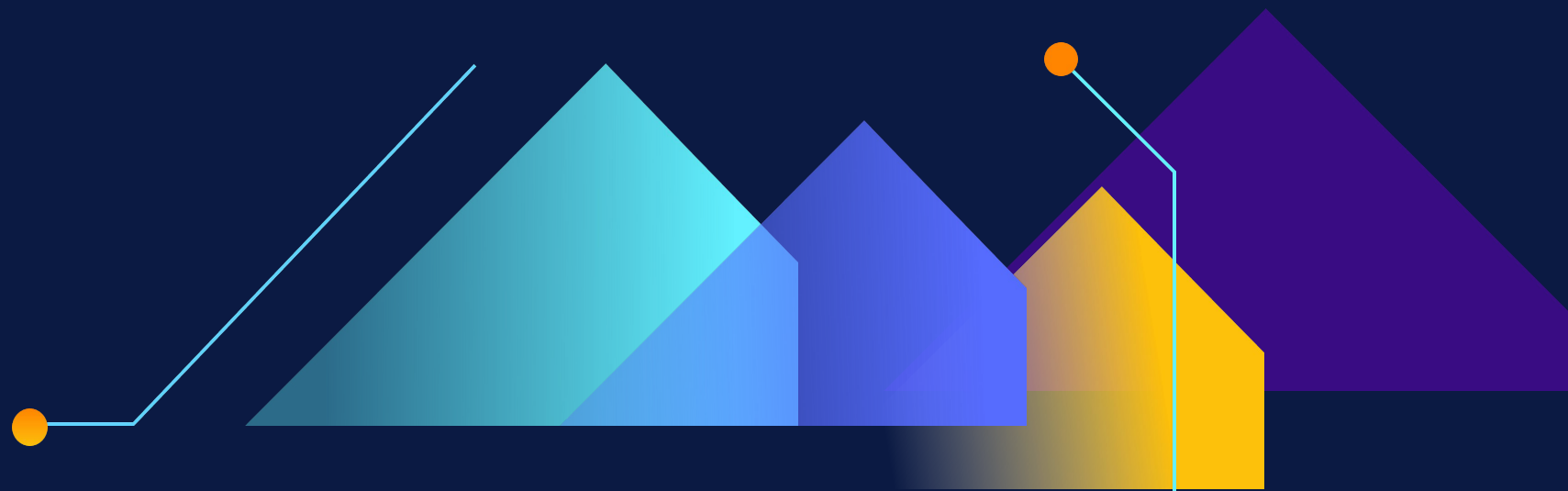
Линковка. GCC Vs Clang здесь наиболее заметен. GCC использует ld в качестве линковщика с поддержкой ld-gold. Clang же использует lld. С помощью некоторых тестов видно, что lld быстрее ld, даже нового ld-gold.

Отладчик. GCC имеет отличный отладчик GDB — проверено временем и хорошо работает. Clang имеет сборку отладчика LLDB.

Как видно из названия, Clang поддерживает в основном C, C++ и Objective-C. Но структура под названием LLVM, лежащая в основе Clang, достаточно расширяема, чтобы поддерживать новые языки, такие как Julia и Swift.

С точки зрения C, оба являются отличными компиляторами.

Оптимизация



Компиляция без оптимизации


Без каких-либо опций цель компилятора — снизить стоимость компиляции и добиться при отладке ожидаемых результатов.

Все операторы изолированы: если мы ставим breakpoint между операторами, то затем можно присвоить новое значение любой переменной или изменить счётчик программы на любой другой оператор в функции и получить именно нужные нам результаты.

Включение флагов оптимизации заставляет компилятор пытаться улучшить производительность, а также размер кода за счёт времени компиляции и отключения возможности отладки программы. Не пытайтесь компилировать не отлаженную программу с оптимизационными флагами.

Компилятор выполняет оптимизацию на основе имеющихся у него знаний о программе. Компиляция нескольких файлов одновременно в один исполняемый файл позволяет компилятору использовать информацию, полученную из всех файлов и собранную при компиляции каждого из них.

Оптимизация



ГСС — компилятор, который умеет оптимизировать код. Он предоставляет широкий спектр опций, направленных на увеличение скорости или уменьшение размера исполняемых файлов.

Оптимизация — это сложный процесс. Для каждой высокоуровневой команды в исходном коде обычно существует множество возможных комбинаций машинных инструкций, которые можно использовать для достижения соответствующего конечного результата. Компилятор должен уметь учитывать эти возможности и выбирать среди них.

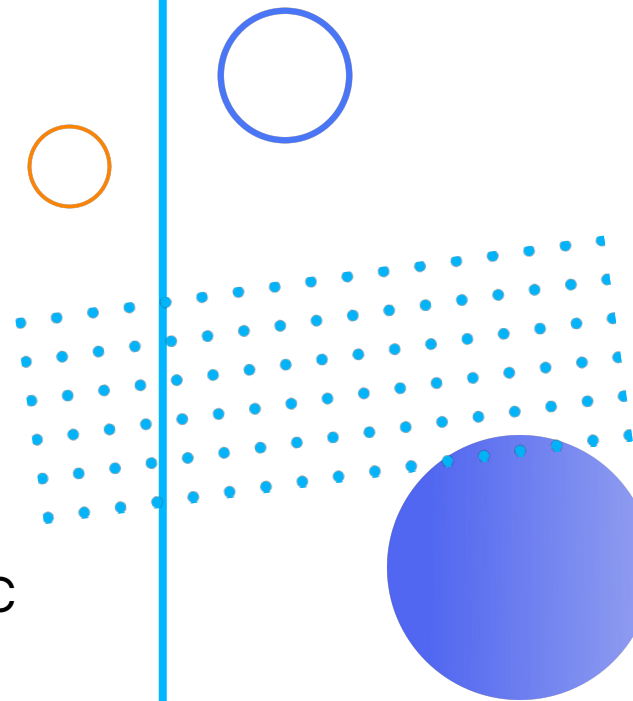


Оптимизация

Как правило, для разных процессоров необходимо сгенерировать разный код, поскольку они используют несовместимые языки сборки и машинные языки. У каждого типа процессора также есть свои особенности — некоторые процессоры предоставляют большое количество регистров для хранения промежуточных результатов вычислений, в то время как другие должны сохранять и извлекать промежуточные результаты из памяти.

В каждом случае это необходимо учесть и сгенерировать соответствующий код.

Кроме того, для выполнения разных инструкций требуется разное количество времени, в зависимости от того, как они упорядочены. GCC принимает во внимание все эти факторы и пытается создать самый быстрый исполняемый файл для данной системы при компиляции с оптимизацией.



Оптимизация с использованием ключей

При компиляции обычно использует ключ **-O** с дополнительным индексом. Большинство оптимизаций полностью отключается при **-O0** или если ключ **-O** не задан в командной строке, даже если указаны отдельные флаги оптимизации. Увеличение уровня оптимизации с **-O1**, **-O2** и **-O3** приводит к увеличению ускорения по сравнению с не оптимизированным кодом, скомпилированным с **-O0**.

Пример

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

double pown (double d, unsigned n){
    double x = 1.0;
    for (size_t j = 1; j <= n; j++) {
        x *= d;
    }
    return x;
}

int main (void) {
    double sum = 0.0;

    for (size_t i = 1; i <= 0xffffffff; i++) {
        sum += pown (i, i % 5);
    }
    printf ("sum = %g\n", sum);
    return 0;
}
```

```
$ gcc -O0 main.c -lm
$ time ./a.out
sum = 5.57519e+40
real    0m1.882s
user    0m1.562s
sys     0m0.004s
```

```
$ gcc -O1 main.c -lm
$ time ./a.out
sum = 5.57519e+40
real    0m1.164s
user    0m1.157s
sys     0m0.004s
```

```
$ gcc -O2 main.c -lm
$ time ./a.out
sum = 5.57519e+40
real    0m0.796s
user    0m0.787s
sys     0m0.004s
```

```
$ gcc -O3 main.c -lm
$ time ./a.out
sum = 5.57519e+40
real    0m0.779s
user    0m0.772s
sys     0m0.004s
```

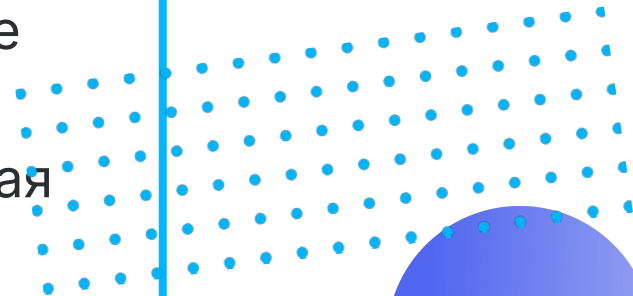
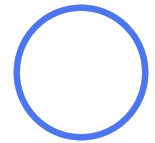


Оптимизация

Результат вывод для процессора: Intel(R) Core(TM) i5-1030NG7 CPU @ 1.10GHz.

Одним из способов измерения времени является утилита `time` GNU Bash. В качестве выходных данных она выдает три параметра:

- **user** — это «пользовательское» время, которое даёт фактическое время процессора, затраченное на выполнение процесса.
- **real** — общее реальное время выполнения процесса (включая время, когда другие процессы использовали ЦП)
- **sys** — время, потраченное на ожидание вызовов операционной системы.



Скрипт для измерения времени timescmd.bat

```
@echo off
@setlocal
set start=%time%
:: Runs your command
cmd /c %*
set end=%time%
set options="tokens=1-4 delims=.:,"
for /f %options% %%a in ("%start%") do set start_h=%%a&set /a start_m=100%%b %% 100&set /a start_s=100%%c %% 100&set /a start_ms=100%%d %% 100
for /f %options% %%a in ("%end%") do set end_h=%%a&set /a end_m=100%%b %% 100&set /a end_s=100%%c %% 100&set /a end_ms=100%%d %% 100
set /a hours=%end_h%-start_h%
set /a mins=%end_m%-start_m%
set /a secs=%end_s%-start_s%
set /a ms=%end_ms%-start_ms%
if %ms% lss 0 set /a secs = %secs% - 1 & set /a ms = 100%ms%
if %secs% lss 0 set /a mins = %mins% - 1 & set /a secs = 60%secs%
if %mins% lss 0 set /a hours = %hours% - 1 & set /a mins = 60%mins%
if %hours% lss 0 set /a hours = 24%hours%
if 1%ms% lss 100 set ms=0%ms%
:: Mission accomplished
set /a totalsecs = %hours%*3600 + %mins%*60 + %secs%
echo command took %hours%:%mins%:%secs%.%ms% (%totalsecs%.%ms%s total)
```



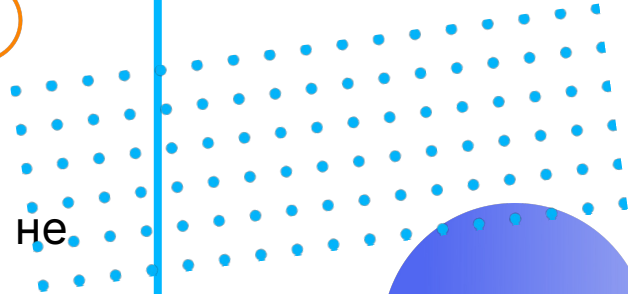
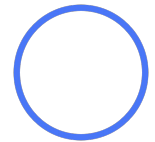
Оптимизация

Каждая из опций (**-00**, **-01**, **-02**, **-03**) подразумевает включение определенного набора флагов, описание которых доступно в документации компилятора или в man.

Бывают ситуации, когда после компиляции с флагом **-01** программа работает быстрее, чем при компиляции с флагом **-02**.

Рассмотрим, за что отвечает каждый из ключей: **-00**, **-01**, **-02**, **-03**, **-Os**.

Внимание! оптимизация сильно зависит от конкретной архитектуры и не обязательно делает программу быстрее





Оптимизация

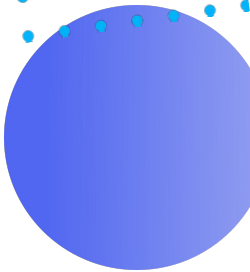
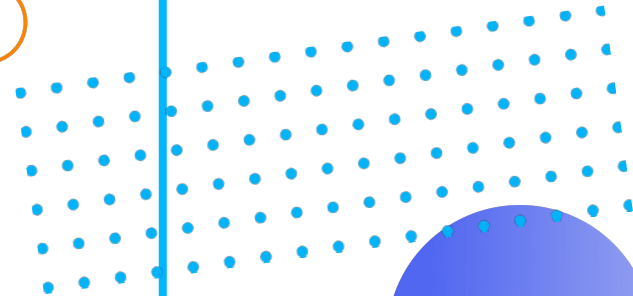
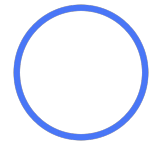
-O0 или без ключа O.

Компилятор GCC не выполняет никакой оптимизации и компилирует исходный код наиболее простым способом.

Каждая команда в исходном коде преобразуется непосредственно в соответствующие инструкции в исполняемом файле без изменения

Используется по умолчанию, если не указан параметр уровня оптимизации.

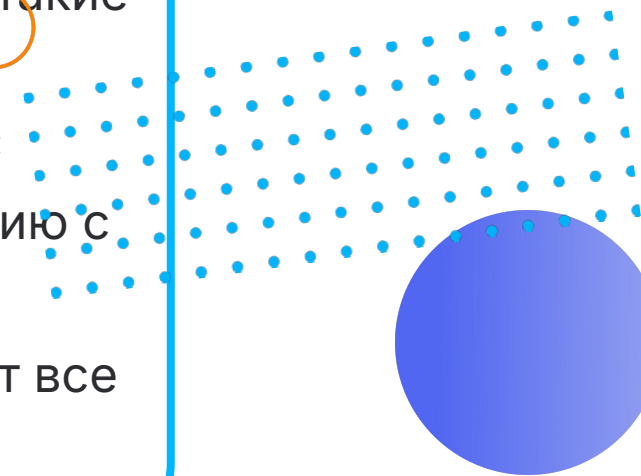
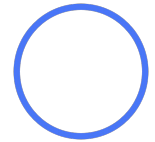
Этот вариант лучше всего использовать при отладке программы!





Оптимизация

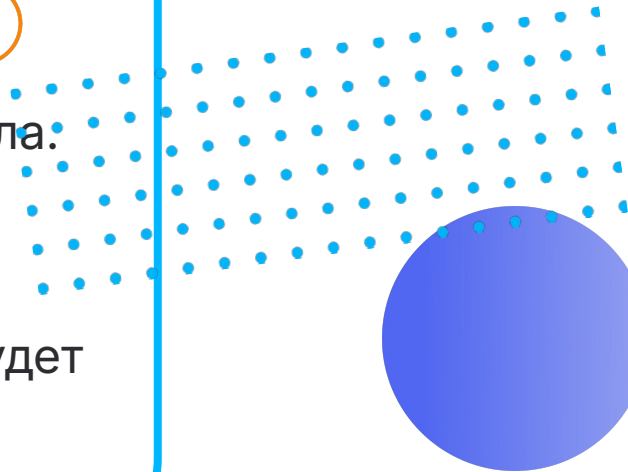
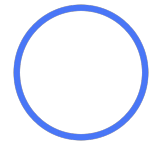
- **-O1.** Процесс оптимизации занимает чуть больше времени, компилятор использует больше памяти. В данном случае компилятор пытается уменьшить размер кода и время выполнения, не выполняя никаких продвинутых оптимизаций, которые занимали бы много времени при компиляции. Более дорогие способы оптимизации, такие как scheduling, на этом уровне не используются.
- **-O2.** Данный процесс оптимизации ещё более продвинутый. GCC выполняет почти все поддерживаемые оптимизации. По сравнению с -O1 этот параметр увеличивает время компиляции и производительность сгенерированного кода. Также -O2 включает все флаги оптимизации, указанные в -O1.





Оптимизация

- **-O3.** Следующий уровень оптимизации включает все оптимизации, указанные параметром `-O2`, а также некоторые дополнительные флаги. Обычно этот уровень увеличивает размер исполняемого файла. Также в него включена опция `loop-unroll` (развёртывание цикла), которая не всегда приводит к ускорению, но почти всегда увеличивает размер исполняемого файла.
- **-Os.** Эта опция оптимизации уменьшает размер исполняемого файла. Её цель — создать исполняемый файл минимального размера для систем, ограниченных памятью или дисковым пространством. В некоторых случаях исполняемый файл меньшего размера также будет работать быстрее из-за лучшего использования кеша.



Оптимизация с использованием ключей

Преимущества оптимизации на самом высоком уровне необходимо сопоставлять с затратами. Стоимость оптимизации включает более сложную отладку, а также увеличенные требования к времени и памяти во время компиляции. Для большинства программ достаточно использовать -O0 для отладки и -O2 для разработки.

В GCC можно использовать оптимизацию в сочетании с параметром отладки -g. Другие компиляторы обычно этого не позволяют.

При одновременном использовании отладки и оптимизации внутренние перестройки, выполняемые оптимизатором, могут затруднить понимание того, что происходит при проверке оптимизированной программы в отладчике. Например, временные переменные часто удаляются, а порядок операторов может быть изменён.

Внимание! В случае неожиданного сбоя программы любая отладочная информация лучше, чем её отсутствие, поэтому для оптимизированных программ рекомендуется использовать -g как для разработки, так и для развёртывания. Параметр отладки -g включён по умолчанию для выпусков пакетов GNU вместе с параметром оптимизации -O2.

Проверка переменных

В процессе оптимизации компилятор проверяет использование всех переменных и их начальных значений — это называется анализом потока данных.

Он формирует основу для возможных стратегий оптимизации, таких как планирование инструкций.

Побочным эффектом анализа потока данных является то, что компилятор может обнаруживать использование неинициализированных переменных.

Параметр `-Wuninitialized` (включается с `-Wall`) предупреждает о переменных, которые читаются без инициализации.

Планирование (scheduling)

Самый низкий уровень оптимизации — это планирование(scheduling), при котором компилятор определяет наилучший порядок отдельных инструкций.

- Большинство процессоров позволяют одной или нескольким новым инструкциям начать выполнение до того, как завершатся другие.
- Многие многоядерные процессоры также поддерживают конвейерную обработку (pipeline), при которой несколько инструкций выполняются на одном и том же процессоре параллельно.
- Когда планирование включено, инструкции должны быть организованы так, чтобы их результаты становились доступными для последующих инструкций в нужное время, и чтобы обеспечить максимальное параллельное выполнение.
- Планирование увеличивает скорость исполняемого файла без увеличения его размера, но требует дополнительной памяти и времени при компиляции.
- Этот метод включён в -O2 -O3.

Пример

Рассмотрим пример, в котором реализована рекурсивная функция вычисления факториала. Вынесем реализацию функции в отдельный файл и будем собирать программу по частям.

main.c

```
#include <stdio.h>
extern unsigned int fact(unsigned
int);
int main (void) {
    unsigned int n;
    scanf ("%u", &n);
    printf ("fact = %lld\n", fact (
n ));
    return 0;
}
```

function.c

```
unsigned int fact(unsigned n) {
    if (n==0)
        return 1;
    return n * fact(n-1);
}
```

Соберём программу без ключей оптимизации

```
$ gcc -c -o main.o main.c  
$ gcc -c -o function.o function.c  
$ gcc -o prog function.o main.o
```

Утилита objdump

Утилита **objdump** позволит отобразить файл function.o, скомпилированный из файла function.c, в виде ассемблерных команд.

Disassembly of section **.text**:

0000000000000000 <fact>:

0:	55	push	%rbp
1:	48 89 e5	mov	%rsp,%rbp
4:	48 83 ec 20	sub	\$0x20,%rsp
8:	89 4d 10	mov	%ecx,0x10(%rbp)
b:	83 7d 10 00	cmpl	\$0x0,0x10(%rbp)
f:	75 07	jne	18 <fact+0x18>
11:	b8 01 00 00 00	mov	\$0x1,%eax
16:	eb 11	jmp	29 <fact+0x29>
18:	8b 45 10	mov	0x10(%rbp),%eax
1b:	83 e8 01	sub	\$0x1,%eax
1e:	89 c1	mov	%eax,%ecx
20:	e8 db ff ff ff	call	0 <fact>
25:	0f af 45 10	imul	0x10(%rbp),%eax
29:	48 83 c4 20	add	\$0x20,%rsp
2d:	5d	pop	%rbp
2e:	c3	ret	
2f:	90	nop	

Используем ключ -O2

Из кода видно, что в теле функции `fact` происходит рекурсивный вызов самой себя (строка `2f`). Компилятор построил данный код самым примитивным образом — без оптимизации.

Соберём файл `function.o`, используя ключ `-O2`, который также подразумевает замену хвостовой рекурсии на цикл.

```
$ gcc -c -o function.o function.c -O2  
$ objdump -D function.o
```

Результат

Как видно из листинга, количество строк машинного кода сократилось до 18. Также отсутствует инструкция call.

Disassembly of section `.text`:

0000000000000000 <fact>:

0:	b8 01 00 00 00	mov	\$0x1,%eax
5:	85 c9	test	%ecx,%ecx
7:	74 0f	je	18 <fact+0x18>
9:	0f 1f 80 00 00 00 00	nopl	0x0(%rax)
10:	0f af c1	imul	%ecx,%eax
13:	83 e9 01	sub	\$0x1,%ecx
16:	75 f8	jne	10 <fact+0x10>
18:	c3	ret	

Используем ключ -Os

Как видно из листинга, в файле `function.o` отсутствует инструкция `call`, подразумевающая вызов функции. Однако стоит заметить, что сам машинный код стал занимать гораздо больше места (69 строк против 391).

Пересоберём файл `function.o` с ключом `-Os`, который попросит компилятор уменьшить размер исполняемого файла.

```
$ gcc -c -o function.o function.c -Os
$ objdump -disassemble -x86-asm-syntax=intel function.o
```

Результат

Как видно из листинга, количество строк машинного кода сократилось до 10. Также отсутствует инструкция call.

```
0000000000000000 <fact>:
```

```
0:    b8 01 00 00 00
```

```
5:    85 c9
```

```
7:    74 07
```

```
9:    0f af c1
```

```
c:    ff c9
```

```
e:    eb f5
```

```
10:   c3
```

```
mov    $0x1,%eax
```

```
test   %ecx,%ecx
```

```
je     10 <fact+0x10>
```

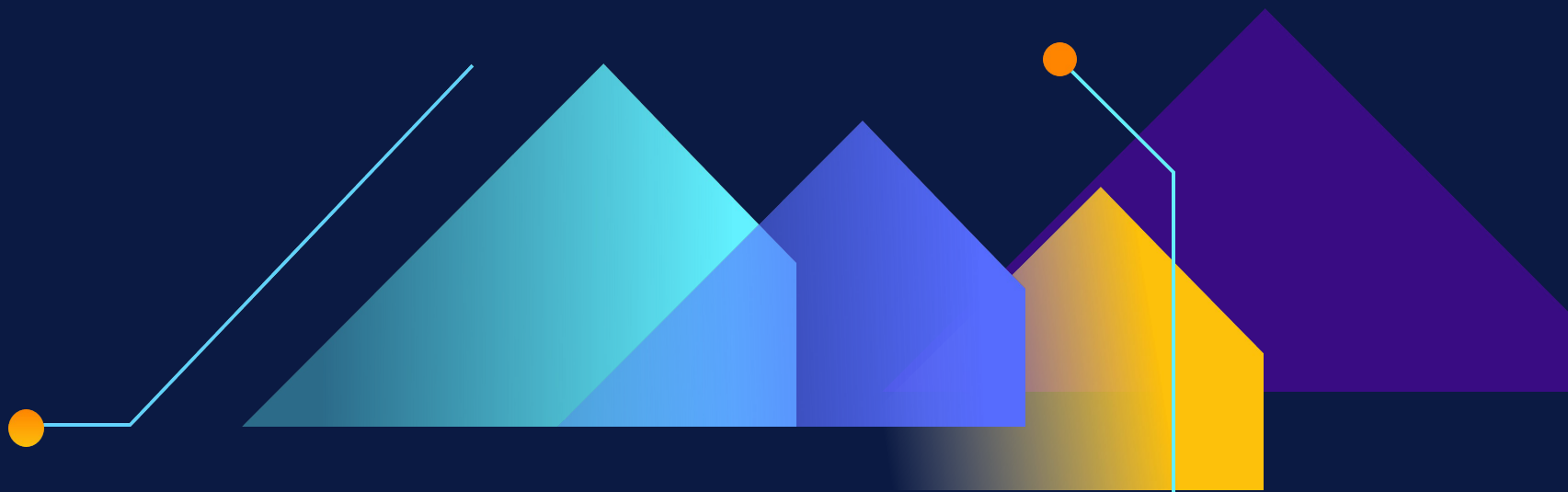
```
imul   %ecx,%eax
```

```
dec    %ecx
```

```
jmp     5 <fact+0x5>
```

```
ret
```

Исследуем исполняемый файл



Утилита file

Рассмотрим несколько полезных инструментов для анализа исполняемых файлов.

После того как исходный файл был скомпилирован в объектный файл или исполняемый файл, параметры, используемые для его компиляции, больше не очевидны. Команда `file` просматривает содержимое объектного файла или исполняемого файла и определяет некоторые из его характеристик, например, был ли он скомпилирован с динамической или статической компоновкой.

```
$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 3.2.0,
BuildID[sha1]=935b22c022981fe9d9b3b4b39fcbf7e970762413, not
stripped
```

Утилита nm

Исполняемый файл содержит таблицу символов (её можно удалить с помощью команды `strip`). В этой таблице хранится расположение функций и переменных по имени, и её можно отобразить с помощью команды `nm`:

```
$ nm a.out
0000000000200db8 d _DYNAMIC
0000000000200fa8 d _GLOBAL_OFFSET_TABLE_
...
00000000000000768 T main
000000000000006f5 T print
                  U printf@@GLIBC_2.2.5
00000000000000670 t register_tm_clones
```

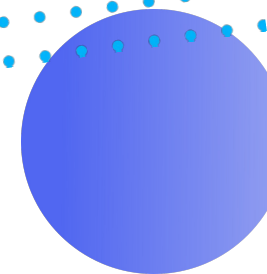
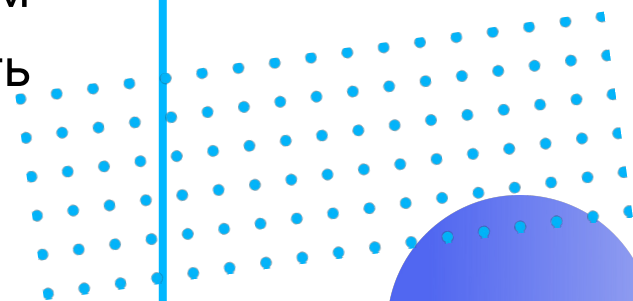
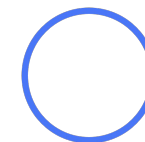


Утилита nm

Среди содержимого таблицы символов видно, что начало основной функции имеет шестнадцатеричное смещение 00000000000000768.

Большинство символов предназначены для внутреннего использования компилятором и операционной системой. «Т» во втором столбце указывает на функцию, определённую в объектном файле, а «U» — на функцию, которая не определена (и должна быть разрешена путём связывания с другим объектным файлом).

Чаще всего команда nm используется для проверки наличия в библиотеке определения конкретной функции путём поиска записи «Т» во втором столбце напротив имени функции.



Утилита nm

Исследуем исполняемый файл prog состоящий из двух объектных файлов main.o и func.o.

```
//function.c
unsigned int fact(unsigned n)
{
    if(n==0)
        return 1;
    return n * fact(n-1);
}
```

```
//main.c
#include <stdio.h>
extern unsigned int
fact(unsigned int);
int main (void) {
    unsigned int n;
    scanf ("%u", &n);
    printf ("fact =
    %u\n", fact(n));
    return 0;
}
```

Утилита nm

```
$ nm -A *.o
function.o:000000000000000000 T fact
function.o:000000000000000004 C my_var
function.o:000000000000000000 D my_var2
main.o:                U _GLOBAL_OFFSET_TABLE_
main.o:                U __isoc99_scanf
main.o:                U __stack_chk_fail
main.o:                U fact
main.o:000000000000000000 T main
main.o:                U printf
```

Утилита nm

Обратите внимание на функцию fact, она была ее адрес был неизвестен до этапа линковки.

```
$ nm -A prog
prog:0000000000200db8 d _DYNAMIC
prog:0000000000200fa8 d _GLOBAL_OFFSET_TABLE_
...
prog:0000000000000075c T fact
prog:000000000000006f0 t frame_dummy
prog:000000000000006f5 T main
prog:                U printf@@GLIBC_2.2.5
prog:00000000000000670 t register_tm_clones
```

Утилита ldd

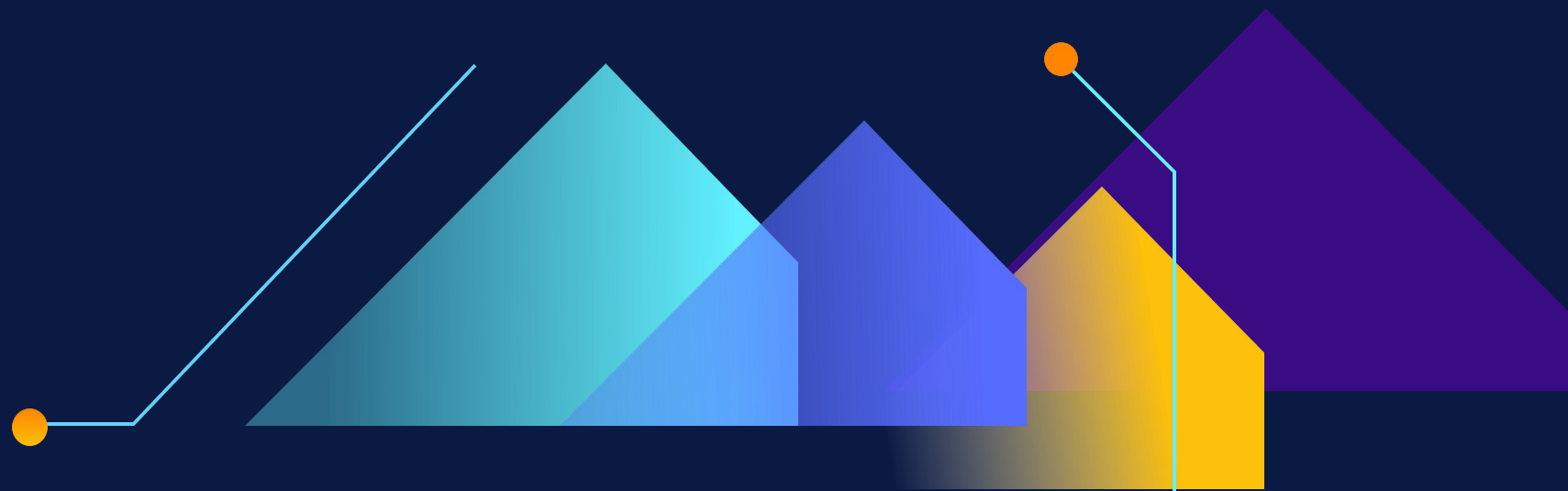
Когда программа скомпилирована с использованием разделяемых библиотек, ей необходимо динамически загружать эти библиотеки во время выполнения, чтобы вызывать внешние функции. Команда ldd проверяет исполняемый файл и отображает список необходимых ему разделяемых библиотек. Эти библиотеки называются зависимостями разделяемых библиотек исполняемого файла.

Например, следующие команды демонстрируют, как найти зависимости разделяемых библиотек программы Hello World:

```
$ gcc helloworld.c
$ ldd a.out
    linux-vdso.so.1 (0x00007fff287ae000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007fa617432000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fa617a25000)
```

Программа Hello World зависит от библиотеки libc и библиотеки динамического загрузчика ld-linux.

Ассемблер

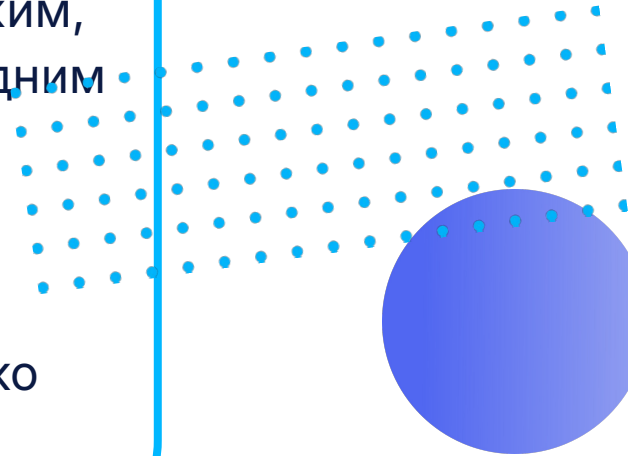
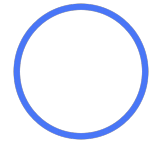




Утилита nm

Почему стоит изучать код ассемблера своей программы? Есть несколько причин, по которым следует погрузиться в эту среду.

- Изучая ассемблерный код можно понять, что компилятор действительно сделал, а что нет.
- Иногда найти ошибку можно только на нижнем уровне. Предположим, какая-то ошибка появилась только после запуска компилятора с одним из ключей оптимизации.
- Можно модифицировать ассемблерный код вручную, если другие подходы уже не работают.
- Можно понять, что делает данная программа, если в наличии только исполняемый файл



Пример

Рассмотрим пример, в котором происходит проверка пароля на соответствие. Если пароль введен верно, то программа начинает работать. У вас на руках только исполняемый файл данной программы — prog. Задача: обойти проверку пароля 😊.

```
_Bool checkPass(char *p) {
    if(strcmp(p, "secret")==0)
        return 1;
    return 0;
}

int main(void) {
    char password[100];
    printf("Input your password: ");
    scanf("%s", password);
    if(checkPass(password))
        printf("Access granted\n");
    else
        printf("Access denied\n");
    return 0;
}
```

```
$ gcc -o prog main.c
```

```
$ ./prog
```

Input your password: test

Access denied

```
$ ./prog
```

Input your password: secret

Access granted

Утилита strings из binutils

В этом случае программа написана не совсем удачно, и мы можем воспользоваться утилитой strings из binutils, которая выведет на печать все строки из данного бинарника, в том числе и сам пароль.

```
$ strings prog
secret
Input your password:
Access granted
Access denied
```


Пример

Рассмотрим более удачный пример, в котором пароль не хранится в явном виде, а хранится только его hash. В этом случае утилита strings не сработает, т. к в исполняемом файле пароль не хранится в виде строки.

```
uint64_t getHash(char const *s) {
    const int p = 31;
    uint64_t hash = 0, p_pow = 1;
    while(*s) {
        hash += (*s++ - 'a' + 1) * p_pow;
        p_pow *= p;
    }
    return hash;
}

_Bool checkPass(char *p) {
    if(getHash(p) == 577739920) //secret
        return 1;
    return 0;
}
```

Пример

```
int main(void) {  
    char password[100];  
    printf("Input your password: ");  
    scanf("%s", password);  
  
    if (checkPass(password))  
        printf("Access granted\n");  
    else  
        printf("Access denied\n");  
    return 0;  
}
```

```
$ gcc -o prog main.c
```

```
$/prog
```

```
Input your password: secret
```

```
Access granted
```

```
$/prog
```

```
Input your password: test
```

```
Access denied
```

```
$ strings prog
```

```
Input your password:
```

```
Access granted
```

```
Access denied
```

Утилита objdump с ключом -d

Воспользуемся другой полезной утилитой **objdump** с ключом **-d**, которая отобразит дизассемблированный код нашего файла. Нас интересует только функция проверки пароля. Посмотрев на ассемблерный код, можно понять, в каком месте происходит проверка, и внести изменения в бинарный файл. Изменения должны быть минимальны, иначе придётся вручную рассчитывать все сдвиги адресов, а это очень трудно. Поэтому мы изменим только одну инструкцию JNE (jump if not equal ее код - 0F 85) — просто заменим её на похожую: инструкцию JE(jump if equal ее код 0F 84). Логика проверки пароля кардинально изменится, и можно будет ввести любой пароль, кроме корректного.

[je to jne in assembly - Stack Overflow](#)

Утилита objdump с ключом -d

```
$ objdump -d prog
prog: file format Mach-O 64-bit x86-64
Disassembly of section __TEXT,__text:
...
0000000100003e40 _checkPass:
100003e40: 55                pushq %rbp
100003e41: 48 89 e5          movq %rsp, %rbp
100003e44: 48 83 ec 20       subq $32, %rsp
100003e48: 48 89 7d f0       movq %rdi, -16(%rbp)
100003e4c: 48 c7 45 e8 90 9c 6f 22 movq $577739920, -24(%rbp)
100003e54: 48 8b 7d f0       movq -16(%rbp), %rdi
100003e58: e8 73 ff ff ff   callq -141 <_getHash>
100003e5d: 48 3b 45 e8       cmpq -24(%rbp), %rax
100003e61: 0f 85 09 00 00 00 jne 9 <_checkPass+0x30>
100003e67: c6 45 ff 01      movb $1, -1(%rbp)
100003e6b: e9 04 00 00 00   jmp 4 <_checkPass+0x34>
100003e70: c6 45 ff 00      movb $0, -1(%rbp)
100003e74: 8a 45 ff         movb -1(%rbp), %al
100003e77: 24 01           andb $1, %al
100003e79: 0f b6 c0        movzbl %al, %eax
100003e7c: 48 83 c4 20     addq $32, %rsp
100003e80: 5d             popq %rbp
100003e81: c3             retq
100003e82: 66 2e 0f 1f 84 00 00 00 00 00 nopw %cs:(%rax,%rax)
100003e8c: 0f 1f 40 00     nopl (%rax)
...

```

Утилита objdump с ключом -d

В данном выводе нас интересует адрес смещения в файле, по которому надо произвести замену: 00003e61. Для редактирования файла воспользуемся консольным редактором vim с ключом -b, который откроет файл в бинарном формате. Внутри редактора vim используем команду xxd для преобразования файла в шестнадцатеричное представление.

```
$ vim -b prog
:%!xxd -g1
:/3E00
00003e00: 48 89 c1 48 81 c1 01 00 00 00 48 89 4d f8 0f be  H..H.....H.M...
00003e10: 10 83 ea 61 83 c2 01 48 63 c2 48 0f af 45 e0 48  ...a...Hc.H..E.H
00003e20: 03 45 e8 48 89 45 e8 48 69 45 e0 1f 00 00 00 48  .E.H.E.HiE.....H
00003e30: 89 45 e0 e9 b7 ff ff ff 48 8b 45 e8 5d c3 66 90  .E.....H.E.].f.
00003e40: 55 48 89 e5 48 83 ec 20 48 89 7d f0 48 c7 45 e8  UH..H.. H.}.H.E.
00003e50: 90 9c 6f 22 48 8b 7d f0 e8 73 ff ff ff 48 3b 45  ..o"H.}..s...H;E
00003e60: e8 0f 85 09 00 00 00 c6 45 ff 01 e9 04 00 00 00  ....E.....
00003e70: c6 45 ff 00 8a 45 ff 24 01 0f b6 c0 48 83 c4 20  .E...E.$....H..
```

Утилита objdump с ключом -d

Отредактируем найденную строку заменив инструкцию 0f 85 (JNE - Jump if Not Equal) на инструкцию 0f 84 (JE - Jump if Equal) и сохраним файл. После внесения изменений (в шестнадцатеричной части) вы можете вернуться к тексту с помощью команды -r на xxd.

```
:%!xxd -r  
:wq
```

Утилита objdump с ключом -d

Проверим, что всё верно сделали с помощью objdump:

```
$ objdump -d prog
0000000100003e40 _checkPass:
100003e40: 55                                pushq   %rbp
100003e41: 48 89 e5                        movq    %rsp, %rbp
100003e44: 48 83 ec 20                     subq    $32, %rsp
100003e48: 48 89 7d f0                     movq    %rdi, -16(%rbp)
100003e4c: 48 c7 45 e8 90 9c 6f 22        movq    $577739920, -24(%rbp)
100003e54: 48 8b 7d f0                     movq    -16(%rbp), %rdi
100003e58: e8 73 ff ff ff                callq   -141 <_getHash>
100003e5d: 48 3b 45 e8                     cmpq    -24(%rbp), %rax
100003e61: 0f 84 09 00 00 00             je      9 <_checkPass+0x30>
100003e67: c6 45 ff 01                    movb     $1, -1(%rbp)
100003e6b: e9 04 00 00 00                jmp     4 <_checkPass+0x34>
100003e70: c6 45 ff 00                    movb     $0, -1(%rbp)
```

Утилита objdump с ключом -d

Запустим файл prog и убедимся, что всё работает:

```
$ ./prog
```

```
Input your password: test
```

```
Access granted
```

```
$ ./prog
```

```
Input your password: anotherTest
```

```
Access granted
```


Задание



Дедлайн: конец курса

Советуем регулярно выполнять ДЗ
(наверстать пропуски тяжело)

1. Реализовать пропущенный код
2. * внести изменения в исполняемый файл, чтобы в стандартном режиме змейка не погибала при самопересечении. Для этого необходимо найти вызов функции `isCrush()` и поменять вызов на нужное нам возвращаемое значение 0
3. * Добавить цвет для двух змеек и еды.
4. * Добавить стартовое меню (приветствие, выбор режима, выбор цвета змейки и т.д.). Написать функцию `void startMenu()`
5. ** Сделать свои игровые механики

Задание

```
void setColor(int objectType) {
    attroff(COLOR_PAIR(1));
    attroff(COLOR_PAIR(2));
    attroff(COLOR_PAIR(3));
    switch (objectType) {
        case 1: { // SNAKE1
            attron(COLOR_PAIR(1));
            break;
        }
        case 2: { // SNAKE2
            attron(COLOR_PAIR(2));
            break;
        }
        case 3: { // FOOD
            attron(COLOR_PAIR(3));
            break;
        }
    }
}
```

В теле main() цвета инициализируем:

```
start_color();
init_pair(1, COLOR_RED, COLOR_BLACK);
init_pair(2, COLOR_BLUE, COLOR_BLACK);
init_pair(3, COLOR_GREEN,
COLOR_BLACK);
```