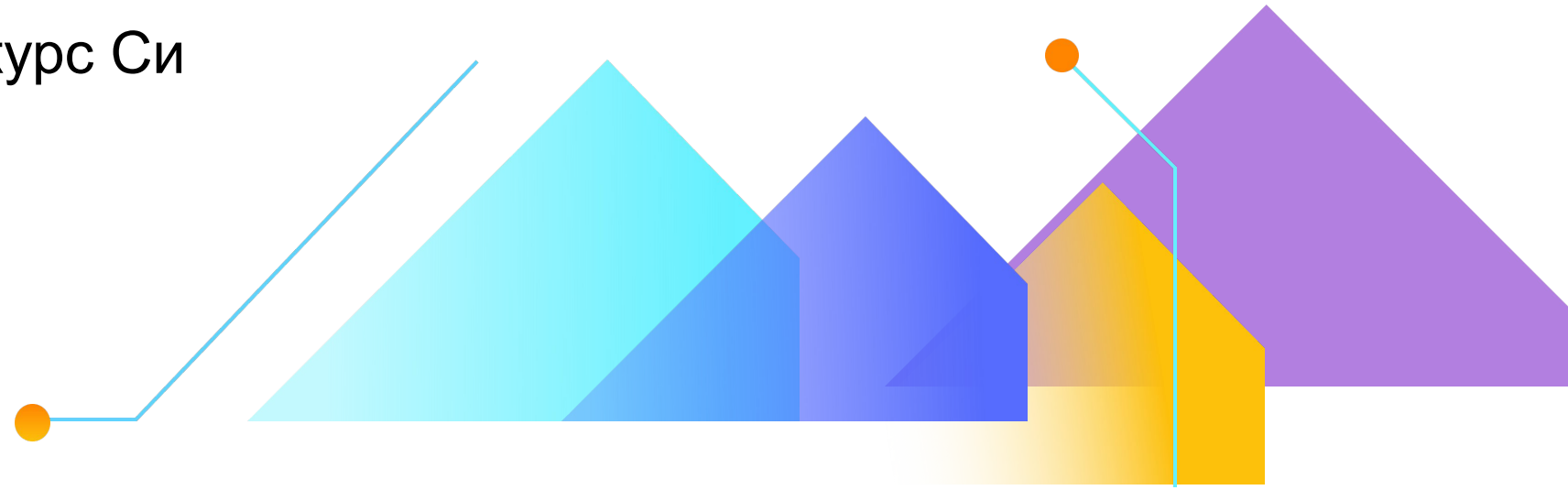
EASY EMBEDDED

Лекция №3

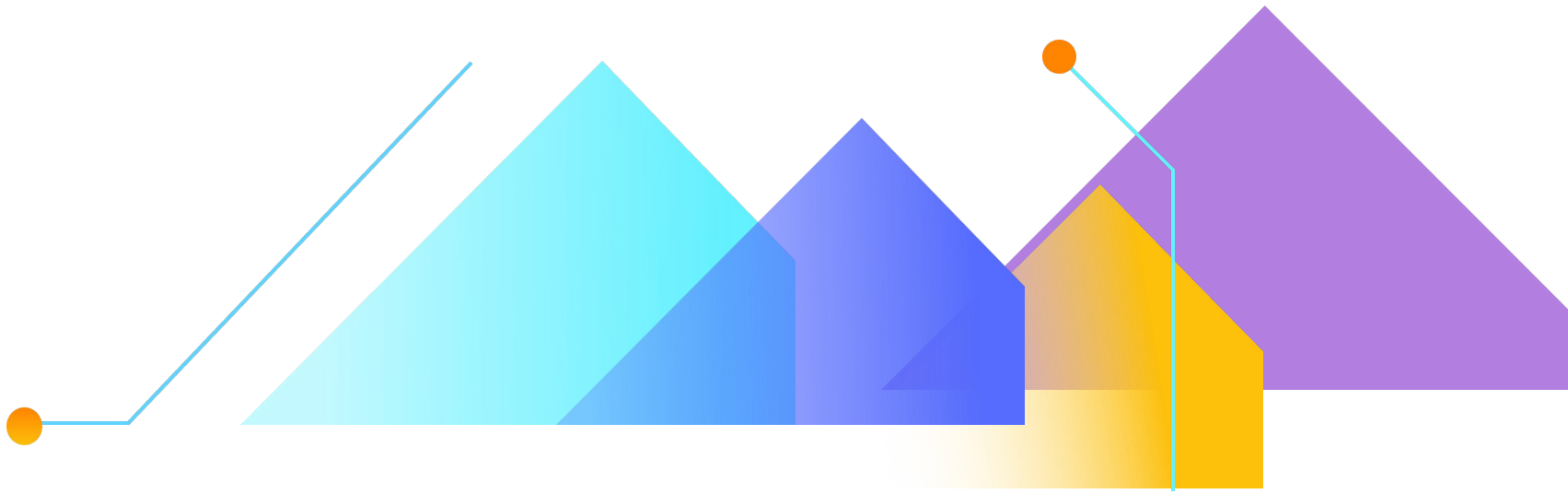
Библиотеки языка С

Продвинутый курс Си



План курса

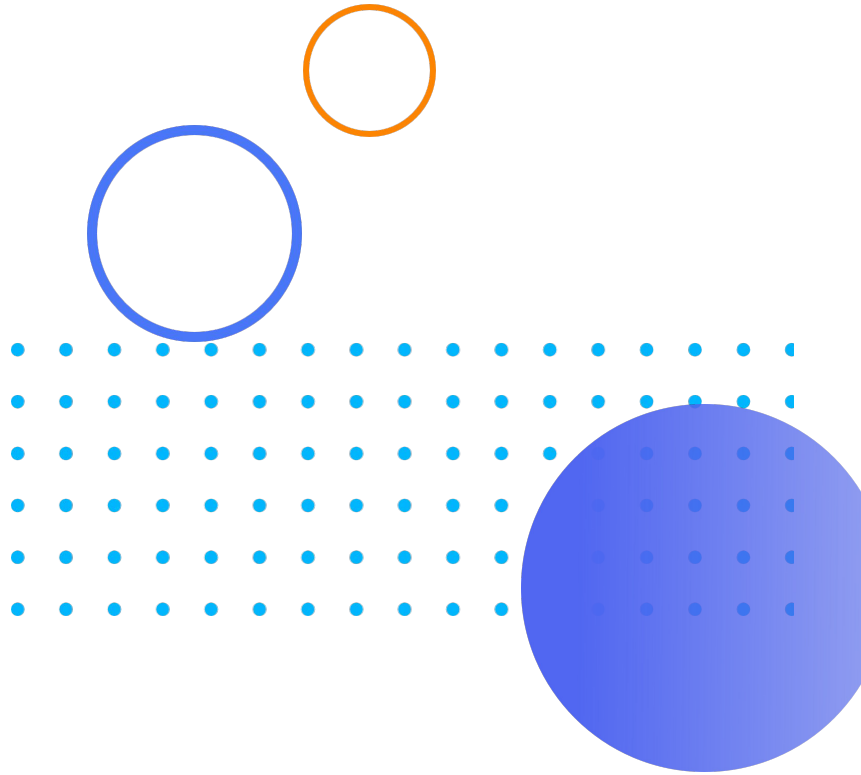
- Вводный урок
- Структуры. Динамические типы
- Библиотеки языка С
- Оптимизация кода
- Алгоритмы
- Компиляция и компиляторы
- Динамические структуры данных
- Курсовая работа



Маршрут

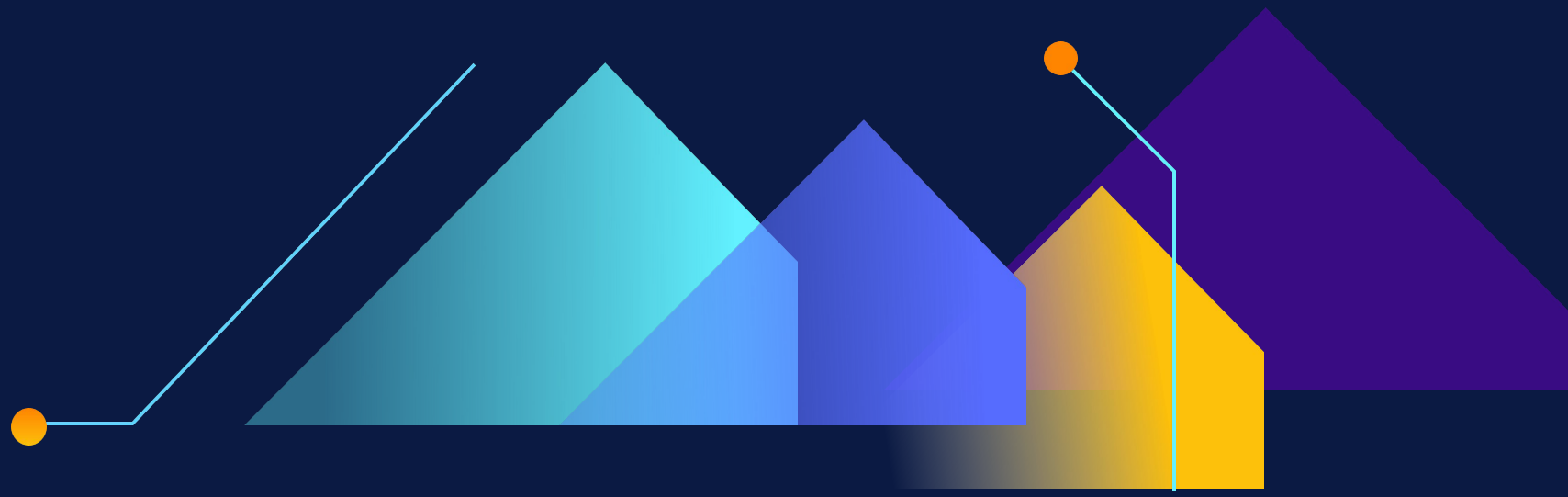
Библиотеки языка C

- Рассмотрим различные библиотеки языка C
- Повторим как работает стандартный ввод/вывод
- Изучим библиотеку для работы со строками - string.h
- Рассмотрим библиотеку - assert.h
- Порешаем задачи



EASY EMBEDDED

Библиотеки



Библиотеки

На уроке мы рассмотрим следующие библиотеки:

- **stdio.h** — библиотека ввода/вывода
- **string.h** — библиотека для работы со строками
- **assert.h** — библиотека для вывода диагностической информации
- **time.h** — библиотека для работы со временем
- **ncurses.h** — библиотека для вывода тестовой информации на терминал

Библиотеки

Раздельная трансляция файлов делает доступным систематическое повторное использование кода: объектные файлы, содержащие функции, которые могут быть использованы в разных программах, собираются в библиотеку программного кода. Библиотека представляет собой единый файл, который содержит весь код объектных файлов с некоторой дополнительной информацией (наподобие индекса в настоящих библиотеках). Последнее может ускорять процесс линковки (при использовании библиотеки) по сравнению с обычной сборкой программы из большого числа объектных файлов. Информация об исходных объектных файлах также сохраняется.

Библиотеки

Библиотеки могут существовать в двух вариациях:

- **статические**: код из библиотеки добавляется в исполняемый файл на стадии линковки, и после её окончания файл библиотеки больше не нужен полученной программе
- **динамические**: код из библиотеки не добавляется в исполняемый файл, а загружается в память во время запуска программы. Таким образом, он должен быть доступен при каждом запуске

Библиотеки

Бесспорное преимущество динамических библиотек состоит в том, что если несколько программ используют одну библиотеку, то она загружается в память только один раз. Иными словами, сразу несколько программ могут (и будут) использовать один загруженный экземпляр библиотеки «одновременно». В то же время использование статической библиотеки заставит добавлять части её кода в каждый исполняемый файл по отдельности. Обновление динамической библиотеки потребует перезапуска использующих её программ, статической — их перелинковки (что обычно занимает немало времени).

Создание библиотек

Рассмотрим пример, состоящий из трех файлов:

```c
// main.c
#include <stdio.h>
#include "lib.h"

int main(){
    int a,b,c;
    scanf("%d%d%d",&a,&b,&c);
    printf("max3 =
%d\n",max3(a,b,c));
    return 0;
}
```

```c
//lib.h
int max(int a, int b);
int max3(int a, int b, int c);

//lib.c
#include "lib.h"

int max(int a, int b){
    return (a>b)?a:b;
}
int max3(int a, int b, int c) {
    return max(a,max(b,c));
}
```

Статическая библиотека

Такая библиотека создаётся из обычных объектных файлов, путём их архивации с помощью утилиты ar.

```
$ gcc -c lib.c
$ ar -r libmy1.a lib.o
ar: creating archive libmy1.a
```

Динамическая библиотека

Объектные файлы для динамической библиотеки компилируются иначе. Они должны содержать так называемый позиционно-независимый код (position independent code). Это позволяет библиотеке подключаться к программе, когда последняя загружается в память. Это связано с тем, что библиотека и программа не являются единой программой, а значит как угодно могут располагаться в памяти относительно друг друга. Компиляция объектных файлов для динамической библиотеки должна выполняться с опцией -fPIC. Динамическую создают при помощи опции -shared.

```
$ gcc -c -fPIC lib.c
$ gcc -shared -o libmy2.so lib.o
```

Использование библиотек Linux

Статическая библиотека подключается к основной программе так:

```
$ gcc -o prog1 main.c -L ./ -lmy
```

Динамическая библиотека подключается аналогично, но иногда необходимо явно указать линковщику абсолютный путь, по которому эта библиотека находится.

```
$ gcc -o prog2 main.c -L ./ -lmy2
-rpath /home/user/mylib2/
```

Опцию -L можно не указывать, если библиотека располагается в стандартных для данной системы каталогах для библиотек. Например, в GNU/Linux это /lib/ и /usr/lib/.

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

Использование библиотек Windows

Статическая библиотека подключается к основной программе так:

```
$ gcc -o prog1 main.c -L ./
c:\geekbrains.ru\Projects\C\MIPI_AC_Git\MIPI_AdvancedC\Lect3\35\l
ibmy1.a
```

Динамическая библиотека подключается аналогично, но иногда необходимо явно указать линковщику абсолютный путь, по которому эта библиотека находится.

```
$ gcc -o prog2 main.c -L ./
c:\geekbrains.ru\Projects\C\MIPI_AC_Git\MIPI_AdvancedC\Lect3\3
6\libmy2.so
```

Использование библиотек

Стоит также обратить внимание на размер исполняемых файлов и размер самих библиотек.

```
$ ls -l prog*
-rwxr-xr-x  1 staff  staff  49528 Oct 21 15:15 prog1
-rwxr-xr-x  1 staff  staff  49512 Oct 21 15:19 prog2

$ ls -l libmy*
-rw-r--r--  1 staff  staff    1024 Oct 21 15:07 libmy1.a
-rwxr-xr-x  1 staff  staff  16496 Oct 21 15:11 libmy2.so
```

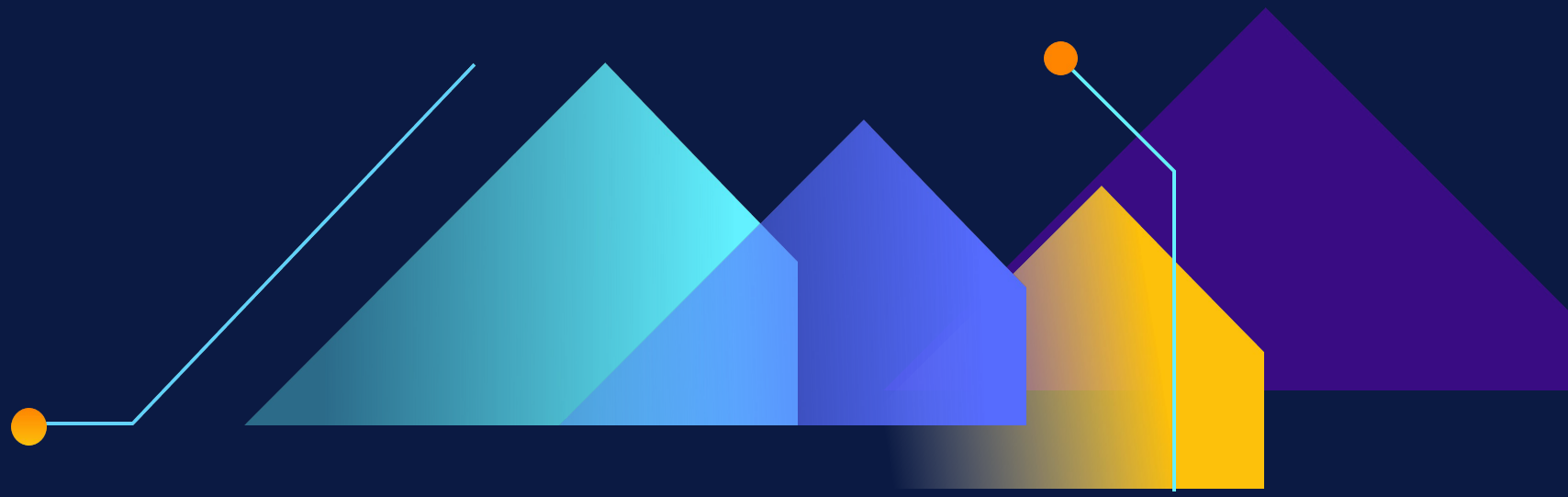
Размер динамическая библиотеки больше, чем статической, а вот исполняемый файл с подключенной динамической библиотекой наоборот меньше.

Сравнение статических и динамических библиотек

- Размер динамической библиотеки больше, чем статической, а вот исполняемый файл с подключенной динамической библиотекой наоборот меньше.
- Подразумевается, что динамическая библиотека — уже есть в системе, и при запуске вашей программы библиотеку не нужно копировать вместе с вашей программой — необходимая (или совместимая) версия библиотеки уже будет доступна в системе.
- Динамические библиотеки в большинстве случаев считаются лучшим подходом.

EASY EMBEDDED

Стандартный ввод/вывод



Стандартный ввод/вывод

При использовании стандартной функции ввода scanf, необходимо помнить, что спецификатор %s не контролирует размер введённых пользователем данных. Слишком длинная строка может привести к buffer overflow. В данном примере пользователь ввёл слишком длинную строку, которая записалась в s1 и затерла строку s2.

```c
#include <stdio.h>

char s1[] = "Hello ";
char s2[] = "world!";

int main(void) {
    scanf("%s",s1);
    printf("s1 = %s\n",s1);
    printf("s2 = %s\n",s2);
    return 0;
}
```

```
abcdefgh
s1 = abcdefgh
s2 = h
```

Стандартный ввод/вывод

Необходимо контролировать размер введенной строки.

```
scanf("%6s",s1);
printf("s1 = %s\n",s1);
printf("s2 = %s\n",s2);
```

```
abcdefgh
s1 = abcdef
s2 = world!
```

Стандартный ввод/вывод

```
char c;
printf("===Enter q to quit===\n");
do
{
    printf("Enter a symbol\n");
    scanf("%c", &c);
    printf("%c\n", c);
}
while (c != 'q');
```

```
===Enter q to quit===
Enter a symbol
a
a
Enter a symbol


Enter a symbol
b
b
Enter a symbol


Enter a symbol
q
q
```

Стандартный ввод/вывод

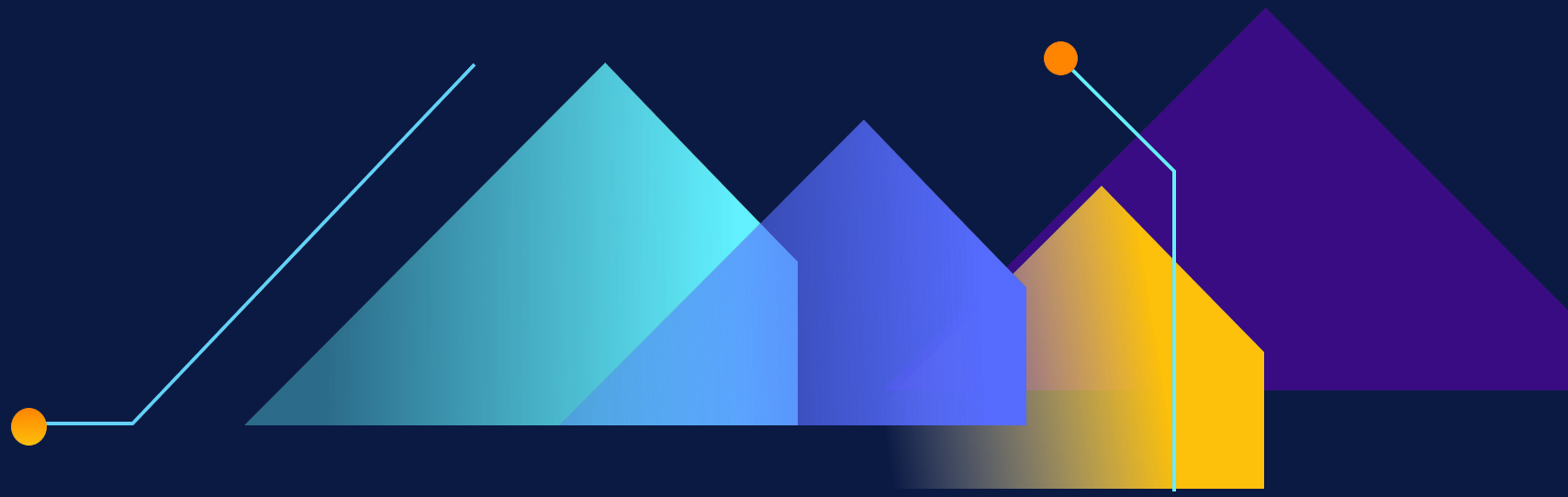
пробела перед спецификатором — “ %c”. Удобно для системы меню.

```
char c;
printf("===Enter q to quit===\n");
do
{
    printf("Enter a symbol\n");
    scanf(" %c", &c);
    // getchar();
    printf("%c\n", c);
}
while (c != 'q');
```

```
========Enter q to quit========
Enter a symbol
a
a
Enter a symbol
b
b
Enter a symbol
d
d
Enter a symbol
q
q
```

EASY EMBEDDED

Строковые функции из string.h



Строковые функции из string.h

Держитесь подальше от **strcpy**, **strcat**, **strncpy** и **strncat**, которые часто приводят к неэффективности и уязвимостям. В этом примере продемонстрирована ошибка **buffer overflow** для функции **strcpy**.

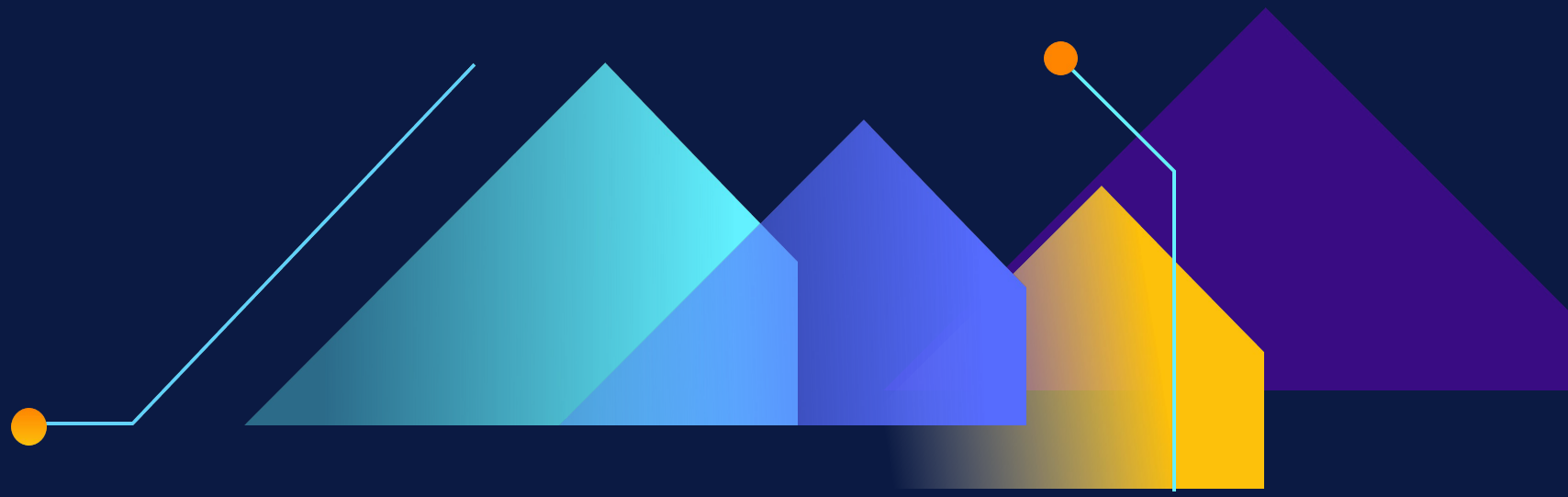
```c
#include <stdio.h>
#include <string.h>

int main() {
    char str1[] = "Small string.";
    char str2[15];
    strcpy(str2, str1);
    puts(str1);
    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
//ОШИБКА! Переполнение буфера
int main() {
    char str1[] = "This is very big string.";
    char str2[15];
    strcpy(str2, str1);
    puts(str1);
    return 0;
}
```

EASY EMBEDDED

Библиотека assert.h



Библиотека assert.h

assert — это macro библиотека для вывода диагностической информации в программе. Функция `void assert(int expression)` используется для проверки предположений, сделанных программистом. Мы можем использовать утверждение, чтобы проверить, является ли указатель, возвращаемый malloc (), NULL или нет. Рассмотрим пример вывод диагностического сообщения в stderr:

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

int main(){
  int x = 7;
  x = 9;
  assert(x==7); // Условие проверки
// будет напечатано сообщение в
stderr
  return 0;
}
```

```
Assertion failed: (x==7),
function main, file
main.c, line 13.
Abort trap: 6
```

Библиотека assert.h

Предположим, необходимо считать размер динамического массива. Пользователь может ошибиться, что приведёт к непредсказуемой работе программы. Необходимо проверить входные данные.

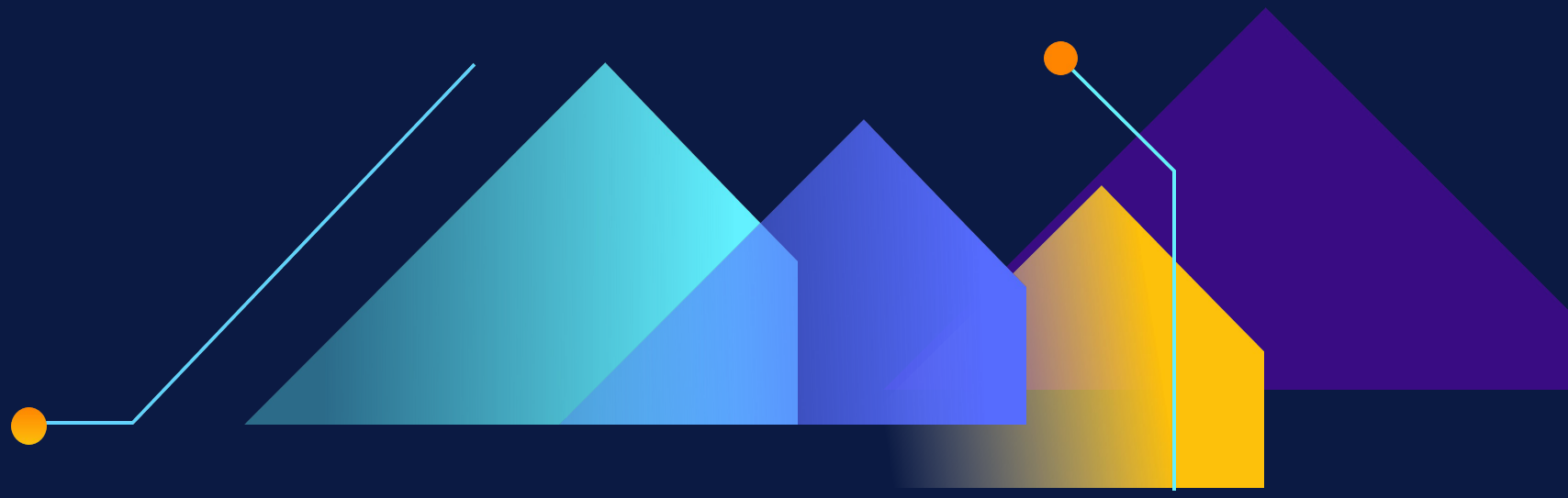
```c
int n;
printf("Input natural number: ");
scanf("%d", &n);
assert(n > 0); // Ожидаемое число
int arr[n];
```

```
Input natural number: -5
Assertion failed: (n > 0),
function main, file
main.c, line 7.
Abort trap: 6
```

Если бы такой защиты не было, то компилятор выделил память под массив размером: sizeof arr = 17179869164 (-5 в дополнительном коде).

EASY EMBEDDED

Печатаем по-русски



locale.h

locale.h — заголовочный файл стандартной библиотеки языка программирования C, который используется для задач, связанных с локализацией.

Функция `setlocale(int category, const char* locale)` устанавливает указанный системный языковой стандарт или его часть в качестве нового языкового стандарта C. Изменения остаются в силе и влияют на выполнение всех функций библиотеки C, зависящих от локали, до следующего вызова setlocale. Если locale является нулевым указателем, setlocale запрашивает текущий языковой стандарт C, не изменяя его.

Примеры

```c
#include <stdio.h>
#include <locale.h> //setlocale()
#include <inttypes.h>
#include <wchar.h> //«широкие» символы

int main(){
    setlocale(LC_ALL,"en_US.UTF-8");
    wchar_t str1[] = L"Привет";
    printf("str1 = %S\n", str1);
    printf("sizeof str1 = %lu\n", sizeof(str1));
    char str2[] = "Привет";
    printf("str2 = %s\n",str2);
    printf("sizeof str2 = %lu\n", sizeof(str2));
    char str3[] = "Hello!";
    printf("str3 = %s\n",str3);
    printf("sizeof str3 = %lu\n", sizeof(str3));
    return 0;
}
```

str1 = Привет
sizeof str1 = 28

str2 = Привет
sizeof str2 = 13

str3 = Hello!
sizeof str3 = 7

wchar_t символы

L — перед строкой означает, что строка состоит из wchar_t символов.

```
printf("%ls\n", L"Hello"); // Напечатать строку из  wchar_t
printf("%s\n", "Hello"); // Напечатать строку из  char
```

Функции wprintf

Функция **wprintf** аналогична функции **printf** с той лишь разницей, что в качестве первого аргумента(форматная строка) передаётся указатель на строку **wchar_t**.

```
wchar_t str[] = L"Привет";
wprintf(L"%ls\n",str);
```

Привет

UTF-8 и ASCII

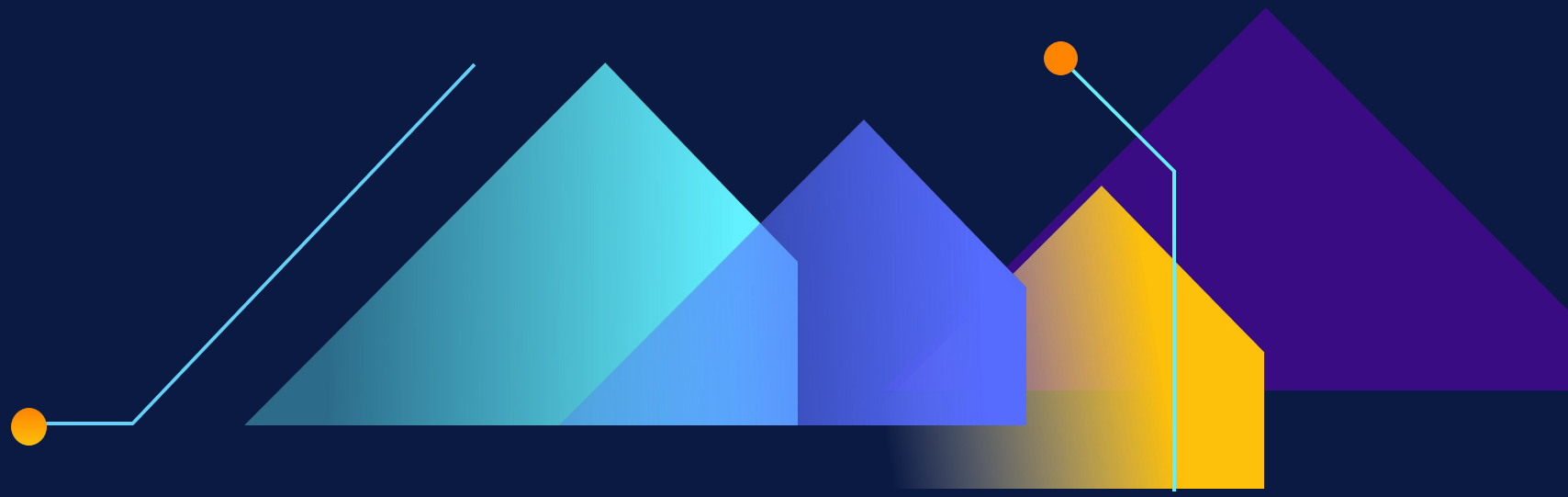
Каждый символ строки типа wchar_t занимает 4 байта и хранится в памяти в виде целого числа (код символа), в формате bigendina.

```
    setlocale(LC_ALL,"en_US.UTF-8");
    wchar_t str[] = L"БВ";
    printf("%ls\n",str);
    printf("sizeof(wchar_t) =
%lu\n",sizeof(wchar_t));
    printf("str[0] = %x\n", str[0]);
    printf("str[1] = %x\n", str[1]);
```

БВ
sizeof(wchar_t) = 4
str[0] = 411
str[1] = 412

EASY EMBEDDED

UTF-8, Unicode и ASCII



UCS и Unicode

UCS и Unicode — это кодовые таблицы, которые хранят коды символов — целые числа. Существует несколько альтернатив того, как последовательность таких символов или их соответствующих целочисленных значений может быть представлена в виде последовательности байтов. Две наиболее распространённые кодировки хранят текст Unicode в виде последовательностей из 2 или 4 байтов. Это UCS-2 и UCS-4 соответственно. Если не указано иное, в них первым идёт старший байт (bigendia формат).

Файл ASCII или Latin-1 можно преобразовать в файл UCS-2, просто вставив байт 0x00 перед каждым байтом ASCII. Если мы хотим получить файл UCS-4, то вместо этого надо вставить три байта 0x00 перед каждым байтом ASCII.

Использование под Unix

Использование UCS-2 (или UCS-4) под Unix привело бы к очень серьезным проблемам. Строки с этими кодировками могут содержать части байтов многих широких символов, таких как «\0» или «/», которые имеют особое значение в именах файлов и других параметрах функций библиотеки C.

Кроме того, большинство инструментов UNIX ожидают файлы ASCII и не могут читать 16-битные слова. По этим причинам UCS-2 не является подходящей кодировкой Unicode в именах файлов, текстовых файлах, переменных среды и т. д.

Кодировка UTF-8

Кодировка UTF-8 не имеет этих проблем. С помощью неё можно кодировать все 231 допустимых символов Unicode с использованием от одного до четырёх однобайтовых последовательностей.

Символы с более низкими числовыми значениями, которые, как правило, встречаются чаще, кодируются с использованием меньшего количества байтов.

UTF-8 разработан для обратной совместимости с ASCII:

первые 128 символов Unicode, которые взаимно-однозначно соответствуют ASCII, кодируются с использованием одного байта с тем же двоичным значением, что и ASCII.

Поэтому действительный текст ASCII является действительным UTF-8. — кодированным Unicode.

Таблица кодов кириллицы в Unicode, UTF-8 и Windows-1251

Символ	Unicode		UTF-8		Windows-1251
	16-ричн.	10-тичн.	16-ричн.	10-тичн.	
А	0410	1040	D090	208 144	192
Б	0411	1041	D091	208 145	193
В	0412	1042	D092	208 146	194
Г	0413	1043	D093	208 147	195
Д	0414	1044	D094	208 148	196
Е	0415	1045	D095	208 149	197

Пример

```
uint8_t ch[] = {0xd0,0x90,0xd0,0x91}; // "АБ"
uint32_t ch32[] = {0x91d090d0,0}; // "АБ"

printf("%x %x %x %x =
%s\n",ch[0],ch[1],ch[2],ch[3],ch);
printf("%x = %s\n",ch32[0], ch32);
```

d0 90 d0 91 = АБ
91d090d0 = АБ

Как декодировать UTF-8?

Чтобы декодировать UTF-8, нужно посмотреть первые (самые старшие) 2 бита каждого байта. Если они «01» или «00», — это 8-битный символьный код, если они «11», — это первый байт многобайтовой последовательности. Если они равны «10», то это один байт внутри многобайтовой последовательности. В следующем примере проверяем первый байт UTF-8 символа: он показывает, сколько всего байт в символе.

Как декодировать UTF-8?

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

int numberOfBytesInChar(unsigned
char val) {
    if (val < 128) {
        return 1;
    } else if (val < 224) {
        return 2;
    } else if (val < 240) {
        return 3;
    } else {
        return 4;
    }
}
```

```c
int utf8strlen(char *s) {
    char *tmp = s;
    int len = 0;
    while( *tmp ) {
        tmp += numberOfBytesInChar(*tmp);
        len++;
      // len += (*tmp++ & 0xC0) != 0x80;
// *tmp++ & 0xC0  проверяем шаблон
11xxxxxx
// !=0x80 не является 10xxxxxx
//              или внутренним байтом
UTF-8 символа
    }
    return len;
}
```

Как декодировать UTF-8?

```
int main(){
    char s[] = "Hello world";
    char s2[] = "Привет Мир";
    printf("strlen(s2) = %lu\n",strlen(s2));
    printf("utf8strlen(s) = %d\n",utf8strlen(s));
    printf("utf8strlen(s2) =
%d\n",utf8strlen(s2));
    return 0;
}
```

strlen(s) = 11
strlen(s2) = 19
utf8strlen(s) = 11
utf8strlen(s2) = 10

Задачи

По пройденному материалу

1. Написать программу, которая обрабатывает бинарный файл, состоящий из wchar_t символов и копирует всё содержимое данного файла в текстовый файл формата ASCII. Гарантируется, что все символы помещаются в 1 байт.

Задачи

По пройденному материалу

```
//Пример бинарного файла
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(){
    FILE *fin;
    wchar_t s[]=L"Hello";
    fin = fopen("input_wchar.txt","w");
    for(size_t i=0; s[i]; i++)
        fwrite(&s[i],sizeof(wchar_t),1,fin);
    fclose(fin);
    return 0;
}
```

Решение

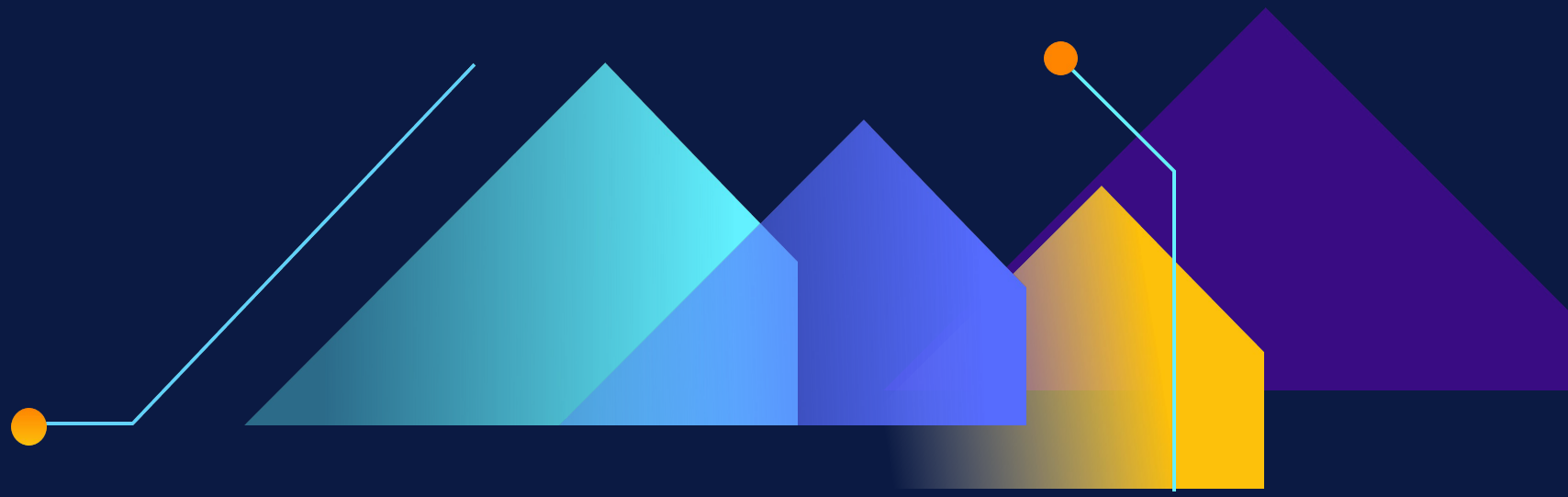
По пройденному материалу

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

int main(){
    FILE *fin;
    FILE *fout;
    int32_t character;
    wchar_t tmp;
    int8_t arr_out[100]={0};
    size_t i=0;
    fin =
fopen("input_wchar.txt","rb");
    while( fread(&tmp,
sizeof(wchar_t), 1, fin)==1 )
        arr_out[i++] = tmp;
    fclose(fin);
    fout =
fopen("output_ascii.txt","w");
    fprintf(fout,"%s",arr_out);
    fclose(fout);
    return 0;
}
```

EASY EMBEDDED

Библиотека time.h



Работаем со временем

Заголовочный файл **time.h** определяет функции для работы с датой и временем. В частности, функция time возвращает текущие дату и время в виде объекта типа **time_t** и имеет следующий прототип:

```
time_t mytime = time(NULL);
```

Чтобы собственно получить дату/время и ее компоненты (часы, минуты и т.д.), нам надо получить из объекта **time_t** структуру tm с помощью функции **localtime()**

```
time_t mytime = time(NULL);
struct tm *now = localtime(&mytime);
```

Структура tm хранит данные в ряде своих элементов, каждый из которых представляет тип **int**.

Работаем со временем time.h

Для конвертации между различными форматами времени и даты используется библиотека time.h. Библиотека, содержит типы и функции для работы с датой и временем.

```c
#include <stdio.h>
#include <time.h>
int main(void)
{
time_t mytime = time(NULL);
struct tm *now = localtime(&mytime);
  printf("Date: %d.%d.%d\n", now->tm_mday,
     now->tm_mon + 1, now->tm_year + 1900);
  printf("Time: %d:%d:%d\n", now->tm_hour,
     now->tm_min, now->tm_sec);
  return 0;
}
```

```
Date: 14.10.2023
Time: 23:59:56
```

Работаем со временем меньше одной секунды

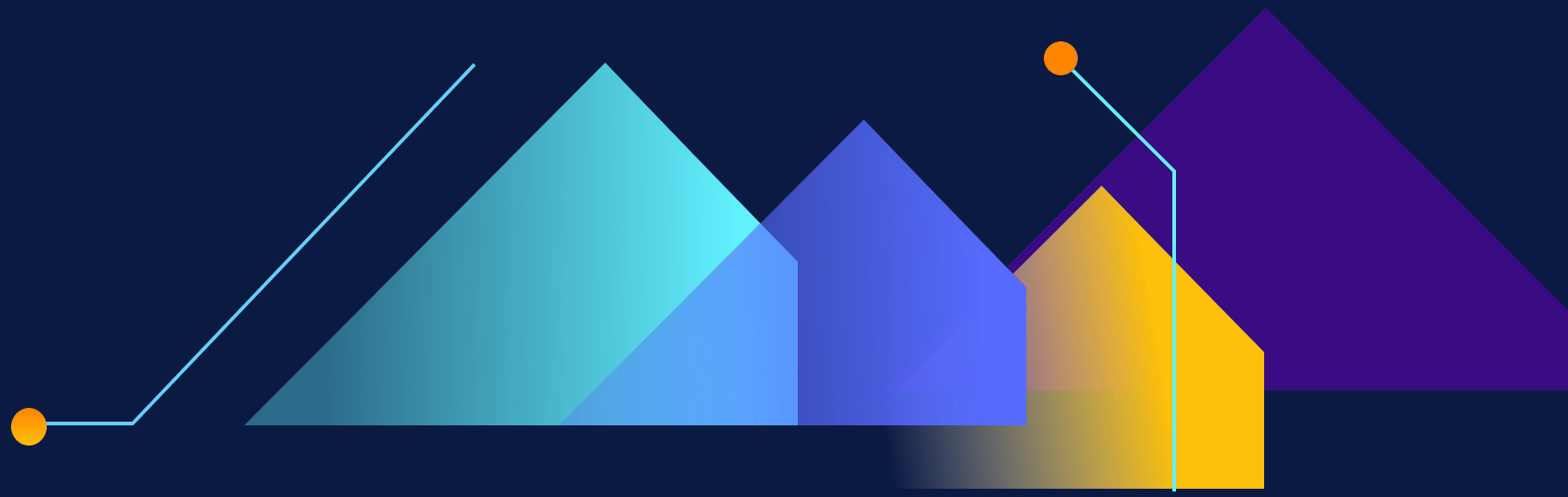
Если нужна точность выше чем 1 секунда:

```
#include <stdio.h>
#include <time.h>
double DELAY = 3;
int main()
{
    clock_t begin = clock();
    while((double)(clock() -
begin)/CLOCKS_PER_SEC<DELAY)
    {/*printf("%4d\n",clock());*/ }
    printf("Hello World %ld",CLOCKS_PER_SEC);
    return 0;
}
```

```
В Windows:
Hello World 1000
В GDBOnline:
Hello World 1000000
```

EASY EMBEDDED

Библиотека ncurses.h



Библиотека ncurses.h

`ncurses.h` — это библиотека функций, которая управляет отображением приложения на терминалах с символьными ячейками. Данная библиотека образует оболочку для работы с необработанными кодами терминала и предоставляет очень гибкий и эффективный API (интерфейс прикладного программирования). Она предоставляет функции для перемещения курсора, создания окон, создания цветов, игры с мышью и т. д. Прикладным программам больше не нужно беспокоиться о базовых возможностях терминала.

Пример работы функций

Библиотека не только создаёт оболочку для использования терминала, но и предоставляет надежную структуру для создания красивого пользовательского интерфейса в текстовом режиме. Она предоставляет функции для создания окон, панелей, меню и т. д. Можно создавать приложения, содержащие несколько окон, меню, панелей и форм. Все окна могут управляться независимо, можно реализовать «возможность прокрутки» и даже минимизации окна.

```
#include <ncurses.h>

int main(){
    int ch;
    initscr();  // Начать curses mode
    raw();      // Отключаем buffering
    noecho(); // Отключаем echo()
режим пока считываем символы getch
    printw("Type text: \n");
    while( (ch = getch()) != '.'){
        printw("%c", ch);
    }
    //refresh(); // Печатаем это на
экран
    getch();   // Ждем пока
пользователь нажмет клавишу
    endwin();  // Завершить curses
mode
    return 0;
}
```

Обратите внимание

Внимание! Для компиляции данного примера необходимо добавить ключ -lncurses.

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

Внимание! Для компиляции в Windows данного примера необходимо добавить путь до библиотеки

```
gcc.exe -Wall -std=c99 -g -ID:\geekbrains.ru\Projects\C\PDCurses-master -c main.c -o main.o

gcc.exe                                -o              pdcurses_test2.exe              main.o
D:\geekbrains.ru\Projects\C\PDCurses-master\wincon\pdcurses.a
-ID:\geekbrains.ru\Projects\C\PDCurses-master
```

или https://packages.msys2.org/package/ncurses?repo=msys&variant=x86_64

https://stackoverflow.com/questions/75556484/how-to-install-ncurses-on-windows

Обратите внимание

Обычно терминал использует буферный ввод.

Вводимые пользователем символы считываются до тех пор, пока не встретится новая строка или возврат каретки.

Иногда необходимо, чтобы символы были доступны с момента, как только пользователь их наберёт.

Функция **raw** отключает буферный ввод.

Даже управляющие символы CTRL+Z и CTRL+C будут заблокированы и переданы программе без генерации сигнала прерывания.

Аналогичная функция **cbreak** пропустит сигналы прерывания (CTRL+C и CTRL+Z) и передаст их операционной системе.

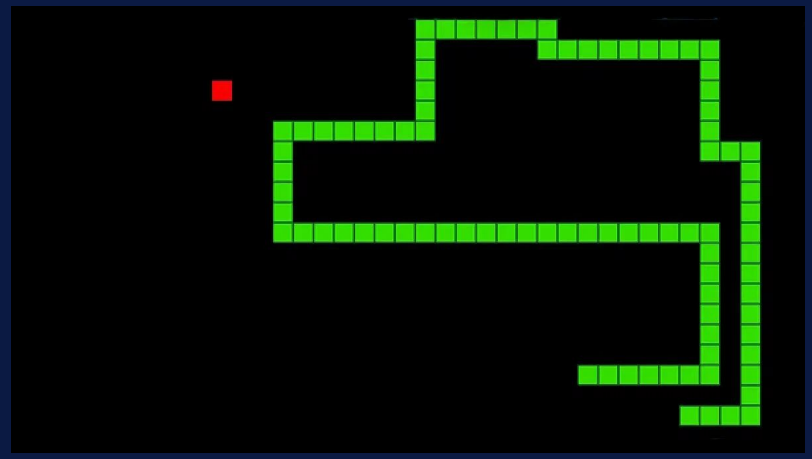
Пример работы функций

Еще одна функция **noecho** управляет отображением символов, вводимых пользователем на терминале. Она отключает «эхо» для того, чтобы получить больший над ним контроль или подавить ненужное «эхо» при вводе с терминала от пользователя через функцию `getch()`. Обычно её вызывают при инициализации и контроля отображения символов. Это даёт возможность отображать символы в любом месте окна без обновления текущих координат (y, x). В следующем примере мы рассмотрим печать строки на экране терминала:

```
char string[]="Hello world!";
int x=0,y=10;
printw(string); /* Напечатать на стандартном экране stdscr */
                /* в текущей позиции курсора  */
mvprintw(y, x, string);/* Напечатать в позиции (y, x)  */
```

EASY EMBEDDED

Игра змейка



Формат данных: голова и хвост

Рассмотрим более сложный пример, который продемонстрирует работу с библиотекой ncurses и не только. Реализуем игру Змейка в консоли терминала. В начале рассмотрим логику игры, и как она устроена с точки зрения архитектуры.

```
/* Голова змейки содержит в себе:
   x,y - координаты текущей позиции
   direction - направление движения
   tsize - размер хвоста
   *tail -  ссылка на хвост   */
struct snake {
    int x;
    int y;
    int direction;
    size_t tsize;
    struct tail *tail;
} snake;
```

```
/*
 Хвост это массив состоящий из
координат x,y
 */
struct tail {
    int x;
    int y;
} tail[MAX_TAIL_SIZE];
```

Формат данных: еда

```
/*  Еда массив точек
  x, y - координата где установлена точка
  put_time - время когда данная точка была установлена
  point - внешний вид точки ('$','E'...)
  enable - была ли точка съедена
 */
struct food {
    int x;
    int y;
    time_t put_time;
    char point;
    uint8_t enable;
} food[MAX_FOOD_SIZE];
```

Инициализирующие константы

```
enum {LEFT=1, UP, RIGHT, DOWN, STOP_GAME='q'};
enum {MAX_TAIL_SIZE=100, START_TAIL_SIZE=3,
MAX_FOOD_SIZE=20, FOOD_EXPIRE_SECONDS=10};
```

Логика игры

Инициализация. В самом начале программы происходит установка начальных значений и выделение памяти:

- snake — голова
- tail[] — хвост
- food[] — еда

За это отвечают функции: `initHead(struct snake *head)`, `initFood(struct food f[], size_t size)`, `init(struct snake *head, struct tail *tail, size_t size)`.

Голова змейки движется в соответствии с заданным направлением. Через промежуток времени `timeout(SPEED)` происходит отрисовка новой позиции головы с учётом текущего направления. Например, если направление задано как RIGHT, то это соответствует прибавлению 1 к текущей координате x (snake.x++). За движение головы отвечает функция `go(struct snake *head)`.

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.

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. Доработайте функционал игры змейка. Реализуйте в игре ситуацию, когда змея врезается сама в себя.
2. Выход за границы экрана

https://github.com/Sudar1977/MIPI_AdvancedC/blob/main/Lect3/snake_seminar_2.c

Дедлайн: конец курса

Советуем регулярно выполнять ДЗ (наверстать пропуски тяжело)