Лекция 6

Строки в Си

- Null-terminated strings – в конце строки находится байт 0 (или '\0') – признак конца строки
- Строковые литералы “abcd” содержат “невидимый” \0 в конце
  - char s[] = “abcd”; // sizeof(s) == 5
- Если под строковый литерал память явно не выделяется, он размещаются в read-only памяти
  - char *s = “abcd”; // sizeof(s) = sizeof(void*)
    s[2] = 'd'; // undefined behavior

Pros & contras

- (+) для работы со строкой достаточно одного указателя
- (+) сдвигая указатель по строке вперед все равно получаем строку
- (-) получение длины строки (strlen) выполняется за линейное время
  - **НИКОГДА!**
    for (int i = 0; i < strlen(s); ++i) {…}
- (-) нельзя использовать \0 в строке

Альтернативы

- Хранить пару <указатель, длина> (std::string)
  - (+) нет проблемы байта \0
  - (-) размер такой структуры в два раза больше (а размер самой строки на один байт меньше)
- Хранить длину в начале строки (pascal style)
  - Либо ограниченный размер (если длина – 1 байт), либо неэффективное использование памяти (4 байта длины для коротких строк - много)

Управление памятью

- В Си практически все управление памятью возложено на программиста
- При работе с указателями важно понимать, как и где выделена память, на которую он указывает:
  - Глобальная/статическая память
  - Thread-local storage
  - Автоматическая память
  - Динамическая память (куча)

Буфер строки

- Буфер – область памяти, отведенная для хранения строки
- Буфер имеет ограниченный размер, но размер может изменяться
- При обработке строки “на чтение” достаточно только указателя на строку
- При формировании строки в памяти важен и адрес буфера, и размер буфера

Переполнение буфера

- Если не контролируется размер данных, записываемых в буфер, возможно **переполнение буфера**
- Может иметь катастрофические последствия для безопасности системы (arbitrary code execution)

Переполнение буфера

- Если не контролируется размер данных, записываемых в буфер, возможно **переполнение буфера**
- Может иметь катастрофические последствия для безопасности системы (arbitrary code execution)
- CVE-2015-2712 (Firefox)
- CVE-2010-1117 (IE)
- CVE-2016-5157 (Chrome)

Good vs evil

- “Плохие” функции: записывают строку, но не принимают параметр размера буфера: gets, scanf(“%s”, …), strcpy, sprintf
  - gets, scanf – запрещены; strcpy, sprintf – крайне осторожно
- “Хорошие” функции: записывают строку и принимают размер буфера строки: fgets, snprintf, scanf(“%100s”, ...)

Кодировки текста

- Исторически: ASCII – символы с кодами 0-127; достаточно для английского языка
- Недостаточно для других языков
  - Использование “верхней” части байта, коды 128-255: (iso8859-X, cpYYYY, koi8-r, …)
  - Специальные символы-переключатели состояния (JIS, EUC-JP – японский язык)
- Обмен текстовыми документами сложен

Unicode

- Определяет 1,114,112 кодовых позиций (Code Points) (обозначаются U+0 … U+10FFFF)
- U+D800 – U+DFFF – недопустимы в корректном Unicode (UCS4, UTF8)
- Кодовые позиции содержат глифы всех известных письменностей, диакритические знаки
- U+0 – U+7F совпадает с ASCII
- Возможны разные кодировки (битовое представление для Code Points)

Кодировки Unicode

- UCS-4 (один CodePoint – 32-битный int)
  - (+) фиксированный размер – удобно обрабатывать
  - (-) 4 байта на все codepoints
  - (-) много байтов \0 в тексте – несовместим с ASCII
- UCS-2 (один CodePoint – uint16_t) – только для U+0 – U+FFFF
- UTF-16 (один CodePoint – один или два uint16_t)

UTF-8

- Байтовый поток
- Один CodePoint кодируется от 1 до 4 байт
- U+0 – U+7f кодируются 1 байтом (совместимость с ASCII)
- Байт \0 всегда обозначает U+0 и может использоваться как терминатор строки – совместимость с Си-строками
- По любому месту в потоке можно найти начало кодировки соответствующего CodePoint

UTF-8

- Кодирование Code points в UTF-8
- Overlong encoding (длина последовательности больше минимальной, например 0xC0 0xAF → '/') запрещен

UTF-8 (2003)

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Поддержка в C/C++

- wchar_t тип данных для хранения unicode codepoints во внутреннем представлении (unsigned short – Windows, int или long – Unix)
- В Unix внутреннее представление – UCS4
- Wide-char literals: L'a'
- Wide-char string literals: L”Привет”
- Функции: getwc, fgetws, wscanf, wprintf, wcslen, ...

Локаль (Locale)

- Определяет кодировку в системе, региональные особенности, язык взаимодействия с пользователем
- Переменные окружения LANG и LC_*
- LANG=en_US.utf8 – язык/регион – американский английский, кодировка UTF8
- LANG=ru_RU.UTF-8 – русский/Россия, UTF8
- LANG=C – по умолчанию ASCII

Setlocale

- setlocale позволяет установить локаль для выполняющейся программы
- По умолчанию – C, не позволяет обрабатывать символы вне ASCII
- Для установки системной локали: setlocale(LC_ALL, “”);

Выравнивание

- Выравнивание — гарантирует размещение переменной (простого или сложного типа) так, чтобы адрес размещения был кратен размеру выравнивания
- Дополнение — добавление в структуру скрытых полей так, чтобы поля структуры были правильно выровнены

Невыровненные данные

- Недопустимы на некоторых платформах (попытка обращения вызовет Bus Error)
- На других платформах (x86) обращение к невыровненным данным требует два цикла обращения к памяти вместо одного
- Работа с невыровненными данными **не атомарна**

Правильное выравнивание

- Тип char не требует выравнивания
- Short — выравнивание по двум байтам
- Int, long (x86), long long (x86), double (x86) — выравнивание по 4 байтам
- Long (x64), long long (x64), double (x64) — выравнивание по 8 байтам
- Выравнивание по границе 16 байтов — для стека в Linux x86
- Выравнивание по 64 байтам – для cache line
- Выравнивание по границе 4096 — размер страницы (mmap)

Базовые типы и их свойства

type	X86 Linux		X64 Linux	
	size	alignment	size	alignment
`char`	1	1	1	1
`short`	2	2	2	2
`int`	4	4	4	4
`long`	4	4	8	8
`long long`	8	4	8	8
`void *`	4	4	8	8
`float`	4	4	4	4
`double`	8	4	8	8
`long double`	12	4	16	16

Пример:

```
struct s {
    char f1;
    long long f2;
    char f3;
};
```

- X86: sizeof(s) == 16
- X64: sizeof(s) == 24

```
struct s {
    long long f2;
    char f1;
    char f3;
};
```

- X86: sizeof(s) == 12
- X64: sizeof(s) == 16

Пример для x64

```
struct s {
    char f1;      // смещение от начала - 0
    // + 7 байт на выравнивание (alignment)
    long long f2; // смещение от начала - 8
    char f3;      // смещение от начала – 9
    // + 7 байт на дополнение (padding)
};
```

- Максимальное требуемое выравнивание – 8 (для поля f2), поэтому:
  - struct s требует выравнивания 8
  - sizeof(struct s) должен быть кратен 8
- Смещение первого поля всегда равно 0
- Смещение каждого поля должно быть выровнено соответственно (быть кратным выраваниванию) типу этого поля

Динамическая память

- Область динамической памяти заданного размера нужно выделять явно
- Получаем указатель на начало области
- В динамической памяти могут размещаться и массивы элементов, и одиночные элементы
- Динамическая память должна освобождаться явно

Динамическая память

- Выделение:
  void *malloc(size_t size);
  void *calloc(size_t nelem, size_t elsize);
- Освобождение:
  void free(void *ptr);
- Изменение размера:
  void *realloc(void *ptr, size_t newsize);

Блоки динамической памяти

- Адрес, возвращенный malloc, должен быть выровнен корректно выровнен, то есть кратен 4 для x86 и кратен 16 для x64
- malloc выделяет память блоками чуть большего размера, чтобы обеспечить выравнивание (x86: 12, 20, 28… + 4 байта на служебный указатель; x64: 24, 40, 56 + 8 байт на служебный указатель) – для glibc