

Лекция 5

Текстовые данные

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, cpYYY, koī8-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, "");`

Размещение данных в памяти

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**
- **UNDEFINED BEHAVIOR!**

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 выравниванию) типу этого поля

Объединения (union)

- В типе union все поля размещаются по одному и тому же смещению (0) от начала структуры
- Размер union – $\max(\text{fields_size})$
- Выравнивание union – $\max(\text{fields_align})$

```
union Float
{
    float fval;
    unsigned uval;
    unsigned char[4];
};
```

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`
- Оператор `new` (C++) работает поверх `malloc`

Small string optimization

- Короткие строки храним в самой структуре

```
enum { STRING_OPT_SIZE = 8 };  
struct String  
{  
    size_t size;  
    union  
    {  
        char *str;  
        char data[STRING_OPT_SIZE];  
    };  
};
```

Функция realloc

- Определена в `<stdlib.h>`
`void *realloc(void *ptr, size_t newsize);`
- Изменяет размер ранее выделенного блока `ptr`, возвращает адрес нового местоположения
- `ptr` может остаться на своем месте, но может быть и перемещен
- Если `ptr` перемещен, `ptr` разыменовывать нельзя
- Если не хватает памяти, возвращается `NULL`, `ptr` остается сохранным
- Если `ptr == NULL`, работает как `malloc`
- Если `newsize == 0`, работает как `free`

Vector implementation

- reserved – сколько памяти выделено
- size – сколько памяти используется
- data – данные
- При полном использовании выделенной памяти она расширяется в C раз с помощью realloc: $C = 2$ или $C = 3/2$ или другое

Vector vs List

- Вектор
 - (+) расположен в памяти последовательно
 - (+) оптимальнее использует кучу
 - (?) вставка и удаление из середины за $O(n)$
-
- Список предпочтительнее при больших размерах одного элемента и добавлении/удалении из середины
- По умолчанию следует использовать вектор
- <https://isocpp.org/blog/2014/06/stroustrup-lists>