Отладчик **gdb**

Чтобы использовать отладчик нужно использовать ключ **-g** при компиляции. Если не добавлять отладочную информацию при компиляции и запустить утилиту **gdb**, то она запустится, но она сообщит об отсутствии отладочных символов

Компиляция с отладочной информацией:

```
gcc main.c -std=c99 -Wall -Werror -g -o main.exe
```

Результат работы **gdb** при отсутствии отладочной информации:

```
Reading symbols from main.exe...
(No debugging symbols found in main.exe)
```

Чтобы запустить программу под отладчиком нужно использовать команду **run**

Для досрочного завершение программы нужно использовать команду **kill**

Чтобы посмотреть в каком месте программы была совершена остановка, можно использовать команду **where**

Значение переменной можно посмотреть с помощью команды **print <var>**

Значение переменной можно поменять с помощью команды **set <var>=<val>**

Программу можно выполнять пошагово с помощью команд:

`Step`	`next`

step, в отличии от **next**, заходит в функции

Чтобы посмотреть последовательность вызванных функций используется команда **backtrace** или сокращенно **bt**

Точку останова можно установить с помощью команды **break**

`break [file_nme:]<line_num>`	Остановка на определенной строке
`break <funct>`	Остановка на начале указанной функции
`break if <condition>`	Остановка при определенном условии

Временная точка останова - действует только один раз, при следующем выполнении инструкции в месте ее установки. После того как программа достигнет этой точки останова и остановится, она автоматически удалится из списка точек останова

Временные точки останова могут быть установлены с помощью команды **tbreak**

Чтобы временно включить/выключить точку останова используют команды **enable/disable** соответственно

Для пропуска некоторого количества срабатываний точки останова используют команду **ignore <count> <num of breakpoint>**

Условие остановки на точке останова можно задать с помощью команды

break <position> if <condition>

Точки останова используются для остановки программы, а точки наблюдения - для отслеживания изменений в значениях переменных. В **gdb** для создания точки наблюдения используется команда **whatch <var>**.

Точку наблюдения удобно использовать в цикле, в котором поитерационно нужно отслеживать значения переменных

Для просмотра содержимого области памяти в отладчике gdb используется команда **x[/nfu] [address]**

- n – сколько единиц памяти должно быть выведено
- f – спецификатор формата
- u – размер выводимой единицы памяти

Переменные

Существуют переменные **char_var, int_var, uns_var, ll_var** соответствующие типам **char, int, unsigned, long long**.

Представление в памяти при присвоенном положительном значении(7):

```
(gdb) info locals
char_var = 7 '\a'
int_var = 7
uns_var = 7
ll_var = 7
```

```
(gdb) x /1xb &char_var
0x7fffffffdd5f:     0x07
(gdb) x /4xb &int_var
0x7fffffffdd60:     0x07 0x00 0x00 0x00
(gdb) x /4xb &uns_var
0x7fffffffdd64:     0x07 0x00 0x00 0x00
(gdb) x /8xb &ll_var
0x7fffffffdd68:     0x07 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Представление в памяти при присвоенном отрицательном значении(-7):

```
(gdb) info locals
char_var = -7 '\371'
int_var = -7
uns_var = 4294967289
ll_var = -7
```

```
(gdb) x /1xb &char_var
```

```
0x7fffffffdd5f:     0xf9
(gdb) x /4xb &int_var
0x7fffffffdd60:     0xf9 0xff 0xff 0xff
(gdb) x /4xb &uns_var
0x7fffffffdd64:     0xf9 0xff 0xff 0xff
(gdb) x /8xb &ll_var
0x7fffffffdd68:     0xf9 0xff 0xff 0xff 0xff 0xff 0xff 0xff
```

Числа представляются в памяти с помощью **Little Endian**. При отрицании в памяти хранится дополнительный код(биты инвертируются и прибавляется младший бит(1)). **unsigned**, тк. беззнаковый тип, хранит в себе обратный код(число = 2^32 - <num>/ инвертированные биты положительного значения).

Одномерный массив

Программа:

```c
#include <stdio.h>

int main(void)
{
    int arr[] = {1, 2, 3, 4, 5};
    int *point = arr;
    return 0;
}
```

Хранение массива в памяти

```
(gdb) x /20xb arr
0x7fffffffdd50:     0x01 0x00 0x00 0x00 0x02 0x00 0x00 0x00
0x7fffffffdd58:     0x03 0x00 0x00 0x00 0x04 0x00 0x00 0x00
0x7fffffffdd60:     0x05 0x00 0x00 0x00
```

Все элементы массива хранятся в памяти в заданном порядке

Особенности адресной арифметики:

```
(gdb) print point //Изначальный адрес
$1 = (int *) 0x7fffffffdd50
(gdb) print *point //Значение в заданной ячейке памяти
$2 = 1
(gdb) print *(point + 1) //Значение следующей ячейки памяти
$3 = 2
(gdb) print ++point
$4 = (int *) 0x7fffffffdd54 //Следующий адрес
(gdb) print --point
$5 = (int *) 0x7fffffffdd50 //Предыдущий адрес
```

Вывод: при сложении адреса с числом мы получаем адрес равный изначальному адресу с добавленным размером типа, который изначальный адрес хранит.

В Примере: new_point = ++point = point + sizeof(int) = point + 4, где point = (int *) 0x7fffffffdd50 и new_point = (int *) 0x7fffffffdd54

Многомерные массивы

Пусть array[2][3][4] – трехмерный массив целых чисел, тогда array – массив состоящий из 2-ух измерений, содержащие матрицы 3x4 (массив, состоящий из 2-ух элементов, которые являются массивами из 3-ех элементов, которые в свою очередь состоят из 4-ех целочисленных значений).

<table>
<tr><td>Adress</td><td>0x00</td><td>0x04</td><td>0x08</td><td>0x0c</td><td>0x10</td><td>0x14</td><td>0x18</td><td>0x1c</td><td>0x20</td><td>0x24</td><td>0x28</td><td>0x2c</td><td rowspan="3">…</td></tr>
<tr><td>a[][][z]</td><td>a[0][0][0]</td><td>a[0][0][1]</td><td>a[0][0][2]</td><td>a[0][0][3]</td><td>a[0][1][0]</td><td>a[0][1][1]</td><td>a[0][1][2]</td><td>a[0][1][3]</td><td>a[0][2][0]</td><td>a[0][2][1]</td><td>a[0][2][2]</td><td>a[0][2][3]</td></tr>
<tr><td>a[][y][]</td><td colspan="4">a[0][0]</td><td colspan="4">a[0][1]</td><td colspan="4">a[0][2]</td></tr>
<tr><td>a[x][][]</td><td colspan="12">a[0]</td><td>a[1]</td></tr>
</table>

- x – первое измерение
- y – второе измерение
- z – третье измерение

Массив array:

```
int arr[2][3][4] = {
        {
                {1, 2, 3, 4},
                {5, 6, 7, 8},
                {9, 10, 11, 12}
        },
        {
                {10, 20, 30, 40},
                {50, 60, 70, 80},
                {90, 100, 110, 120}
        }
}
```

Дамп массива array:

```
0x6f557ff8b0:   0x00000001 0x00000002 0x00000003 0x00000004
0x6f557ff8c0:   0x00000005 0x00000006 0x00000007 0x00000008
0x6f557ff8d0:   0x00000009 0x0000000a 0x0000000b 0x0000000c
0x6f557ff8e0:   0x0000000c 0x0000000b 0x0000000a 0x00000009
0x6f557ff8f0:   0x00000008 0x00000007 0x00000006 0x00000005
0x6f557ff900:   0x00000004 0x00000003 0x00000002 0x00000001
```

Дамп получаем с помощью команды в отладчике **gdb**:

```
(gdb) print sizeof(arr) / sizeof(int)
$1 = 24
(gdb) x /24xw &arr
```

1. Получаем кол-во элементов первого измерения:

```
(gdb) print sizeof(arr) / sizeof(arr[0])
$2 = 2
```

Содержимое этих элементов:

```
(gdb) x/12xw arr[0]
0x6f557ff8b0:  0x00000001     0x00000002     0x00000003     0x00000004
0x6f557ff8c0:  0x00000005     0x00000006     0x00000007     0x00000008
0x6f557ff8d0:  0x00000009     0x0000000a     0x0000000b     0x0000000c
------------------------------------------------------------------------
(gdb) x/12xw arr[1]
0x6f557ff8e0:  0x0000000c     0x0000000b     0x0000000a     0x00000009
0x6f557ff8f0:  0x00000008     0x00000007     0x00000006     0x00000005
0x6f557ff900:  0x00000004     0x00000003     0x00000002     0x00000001
```

2. Получаем кол-во элементов второго измерения:

```
(gdb) print sizeof(arr[0])/sizeof(arr[0][0])
$3 = 3
```

Содержимое этих элементов:

```
(gdb) x/4xw arr[0][0]
0x6f557ff8b0:  0x00000001     0x00000002     0x00000003     0x00000004
------------------------------------------------------------------------
. . .
------------------------------------------------------------------------
(gdb) x/4xw arr[1][2]
0x6f557ff900:  0x00000004     0x00000003     0x00000002     0x00000001
```

3. Получаем кол-во элементов третьего измерения:

```
(gdb) print sizeof(arr[0][0])/sizeof(arr[0][0][0])
$3 = 4
```

Содержимое этих элементов:

```
(gdb) x/1xw &arr[0][0][0]
0x6f557ff8b0:  0x00000001
------------------------------------------------------------------------
. . .
------------------------------------------------------------------------
(gdb) x/1xw &arr[1][2][3]
0x6f557ff900:  0x00000001
```

Массив:

```
int arr[2][3][4];
```

arr – массив из двух элементов типа “int [3][4]”

```
int (*x)[3][4] = arr;
```

arr[i] – массив из трех элементов типа “int [4]” (i ∈[0, 1])

```
int (*y)[4] = arr[i];
```

arr[i][j] – массив из пяти элементов типа “int” (i ∈[0, 1], j ∈ [0, 1, 2])

```
int *z = arr[i][j];
```

arr[i][j][k] – элемент типа “int” (i ∈ [0, 1], j ∈ [0, 1, 2],k ∈ [0, 1, 2, 3])

```
int el = arr[i][j][k];
```

Размеры элементов:

Теооретический расчет	Отладчик gdb
`sizeof(*x) = 3 * 4 * 4 = 48`	`(gdb) print sizeof(*x)` `$1 = 48`
`sizeof(*y) = 4 * 4 = 16`	`(gdb) print sizeof(*y)` `$2 = 16`
`sizeof(*z) = 4`	`(gdb) print sizeof(*z)` `$3 = 4`
`Sizeof(el) = 4`	`(gdb) print sizeof(el)` `$4 = 4`

Способы передачи

Заголовки функций для обработки:

1. Трехмерного массива:

`int process(int arr[X][Y][Z]);`	`int process(int arr[][Y][Z]);`

2. Двумерного массива:

`int process(int arr[X][Y]);`	`int process(int arr[][Y]);`

3. Одномерного массива:

`int process(int *arr);`	`int process(int arr[X]);`	`int process(int arr[]);`

Строки

Строка

Строка:

```
char string[] = "Hello";
```

Дамп памяти строки:

```
(gdb) print sizeof(string) //Получаем длину строки с учетом "\0"
$1 = 6
(gdb) x/6xb &string
0xd9c45ffa6a:   0x48    0x65    0x6c    0x6c    0x6f    0x00
```

Первые 5 байт содержат символы строки (т.е. их числовые коды ASCII), а последний байт равен 0x00, что указывает на конец строки.

Массивы строк

двумерный массив строк	массив указателей на строки
`char arr[][6] = {"Red", "Green", "Blue"};`	`char *rag_arr[] = {"Red", "Green", "Blue"};`

Дамп памяти

двумерный массив строк	массив указателей на строки
(gdb) print (arr) $1 = {"Red\000\000", "Green", "Blue\000"}	(gdb) print rag_arr $2 = {0x7ff6c33c4000 "Red", 0x7ff6c33c4004 "Green", 0x7ff6c33c400a "Blue"}
(gdb) print sizeof(arr) $3 = 18 (gdb) x/18xb &arr 0x48e51ffb00: 0x52 0x65 0x64 0x00 0x00 0x00 0x47 0x72 0x48e51ffb08: 0x65 0x65 0x6e 0x00 0x42 0x6c 0x75 0x65 0x48e51ffb10: 0x00 0x00	(gdb) x /15xb *rag_arr 0x7ff6c33c4000: 0x52 0x65 0x64 0x00 0x47 0x72 0x65 0x65 0x7ff6c33c4008: 0x6e 0x00 0x42 0x6c 0x75 0x65 0x00

Вывод:

- Двумерный массив строк

arr[0]	R	e	d	\0	\0	\0
arr[1]	G	r	e	e	n	\0
arr[2]	B	l	u	e	\0	\0

Массив хранит строку(символы строки и детерминирующий ноль) и дополняет ее тернарными нулями до заданного размера(в случае примера до 6).

- Массив указателей на строки

rag_arr[0]	R	e	d	\0		
rag_arr[1]	G	r	e	e	n	\0
rag_arr[2]	B	l	u	e	\0	

Массив хранит адреса строк

В обоих случаях байты хранят в себе значение символа(соответствующее значение ASCII)

	двумерный массив строк	**массив указателей на строки**
Размер памяти	3 * 6 = 18 байт	24 + (4 + 6 + 5) = 39 байт
Размер полезных данных	15 байт– количество байт выделенное по строки	15 байт – количество байт выделенное по строки
Размер вспомогательных данных	3 байта – количество байт выделенное под вспомогательные тернарные нули.	24 байта – количество байт выделенное под адреса строк. (размер адреса * кол-во строк)

Локальные переменные

Листинг:

```c
#include <stdio.h>

#define OK 0

int main(void)
{
    int var1 = 1;
    double var2 = 1.5;
    char var3 = '$';
    return OK;
}
```

В программе объявлены три переменные типов **int**, **double**, **char**

Дамп памяти:

```
(gdb) print var1
$1 = 1
(gdb) x/4xb &var1
0x7fffffffdd84:     0x01 0x00 0x00 0x00
(gdb) print var2
$2 = 1.5
(gdb) x/8xb &var2
0x7fffffffdd88:     0x00 0x00 0x00 0x00 0x00 0x00 0xf8 0x3f
(gdb) print var3
$3 = 36 '$'
```

```
(gdb) x/1xb &var3
0x7fffffffdd83:     0x24
```

Имя переменной	Размер (в байтах)	Адрес переменной
var1	4	0x7fffffffdd84
var2	8	0x7fffffffdd88
var3	1	0x7fffffffdd83

Каждый тип (кроме типа **char**) выравнивается, т.е. переменные располагаются по адресу, кратному их размеру

(7fffffffdd84 в десятичной с/с) mod 4 = 0
(7fffffffdd88 в десятичной с/с) mod 8 = 0
(7fffffffdd83 в десятичной с/с) mod 1 = 0

Т.к. размер типа **char** равен единице, он не требует выравнивания

$k \% 1 = 0, k \in N$

Структуры

Неупакованная структура

Листинг:

```c
#include <stdio.h>

#define OK 0

int main(void)
{
    struct variable
    {
        int i;
        double d;
        char c;
    };
    struct variable var = {.i = 1, .d = 1.5,.c = '$'};
    return OK;
}
```

Дамп памяти:

```
(gdb) print sizeof(var)
$1 = 24
(gdb) print var
$2 = {i = 1, d = 1.5, c = 36 '$'}

(gdb) x/24xb &var
0x7fffffffdd70:     0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffdd78:     0x00 0x00 0x00 0x00 0x00 0x00 0xf8 0x3f
```

```
0x7fffffffdd80:     0x24 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb) print sizeof(var.i)
$3 = 4
(gdb) x/4xb &(var.i)
0x7fffffffdd70:     0x01 0x00 0x00 0x00
(gdb) print sizeof(var.d)
$4 = 8
(gdb) x/8xb &(var.d)
0x7fffffffdd78:     0x00 0x00 0x00 0x00 0x00 0x00 0xf8 0x3f
(gdb) print sizeof(var.c)
$5 = 1
(gdb) x/1xb &(var.c)
0x7fffffffdd80:     0x24
```

Название поля	Размер в байтах	Адрес поля
i	4	0x7fffffffdd70
d	8	0x7fffffffdd78
c	1	0x7fffffffdd80

Поля располагаются по адресу, кратному размеру своего типа и размеру типа максимальной размерности в структуре (В примере: тип **double** – 8 байт)

Переменная структурного типа располагается по адресу: **0x7fffffffdd70**

На адрес повлиял размер типа максимальной размерности (В примере: тип **double** – 8 байт)

Размер структуры = размер максимального типа в структуе * кол-во элементов в структуре

Запакованная структура

Листинг:

```c
#include <stdio.h>

#define OK 0

int main(void)
{
    #pragma pack(push, 1)
    struct variable
    {
        int i;
        double d;
        char c;
    };
    #pragma pack(pop)
    struct variable var = {.i = 1, .d = 1.5,.c = '$'};
    return OK;
}
```

Дамп памяти:

```
(gdb) print var
$1 = {i = 1, d = 1.5, c = 36 '$'}
(gdb) print sizeof(var)
$2 = 13
(gdb) x/13xb &var
0x7fffffffdd83:     0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffdd8b:     0x00 0x00 0xf8 0x3f 0x24
(gdb) print sizeof(var.i)
$3 = 4
(gdb) x/4xb &(var.i)
0x7fffffffdd83:     0x01 0x00 0x00 0x00
(gdb) print sizeof(var.d)
$4 = 8
(gdb) x/8xb &(var.d)
0x7fffffffdd87:     0x00 0x00 0x00 0x00 0x00 0x00 0xf8 0x3f
(gdb) print sizeof(var.c)
$5 = 1
(gdb) x/1xb &(var.c)
0x7fffffffdd8f:     0x24
```

Название поля	Размер в байтах	Адрес поля
i	4	0x7fffffffdd83
d	8	0x7fffffffdd87
c	1	0x7fffffffdd8f

Поля располагаются по адресу, кратному размеру минимального по размерности типа (В примере: тип **short** – 1 байт)

Переменная структурного типа располагается по адресу: **0x7fffffffdd83**

На адрес повлиял размер типа минимальной размерности (В примере: тип **short** – 1 байт)

Размер равен сумме размеров всех элементов структур

Структура с минимальным размером

Чтобы занимаемых выравниванием битов было минимально, можно поставить меньшие поля структуры рядом. {d, a, c}

Листинг:

```c
#include <stdio.h>

#define OK 0

int main(void)
{
    struct variable
    {
        double d;
        int i;
        char c;
    };
    struct variable var = {.i = 1, .d = 1.5,.c = '$'};
    return OK;
}
```

Размер:

```
(gdb) print sizeof(var)
$2 = 16
(gdb) x/16xb &var
0x5a8c9ffb50:   0x00    0x00    0x00    0x00    0x00    0x00    0xf8    0x3f
0x5a8c9ffb58:   0x01    0x00    0x00    0x00    0x24    0x00    0x00    0x00
```

У данной структуры есть “Завершающее” выравнивание равное трем байтам

айт										0	1	2	3	4	5	6
начение	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[0]	[1]	[2]	[3]		ULL	ULL	ULL

Размер = максимальный тип структуры + выравненная сумма типов меньших структур