Laboratório 2: representação de dados (inteiros sem sinal)

1. a. O tamanho em `bytes` de um inteiro `int` é 4 na linguagem C. No entanto ao executar o código com i = 10000 podemos visualizar o seguinte:

   Código executado:

```
#include <stdio.h>

void dump (void *p, int n) {
  unsigned char *p1 = p;
  while (n--) {
    printf("%p - %02x\n", p1, *p1);
    p1++;
  }
}

int main (void) {
  int i = 10000;
  dump(&i, sizeof(i));
  return 0;
}
```

   Saída gerada:

```
$ ./lab2
0x7ffcca4ed65c - 10 (Menos significativo)
0x7ffcca4ed65d - 27
0x7ffcca4ed65e - 00
0x7ffcca4ed65f - 00 (Mais significativo)
```

10000 (decimal) = 2710 (hexadecimal)
Pode-se observar que a saída do programa gerou como saída dois dados por linha. O primeiro deles é o endereço de memória, o segundo são os dígitos que compõe o número. Como está sendo impresso o número em hexadecimal, como cada linha imprime apenas 1 `byte` por linha e esse valor corresponde a 8 `bits` , além disso são exibidos dois digitos por linha. Pode-se afirmar que o número 10000 está sendo armazenado na memória em formato `little-endian` . Como são impressos 4 linhas no total conclui-se que um `int` possuí 4 bytes.

b. Saída gerada:

```
0x7ffd0bf9c018 - 10 (menos significativo)
0x7ffd0bf9c019 - 27
0x7ffd0bf9c01a - 00
0x7ffd0bf9c01b - 00
0x7ffd0bf9c01c - 00
0x7ffd0bf9c01d - 00
0x7ffd0bf9c01e - 00
0x7ffd0bf9c01f - 00 (mais significativo)
```

Pode-se concluir que o tamanho de um tipo `long` é 8 `bytes` .

c. Saída gerada:

```
0x7ffc3d142c6e - 10
0x7ffc3d142c6f - 27
```

Logo um tipo `short` possui 2 `bytes` .

d. Saída gerada passando o char ‘a’:

```
0x7fff413865bf - 61
```

61 (hexadecimal) = 97 (Decimal), bate com a tabela `ASCII`

Saída gerada com char 97:

```
0x7ffe1013a7bf - 61
```

O que bate com a resposta esperada.

```
0x7fffb0fc08fb - 37
0x7fffb0fc08fc - 35
0x7fffb0fc08fd - 30
0x7fffb0fc08fe - 39
0x7fffb0fc08ff - 00
```

Trocando por ‘A’, ‘ ‘, ‘\n’, e ‘$’.

```
'A'
0x7ffd841e40de - 41
0x7ffd841e40df - 00

' '
0x7fff6967a81e - 20
0x7fff6967a81f - 00

'\n'
0x7ffe5606960e - 0a
0x7ffe5606960f - 00

'$'
0x7fff4532ee0e - 24
0x7fff4532ee0f - 00
```

2. Código executado:

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

```
int string2num (char *s) {
  int a = 0;
  for (; *s; s++)
    a = a*10 + (*s - '0');
  return a;
}

int main (void) {
  printf("⟹ %d\n", string2num("1234"));
  printf("⟹ %d\n", string2num("1234") + 1);
  printf("⟹ %d\n", string2num("1234") + string2num("1"));
  return 0;
}
```

Funcionamento do código: como o número é gerado?

No loop for: (supondo string “1234”)

a = 0 → a = 0 ∗ 10 + (48 - 49)

a = 1 → a = 0 ∗ 10 + (47 - 49)

a = 12 → a = 12 ∗ 10 + (46 - 49)

a = 123 → a = 123 ∗ 10 + (45 - 49)

a = 1234

b. Modificação realizada:

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

int string2num (char *s, int base) {
  int a = 0;
  for (; *s; s++)
    a = a*base + (*s - '0');
  return a;
}

int main (void) {
  printf("%d\n", string2num("777", 8));
```

```
  printf("%d\n", string2num("777", 10));
  return 0;
}
```

Saída gerada:

```
511 (na base octal)
777 (na base decimal)
```

c. Modificação do código:

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

int string2num (char *s, int base) {
  int a = 0;
  for (; *s; s++)
  {
    if (isdigit(*s))
      a = a*base + (*s - '0');

    else
      a = a*base + (*s - 87);
  }
  return a;
}

int main (void) {
  printf("%d\n", string2num("1a", 16));
  printf("%d\n", string2num("a09b", 16));
  printf("%d\n", string2num("z09b", 36));
  return 0;
}
```

Saída gerada:

```
26
41115
1633295
```