CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

Understand your data in memory

- All data saved in memory is in binary format
- Accessing using memory addresses
  - Pointers
- Data type (e.g., signed/unsigned, char, int, float, …)
  - unit length (e.g., 1B, 4B, …) and the way to interpret the bits (sign bit, exponent, value, …)
- All operations directly handle binary data
  - Arithmetic operations (addition, multiplication, …)
  - Bitwise operations.

Bit string: a stream of 0s and 1s.

Bit 1 vs. char ‘1’ vs. integer 1 vs. floating point 1

```
$ cat ./binary.c
#include <stdio.h>
int main() {
  char c='1';   int i=1;   float f=1;
  printf("%c %i %f\n", c, i, f);
}
$ gcc -ggdb -o binary ./binary.c
$ gdb ./binary
(gdb) break 4
(gdb) run
(gdb) x/tb &c
0x7fffffffe407: 00110001
(gdb) x/tw &i
0x7fffffffe408: 00000000000000000000000000000001
(gdb) x/tw &f
0x7fffffffe40c: 00111111100000000000000000000000
```

Type	Size
char	1 bytes
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
pointer	8 bytes
size_t	8 bytes

Binary data in memory (explore using gdb)

```
$ cat ./ binary_content.c
#include <stdio.h>
#include <stdlib.h>

main(){
    int i[20], value, j;
    float f[20];

    value = -10;
    for( j = 0; j < 20; j++) {
        i[j] = value;
        f[j] = value;
        value = value + 1;
    }
    printf("Examine memory now.\n");
}
```

What do arrays I and f look like in memory?

Binary data in memory (explore using gdb)

```
$ gcc -ggdb -o binary_content ./binary_content.c
$ gdb ./binary_content
(gdb) list
(gdb) list
(gdb) break 15
(gdb) r
(gdb) x/20dw i
0x7fffffffe380
0x7fffffffe390
0x7fffffffe3a0
0x7fffffffe3b0
0x7fffffffe3c0
```

data

Memory addreses

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char), s(string)
  and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
```

Binary data in memory (explore using gdb)

```
(gdb) x/20fw f
0x7fffffffe3d0: -10      -9       -8       -7
0x7fffffffe3e0: -6       -5       -4       -3
0x7fffffffe3f0: -2       -1       0        1
0x7fffffffe400: 2        3        4        5
0x7fffffffe410: 6        7        8        9
(gdb) x/20dw f
???
(gdb) x/20fw i
???
(gdb) x/20tw i
???
(gdb) x/20tw f
???
(gdb) x/20tw &i
???
```

- Binary data in memory can be interpreted in different ways (**types**).
- The same data in memory represent different values when casted into different types.
- How to verify this in programs?

Questions:

- Is “type” information saved in memory?
- If it is saved, why changing types is allowed?
- If it is not, how CPU knows the correct way to interpret the data? (ADD vs. FADD)

This allows us to interpret the same data in a different way. (int *) changes the type.

Same numbers printed out as what is printed out in gdb with command *x/20dw f*

```
$ cat ./ binary_content.c
#include <stdio.h>
#include <stdlib.h>

main(){
    int i[20], value, j, k;
    float f[20];
    unsigned int *p;

    value = -10;
    for( j = 0; j < 20; j++) {
        i[j] = value;
        f[j] = value;
        value = value + 1;
    }
    p = (unsigned int *)f;
    for ( j = 0; j < 5; j++ ) {
        for ( k = 0; k< 4; k++)
            printf("%u\t", p[j*4+k]);
        printf("\n");
    }
    printf("Examine memory now.\n");
}
```

Binary vs. text

```
int a="123", b="234";
c=a+b;  /* is C 357? */


int a=123, b=234;
c=a+b;  /* is C 357? */
```

- Write a C program that saves 10 million integers into a file. Write another C program that reads the integers out from the file into an array.
  - Do you save text or binary into the file?
  - Saving text into the file: takes much more time to read/write, uses much more space, lacks uniformity (difficult to calculate the count, difficult to calculate the offset of a particular value).

Let's explore how a structure is saved in memory

```
$ cat structure.c
#include <stdio.h>
#include <stdlib.h>

struct record{
   int  index;
   char name[8];
   float score;
};

main(){
   struct record rec1 = {1, "Tom", 85.5};

   printf("Examine memory now.\n");
}
```

How are the fields in a structure saved in memory?

Structure saved in memory

```
$ gcc -ggdb -o ./structure ./structure.c
$ gdb ./ structure
(gdb) list
(gdb) list
(gdb) break 13
(gdb) r
(gdb) x/16bx &rec1
0x7fffffffe3f0: 0x01   0x00   0x00   0x00   0x54   0x6f   0x6d   0x00
0x7fffffffe3f8: 0x00   0x00   0x00   0x00   0x00   0x00   0xab   0x42
(gdb) x/1dw 0x7fffffffe3f0
0x7fffffffe3f0: 1
(gdb) x/4cb 0x7fffffffe3f4
0x7fffffffe3f4: 84 'T'  111 'o' 109 'm' 0 '\000,
(gdb) x/1fw 0x7fffffffe3fC
0x7fffffffe3fc: 85.5
```

rec1.index starting from 0x7fffffffe3f0

rec1.name starting from 0x7fffffffe3f4

rec1.score starting from 0x7fffffffe3fC

In a program, can we access the data if we know its address and type?

Accessing data if you know address and type

```
$ cat ./address_type_2_data.c
#include <stdio.h>
#include <stdlib.h>

struct record{
   int  index;
   char name[8];
   float score;
};

main(){
   struct record rec1 = {1, "Tom", 85.5};
   int *field1 = (int *)(&rec1);
   char *field2 = (char *)(&rec1) + 0x4;
   float *field3 = (float *)((char *)(&rec1) + 0xC);

   printf("index: %d\n", *field1);
   printf("name: %s\n", field2);
   printf("score: %f\n", *field3);
}
```

```
$ ./address_type_2_data
index: 1
name: Tom
score: 85.500000
```

Let’s explore how 2D and 3D arrays are saved in memory

```
$ cat ./array2d.c
#include <stdio.h>
#include <stdlib.h>

main(){
    int array[3][2], value=0, i, j;

    for( i = 0; i < 3; i++) {
        for ( j = 0; j < 2; j++){
            array[i][j] = value;
            value = value + 1;
        }
    }
    printf("Examine memory now.\n");
}
```

How are the elements in a 2D array saved in memory?

Let’s explore how 2D and 3D arrays are saved in memory

```
$ gcc -ggdb -o array2d ./array2d.c
$ gdb ./array2d
(gdb) list
(gdb) list
(gdb) break 14
(gdb) r
(gdb) x/8dw array
0x7fffffffe3f0: 0       1       2       3
0x7fffffffe400: 4       5       1713559808      143097460
```

Can you tell whether it is a 1D or 2D array, and size of each dimension? Can it be the following array?

1D: 0 1 2 3 4 5

2D: ((0 1 2) (3 4 5))

Questions:

- Since there is no difference in memory, can we use a 2D array as a 1D array in a program, or vise versa?
- Since there is no dimensional information (part of type info), how does a processor locate the proper elements based on indexes?

Data in 2D array used as that in a 1D array

```
$ cat ./array2d_to_1d.c
#include <stdio.h>
#include <stdlib.h>

main(){
    int array[3][2], value=0, i, j;
    int *p=(int *)array;

    for( i = 0; i < 3; i++) {
        for ( j = 0; j < 2; j++){
            array[i][j] = value;
            value = value + 1;
        }
    }

    for( i = 0; i < 6; i++)
        printf("%d ", p[i]);
    printf("\n");
}
```

This allows us to interpret the 2D data as 1D data.
(int *) changes the type.

Prints out 0 1 2 3 4 5

Data in 2D array used as that in a 1D array

This allows us to interpret the 2D data as 1D data.
(int *) changes the type.

Prints out 0 1 2 3 4 5

```
$ cat ./array2d_to_1d.c
#include <stdio.h>
#include <stdlib.h>

main(){
    int array[3][2], value=0, i, j;
    int *p=(int *)array;

    for( i = 0; i < 3; i++) {
        for ( j = 0; j < 2; j++){
            array[i][j] = value;
            value = value + 1;
        }
    }

    for( i = 0; i < 3; i++)
        for ( j = 0; j < 2; j++)
            printf("%d ", p[i*2+j]);
    printf("\n");
}
```

Your turn to explore how 3D arrays are saved in memory

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

main(){
   int array[3][2][2], value=0, i, j,
k;

   for( i = 0; i < 3; i++) {
      for ( j = 0; j < 2; j++){
         for ( k = 0; k < 2; k++) {
            array[i][j][k] = value;
            value = value + 1;
         }
      }
   }
   printf("Examine memory now.\n");
}
```

Use gdb to show the location and contents of the 3D array in memory.

Modify the program and access the elements of the 3D array as accessing those in a 1D array.

Data in 3D array used as that in a 1D array

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

main(){
    int array[3][2][2], value=0, i, j, k;
    int *p=(int *)array;

    for( i = 0; i < 3; i++) {
        for ( j = 0; j < 2; j++){
            for ( k = 0; k < 2; k++) {
                array[i][j][k] = value;
                value = value + 1;
            }
        }
    }
    for( i = 0; i < 12; i++)
        printf("%d ", p[i]);
    printf("\n");
    printf("Examine memory now.\n");
}
```

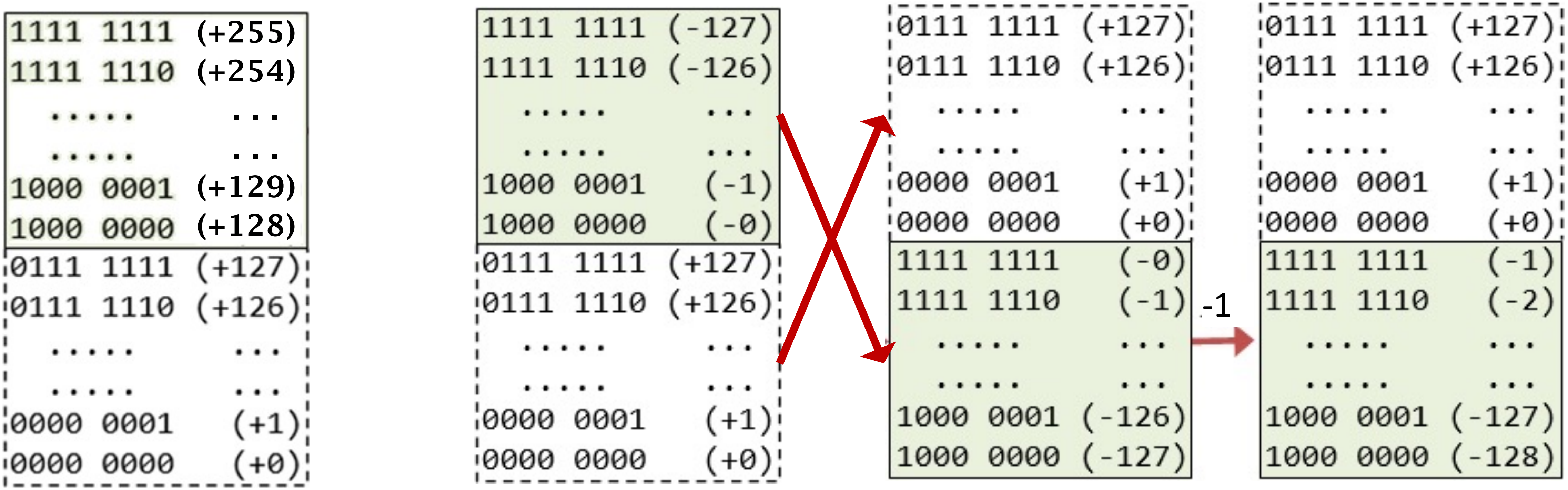
How is binary data "translated" into different types of values?

- Each type has a fixed size

Type	char	short	int	long	float	double	pointer	size_t
Size(bytes)	1	2	4	8	4	8	8	8

- Signed type uses the highest bit as the sign bit
  - 1 --- negative, 0 --- positive.
- In integer types (char, short, int, long, size_t), all/remaining bits represent the value
  - note: not the absolute value.
  - Last bit differentiate odd numbers vs. even numbers.
- *Float* number types (float, double, etc) use some bits for exponents.
- More details will be given using char and float as examples.

How does a char/integer use the bits



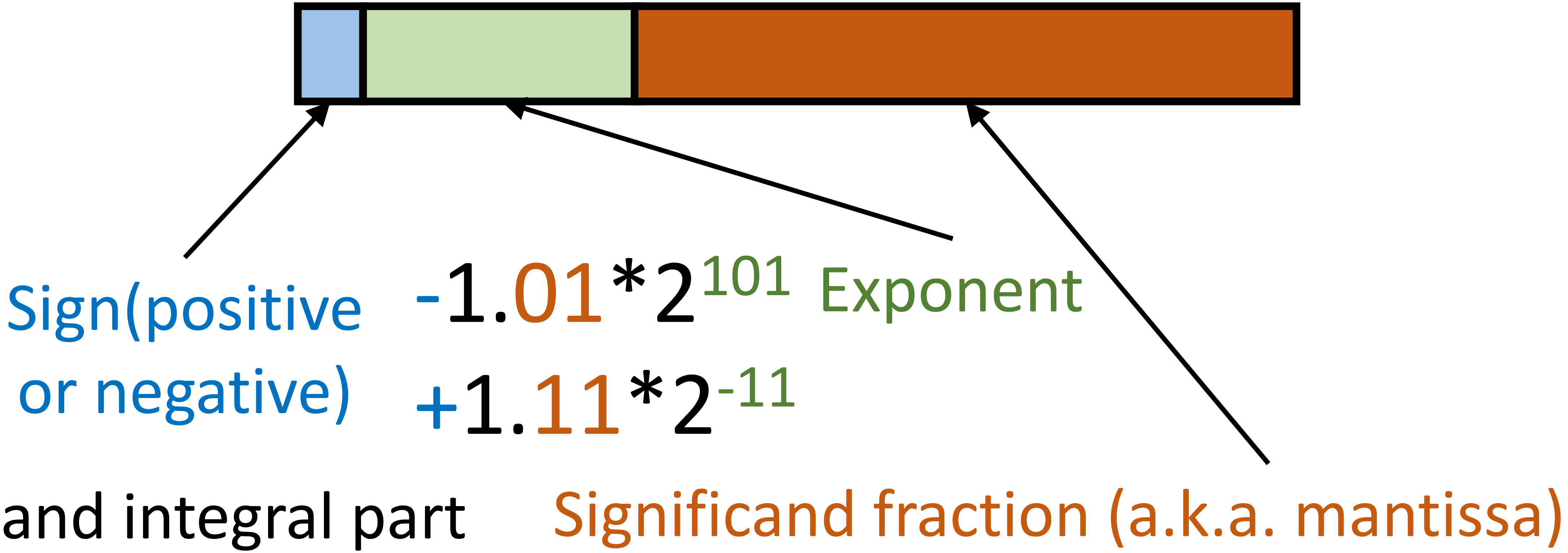
unsigned char

signed char

Int and long values have more bits, but the formats are similar.

Floating-Point Representation in Computer

Computer representation of a floating-point number consists of three fixed-size fields:

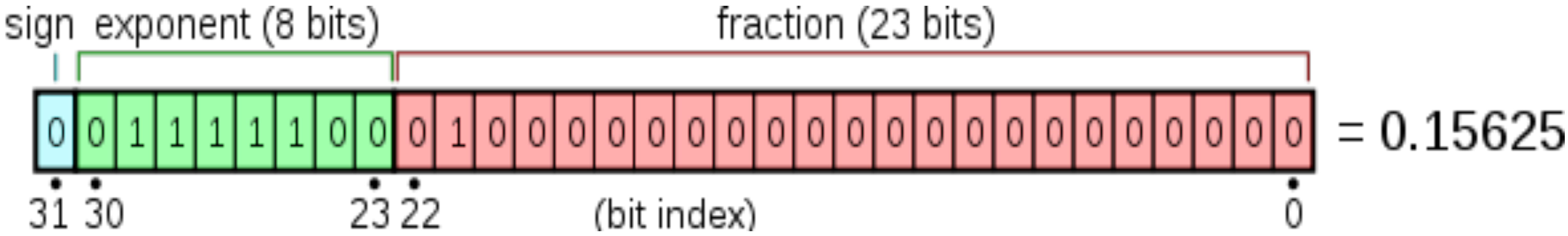


$-1.01*2^{101}$

$+1.11*2^{-11}$

Base(2) and integral part (1) are fixed values

- Sign: 1 bit; 0 --- positive, 1 --- non-positive
- Significand fraction: 23 bits
- Biased exponent: 8 bits.
  - Bias: represent –127 to +127 by adding 127 (so range is 0-254, not 0~255). Subtract 127 to get real exponent.
  - Invalid float if biased exponent is 0xFF. (gdb prints “NaN”.)



$$+1.01*2^{(01111100-01111111)}$$

$$1.01*2^{-11} \Rightarrow 101*2^{-101} \Rightarrow 5/32$$

```
$ cat ./test.c
#include <stdio.h>
#include <stdlib.h>

main()
{
  char c;
  float f;

  while(1) {
     scanf("%d", &c);
     scanf("%f", &f);
     printf("Set breakpoint here.\n");
  }
}
```

Use gdb and the program on the left to check how the values you input are saved in memory.

x commands:

`x/1tb &c`

`x/1tw &f`

Bitwise operations

Bit string and bitwise operations for processing raw data

- There is no data type in C defined for handling bit strings.
- A bit string is usually managed as an array of unsigned int (4B), unsigned long (8B), or unsigned char(1B)
  - No special bits (no sign bits, no exponent bits)
  - A bit string is one or more units, and each unit has multiple bits (32, 64, 8)
  - Processing the bits in a bit string is done unit by unit.

Bitwise operators

Operator	Name	Arity	Description
&	Bitwise AND	Binary	Similar to the && operator, but on a bit-by-bit basis.
|	Bitwise OR	Binary	Similar to the || operator, but on a bit-by-bit basis.
^	Bitwise Exclusive OR	Binary	Set to 1 if one of the corresponding bits is 1, or set to 0 otherwise.
~	Complement	Unary	Flips the bits in the operand.

Bitwise operators

Operator	Name	Arity	Description
<<	Left shift	Binary	Shifts the bits of the first operand to the left by the number of bits specified in the second operand. Right fill with 0 bits.
>>	Right shift	Binary	Shifts the bits of the first operand to the right by the number of bits specified in the second operand. Left fill with 0's for positive numbers, 1's for negatives (machine dependent).

Examples

- Suppose we have the following code

  unsigned short x = 6891;

  unsigned short mask = 11318;
- Assume short is 2 bytes (16 bits)

```
       x:        00011010 11101011
       --------------------------
  x << 2:        01101011 10101100 (27564)
```

Examples

```
y:        00101100 00110110
          -----------------
y >> 4:   00000010 11000011 (707)
x:        00011010 11101011
          -----------------
~x:       11100101 00010100 (58644)
```

Examples

```
x              00011010 11101011
mask           00101100 00110110
    --------------------------
x & mask:      00001000 00100010 (2082)
```

Masking bits to 0: turn some bits into 0 and keep other bits unchanged

Bit mask

- Data are handled unit by unit (e.g., 32-bit unit for unsigned int).
- Bitwise operations work with all bits in a unit.
- A lot of cases, we want to manipulate individual bits (e.g. turn them on or off).
- We need some way to identify the specific bits we want to manipulate.
- A bit mask is a predefined set of bits that is used to select which specific bits will be modified by bitwise operation.

Exacting bits (keep the selected bits, turn-off other bits, and shift)

Consider the following mask and two bit strings from which we want to extract bit(s):

mask   = 00001000
value1 = 10011101
value2 = 10010110

mask & value1 == 00001000
mask & value2 == 00000000
(mask & value1)>>3 == 1
(mask & value2)>>3 == 0

The mask *masks off* seven bits and only let bit 3 show through

Examples

Masking bits to 1: turn some bits into 1 and keep other bits unchanged

```
x              00011010 11101011
mask           00101100 00110110
            ---------------------------
x | mask:      00111110 11111111 (16127)

x              00011010 11101011
mask           00101100 00110110
            ---------------------------
x ^ mask:      00110110 11011101 (14045)
```

flip some bits and keep other bits unchanged

Shortcut assignment operators

- x &= y means x = x & y
- x |= y means x = x | y
- x ^= y means x = x ^ y
- x <<= y means x = x << y
- x >>= y means x = x >> y

```c
/* binary representation of char*/
/* using different masks to get different bits */
#include <stdio.h>

int main() {
    unsigned char a=128;
    unsigned char mask;
    int i;

    for (i=0;i<sizeof(a)*8;i++){
        mask = 1<<(7-i);
        printf("%u", (a & mask)>>(7-i));
    }
    return printf("\n");
}
```

```c
/* binary representation of char*/
/* shifting the bits and use the same mask to get
 * different bits. */
#include <stdio.h>

int main() {
    unsigned char a=128;
    int i;

    for (i=0;i<sizeof(a)*8;i++){
        printf("%u", (a & 0x80)>>7);
        a=a<<1;
    }
    return printf("\n");
}
```

```c
/* binary representation of int */
#include <stdio.h>

int main() {
    /* 32*32 bits */
    unsigned int bitstring[32], i, j, mask, unit;

    for (i=0;i<32;i++) bitstring[i]=-1*i;

    mask=1<<31;
    for (i=0;i<32;i++){
        unit=bitstring[i];
        for (j=0; j<32; j++) {
            printf("%u", (unit & mask)>>31);
            unit=unit<<1;
        }
        printf(" ");
    }
    return printf("\n");
}
```

Creating bit masks

- Determine values directly, e.g., unsigned in mask=0x0F0F0F0F;
- Calculation. E.g., unsigned int mask=(1<<16)-1;
- Masks can be built up by operating on several flags using inclusive OR:

```
flag1 = 00000001
flag2 = 00000010
flag3 = 00000100

mask = flag1 | flag2 | flag3

mask == 00000111
```

- Left-shift 1s if you know bit indexes

```
/* set bit 2, bit 5, and bit 10 */
mask=0;
bit_index=2;
mask = 1<<bit_index;
bit_index=5;
mask = mask | (1<<bit_index) ;
bit_index=10;
mask = mask | (1<<bit_index)
```

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

/* Function returns the only odd occurring element (other elements
occur in pairs.*/
int findOdd(unsigned int arr[], int n) {
   unsigned int res = 0, i;
   for (i = 0; i < n; i++)
     res ^= arr[i];
   return res;
}

int main(void) {
   unsigned int arr[] = {12, 12, 14, 90, 14, 14, 14};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf ("The odd occurring element is %u", findOdd(arr, n));
   return 0;
}
/* Output: The odd occurring element is 90 */
```