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 addresses

(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

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

```
$ 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");
}
```

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");
}
```

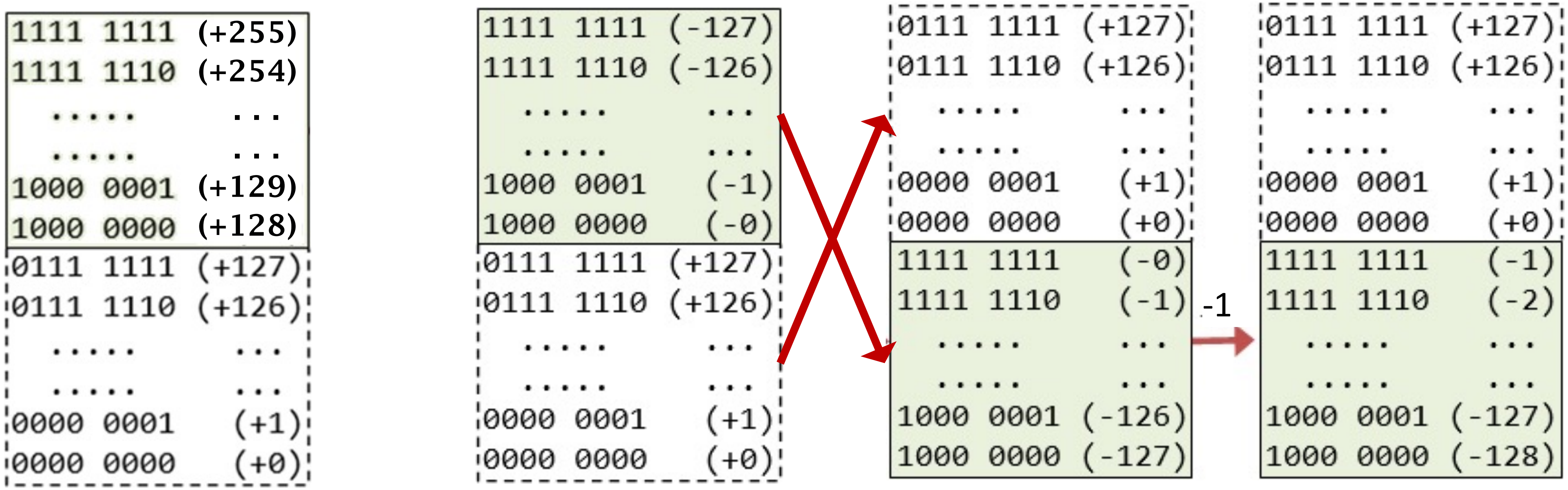
How is binary data “translated” into different types of values?

- 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



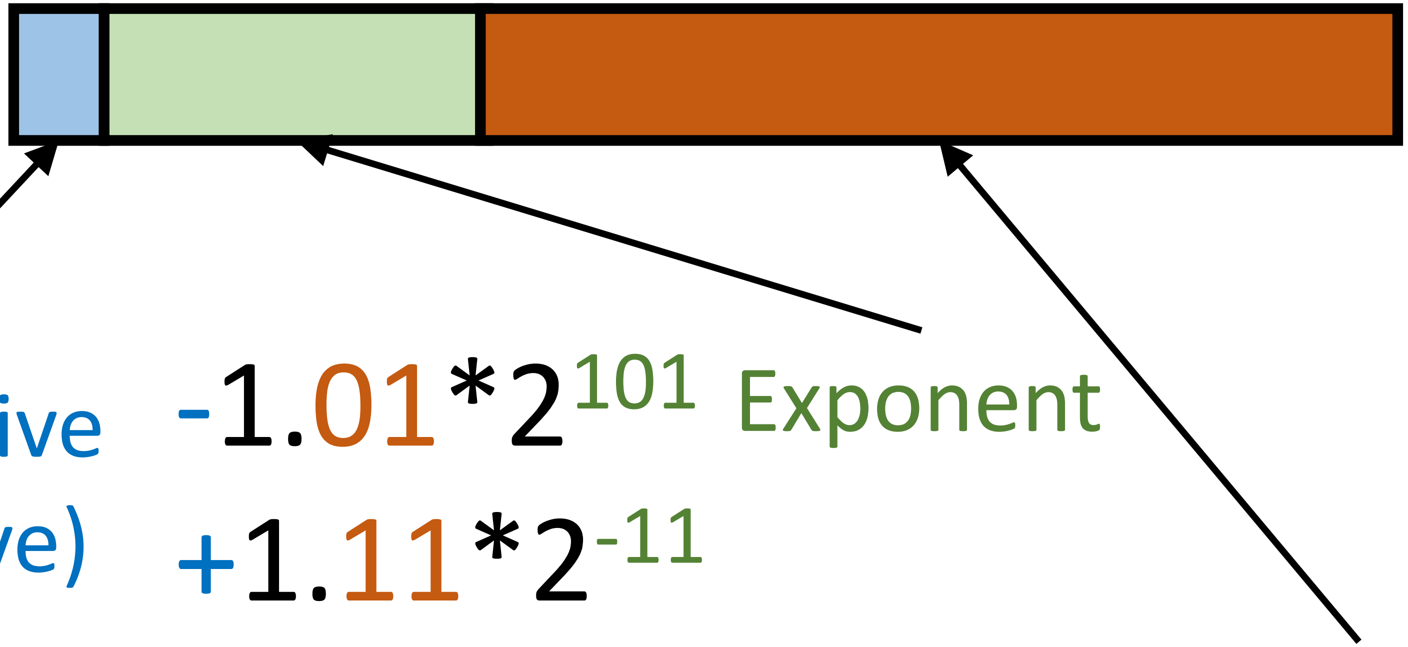
unsigned char

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:



Sign(positive or negative)

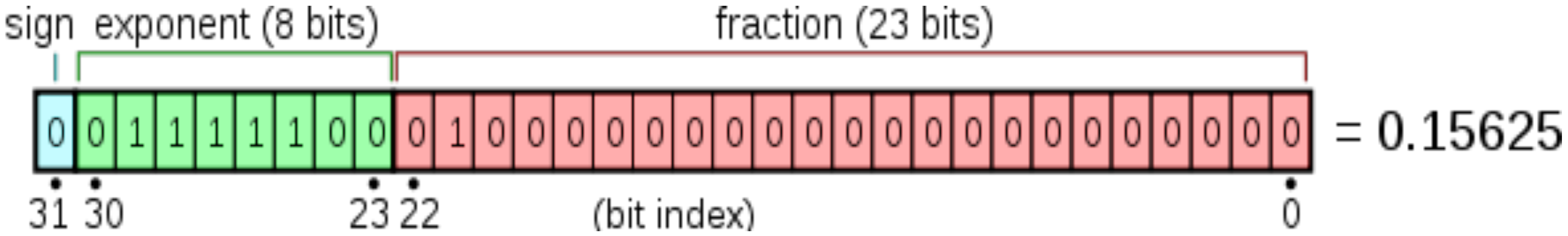
$-1.01*2^{101}$ Exponent

$+1.11*2^{-11}$

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

Significand fraction (a.k.a. mantissa)

- 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

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

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 */
```

CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

Sorting algorithms

- A fundamental application for computers
- Done to make finding data (searching) faster
- Many different algorithms for sorting
  - bubble sort, selection sort, insertion sort, quick sort, heap sort, …
- Sorting is usually done with multiple rounds
  - Simple sorting algorithms run in $O(N^2)$ time. Some uses $O(n\log(n))$ time. Best algorithms use $O(n)$ time.
- Conventional sorting algorithms: https://www.toptal.com/developers/sorting-algorithms
- We discuss sorting values in "ascending" order in the class.
  - it is not difficult to figure out how to change the order to "descending".

Bucket sort

To sort N integer values within a range of (L, H)

- If we use *H-L+1 buckets*, one for each possible value, sorting is done by simply putting each integer into the corresponding bucket.
  - Fast, especially when N is large and range is small.
  - Too expensive, especially when N is small and range is large. E.g., sorting 100 unsigned integers needs 2^{32} buckets.
- Solution: use fewer buckets to find a good trade-off between N and range.
  - Each bucket serves the values within a smaller range.
    - E.g., two buckets, one for range (L, M) and one for range (M, H), M=(H+L)/2.
  - Put values into buckets.
  - Sort the values in each bucket.
    - Apply bucket sort recursively and/or apply other sorting algorithm when the values in a bucket are not many.

Radix Sort: bucket sort on every bit

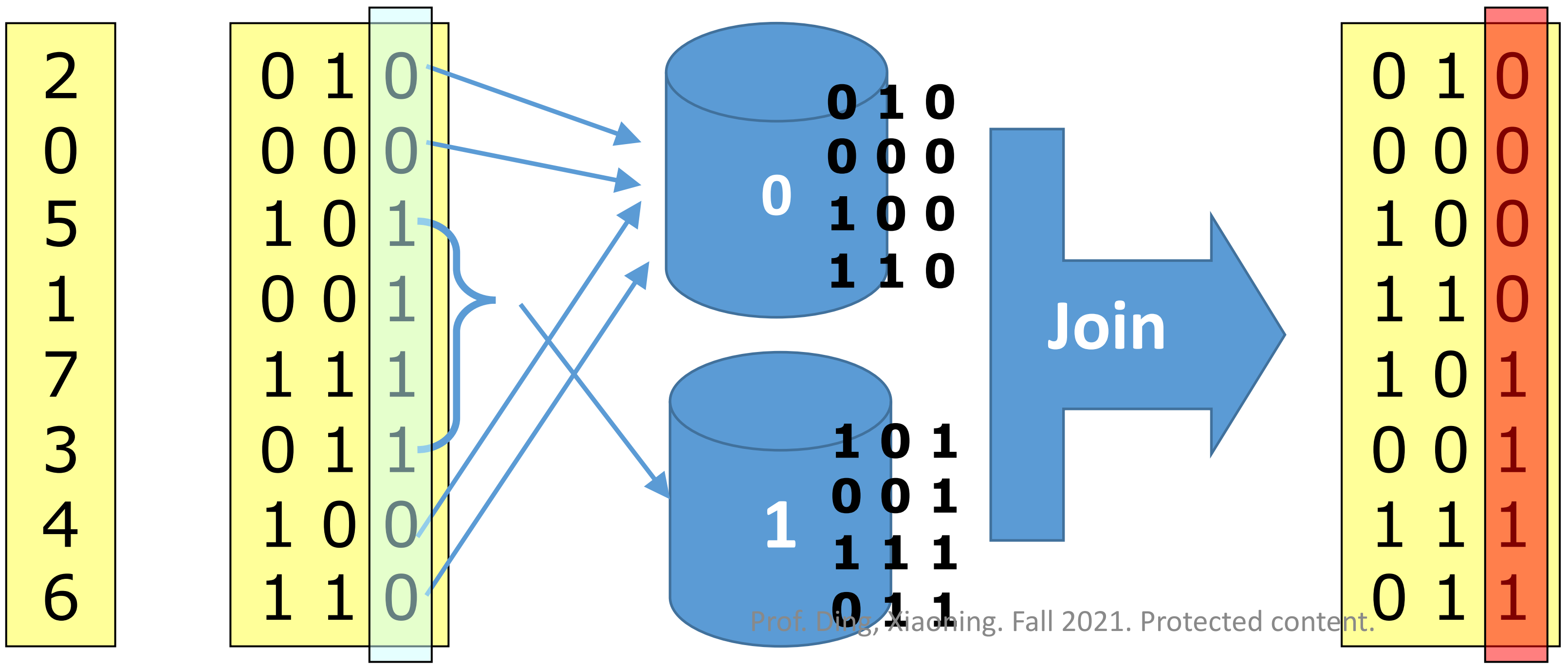
- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is $(-1.7 \times 10^{308}, 1.7 \times 10^{308})$
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.

2	0 1 0
0	0 0 0
5	1 0 1
1	0 0 1
7	1 1 1
3	0 1 1
4	1 0 0
6	1 1 0

Use two buckets.

Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is $(-1.7 \times 10^{308}, 1.7 \times 10^{308})$
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.



Last bits are sorted

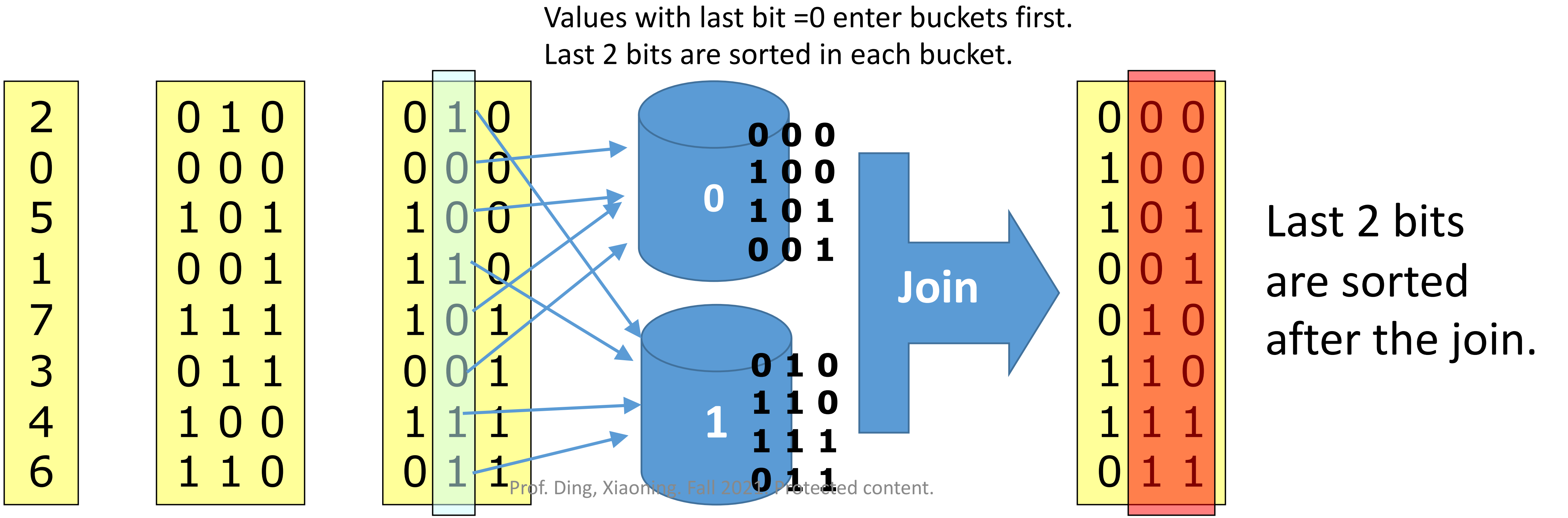
Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is ($-1.7\text{x}10^{308}$, $1.7\text{x}10^{308}$)
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.

2	0 1 0	0 1 0
0	0 0 0	0 0 0
5	1 0 1	1 0 0
1	0 0 1	1 1 0
7	1 1 1	1 0 1
3	0 1 1	0 0 1
4	1 0 0	1 1 1
6	1 1 0	0 1 1

Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is $(-1.7 \times 10^{308}, 1.7 \times 10^{308})$
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.



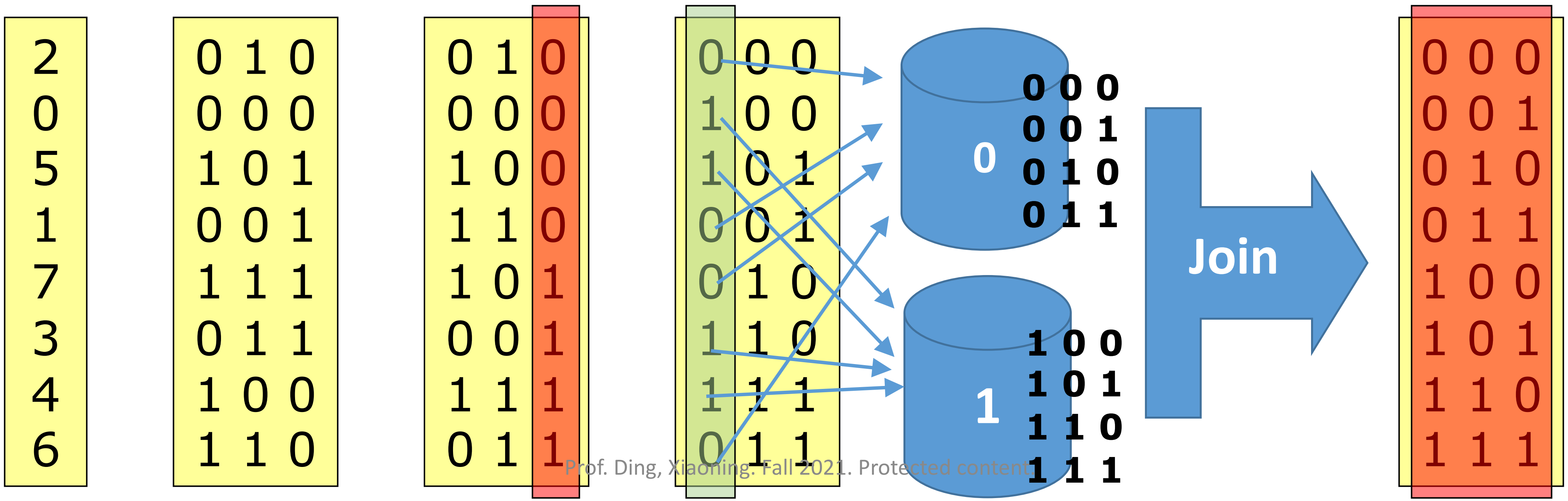
Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is $(-1.7\text{x}10^{308}, 1.7\text{x}10^{308})$
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.

2	0 1 0	0 1 0	0 0 0
0	0 0 0	0 0 0	1 0 0
5	1 0 1	1 0 0	1 0 1
1	0 0 1	1 1 0	0 0 1
7	1 1 1	1 0 1	0 1 0
3	0 1 1	0 0 1	1 1 0
4	1 0 0	1 1 1	1 1 1
6	1 1 0	0 1 1	0 1 1

Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is $(-1.7\text{x}10^{308}, 1.7\text{x}10^{308})$
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.



Radix Sort: bucket sort on every digit/bit

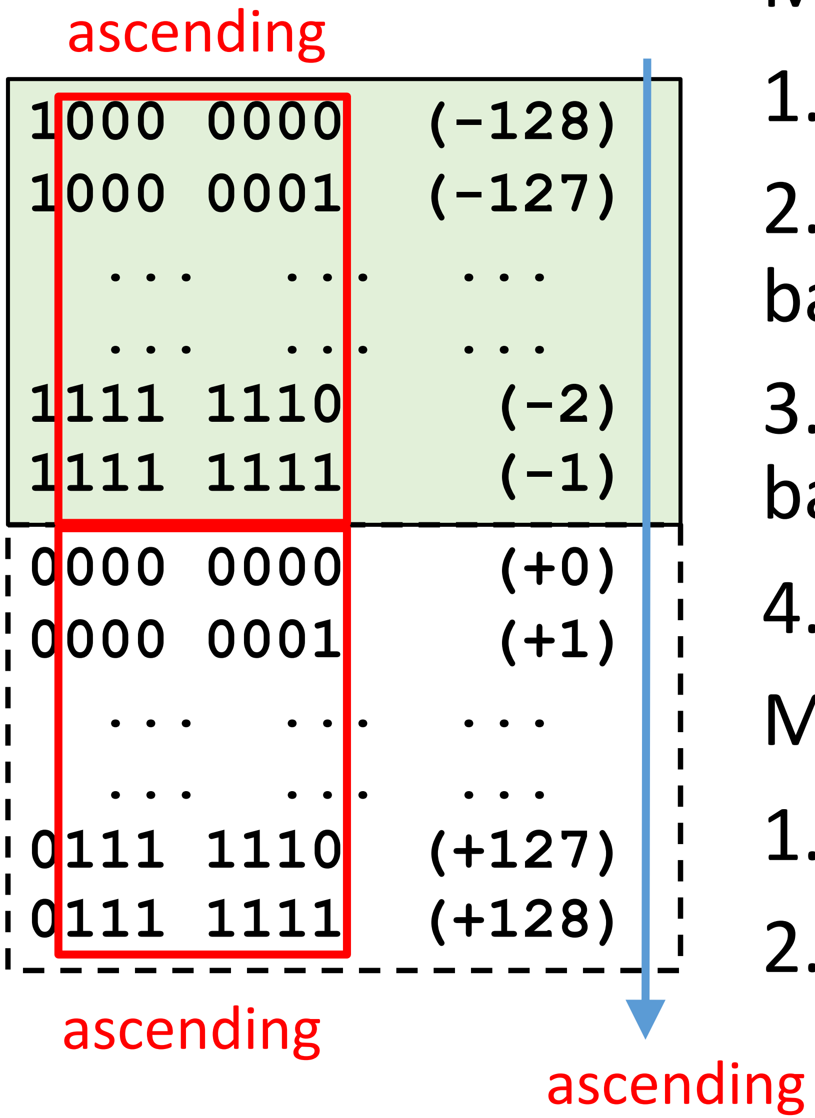
- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is ($-1.7\text{x}10^{308}$, $1.7\text{x}10^{308}$)
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.

2	0 1 0	0 1 0	0 0 0	0 0 0	0
0	0 0 0	0 0 0	1 0 0	0 0 1	1
5	1 0 1	1 0 0	1 0 1	0 1 0	2
1	0 0 1	1 1 0	0 0 1	0 1 1	3
7	1 1 1	1 0 1	0 1 0	1 0 0	4
3	0 1 1	0 0 1	1 1 0	1 0 1	5
4	1 0 0	1 1 1	1 1 1	1 1 0	6
6	1 1 0	0 1 1	0 1 1	1 1 1	7

Radix-sort unsigned integers

```
radix_sort(A, n, k) {   /* A: array; n: number of items; */
    /* k: number of bits in each item (32 for unsigned int) */
        create two buckets  (buckets can be arrays or lists)
        for (d = 0; d <k; d++) {
            /* sort A using d-th bit as the key. */
            for (i = 0; i<n; i++) {
                if the d-th bit (from right) of A[i] is 0
                    add A[i] to bucket #0
                else
                    add A[i] to bucket #1
            }
            A = Join the buckets
        }
}
```

Radix-sort **integers with signs**



Method1

1. Separate positive numbers and negative numbers.
2. Radix-sort positive numbers in ascending order based on low 31 bits
3. Radix-sort negative numbers in ascending order based on low 31 bits
4. Join positive numbers and negative numbers.

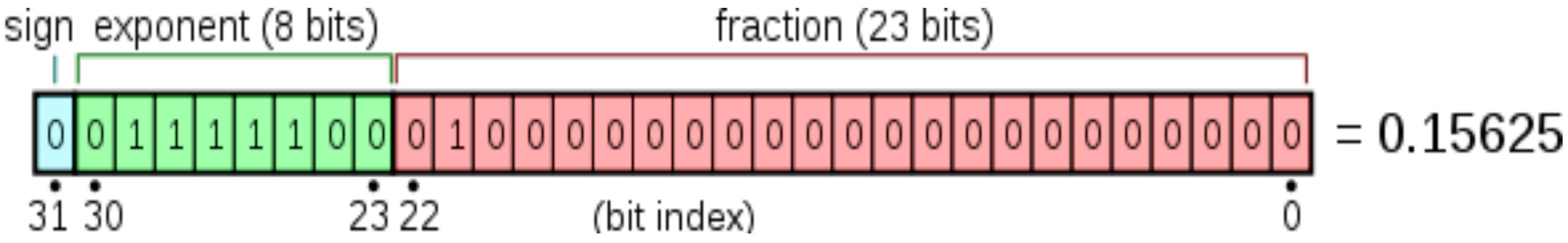
Method2:

1. Sort all values as if they were unsigned
2. In the last round joining the two bucket, put values from bucket 1 before values from bucket 0.

Radix Sort IEEE Floats/Doubles

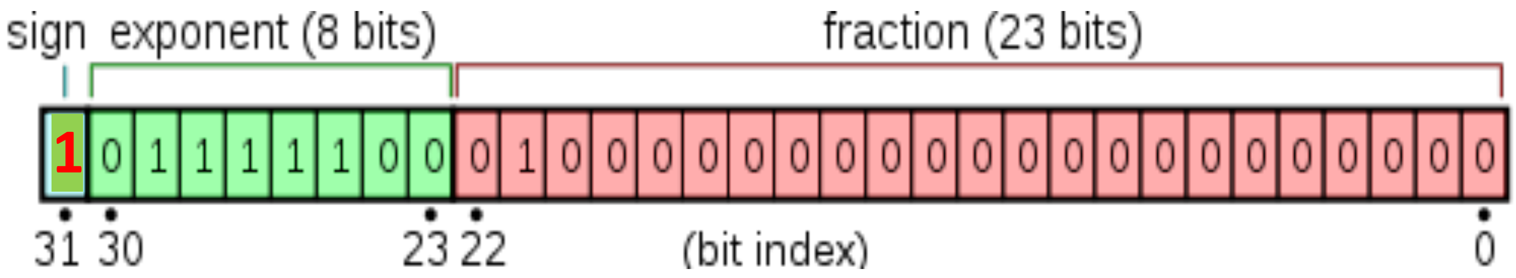
- It is straightforward to use radix sort on integers.
- Some people say you can't Radix Sort real numbers.
- You can Radix Sort real numbers, in most representations
- We do IEEE floats/doubles, which are used in C/C++.

Observations



- Non-negative float point numbers
  - Larger value in a digit means larger number
    - e.g., 0 01111100 **0**10000… =0.15625 (the value above);
      0 01111100 **1**10000… =0.21875
    - When joining buckets, bucket with a **smaller** digit value comes first to achieve “ascending” order
  - Values are more determined by a higher digit than any lower digits
    - e.g., 0 01111100 **01**0000… =0.15625 (the value above);
      0 01111100 **10**0000… =0.18750
    - Exponent always more significant than significand
    - e.g., 0 0111110**0** 0**1**0000… =0.15625 (the value above);
      0 0111110**1** 0**0**0000… =0.25
    - Repeat the rounds from the least significant bit to most significant bit

Observations



= -0.15625

- Negative float point numbers
  - Larger value in a digit means **smaller** number
    - e.g., 1 01111100 **0**10000… = -0.15625 (the value above);
      1 01111100 **1**10000… = -0.21875 (smaller)
    - When joining buckets, the bucket with a **larger** digit value comes first to achieve "ascending" order
  - Values are more determined by a higher digit than any lower digits
    - e.g., 1 01111100 **01**0000… =-0.15625 (the value above);
      1 01111100 **10**0000… =-0.18750 (smaller)
    - Exponent always more significant than significand
    - e.g., 0 0111110**0** 0**1**0000… = -0.15625 (the value above);
      0 0111110**1** 0**0**0000… = -0.25 (smaller)
    - Repeat the rounds from the least significant bit to most significant bit

What if there are non-negative numbers and negative numbers?

- Method 1: sort non-negative numbers and negative numbers separately
  - Pay attention to the way of joining the buckets
  - Put all non-negative numbers after negative numbers
- Method 2: what if we sort non-negative and negative numbers together in the same way?
  - Step 1: sort all the numbers as if they were all **unsigned integers**.
    - Join the buckets in the same way (smaller digits first) for all the numbers
  - As illustrated later, when step 1 is finished,
    - all the negative numbers come after non-negative numbers
    - non-negative numbers are **ascending**
    - negative numbers are **descending**
  - Fix the order by re-organizing the numbers.
    - Flip the order of negative #s, and move negative #s before non-negative #s.

1056964608	0.50
1069547520	1.50
1075838976	2.50
1080033280	3.50
1083179008	4.50
1085276160	5.50
1087373312	6.50
1089470464	7.50
1091043328	8.50
3204448256	-0.50
3217031168	-1.50
3223322624	-2.50
3227516928	-3.50
3230662656	-4.50
3232759808	-5.50
3234856960	-6.50
3236954112	-7.50
3238526976	-8.50
3239575552	-9.50
3240624128	-10.50

If you sort -10.5, -9.5, …, 7.5, 8.5 as you do for unsigned int

a

```
#include <stdlib.h>

main(){
    int i;
    float value, f[20];
  /* typecasting w/ a pointer */
    unsigned int *p =
             (unsigned int *) f;

    value = -10.5;
    for( i = 0; i < 20; i++) {
       f[i] = value;
       value = value + 1;
    } /*-10.5 … 8.5 -> f */

    for( i = 0; i < 20; i++)
        printf("%.2f\t%u\n",
              f[i], p[i]);
}
```

-10.50	3240624128
-9.50	3239575552
-8.50	3238526976
-7.50	3236954112
-6.50	3234856960
-5.50	3232759808
-4.50	3230662656
-3.50	3227516928
-2.50	3223322624
-1.50	3217031168
-0.50	3204448256
0.50	1056964608
1.50	1069547520
2.50	1075838976
3.50	1080033280
4.50	1083179008
5.50	1085276160
6.50	1087373312
7.50	1089470464
8.50	1091043328

Numbers sorted properly

Radix-sort float point numbers (method 1)

You need to know how to extract bits correctly.

1. Radix-sort all numbers based on all 32 bits

2. Reverse the order of negative numbers

3. Put all negative numbers before positive numbers.

Radix-sort float point numbers (method 2)

Similar to method 1 of radix sorting a mixture of positive and negative integers. But you need to know how to extract bits correctly.

1. Separate positive numbers and negative numbers.
2. Radix-sort positive numbers in ascending order based on low 31 bits
3. Radix-sort negative numbers in descending order based on low 31 bits
4. Join positive numbers and negative numbers.

Other info about radix sort

- Radix sort was first used in 1890 U.S. census by Hollerith
- Used to sort numbers or texts
- Very efficient when sorting a large number of elements
  - O(M*N). M: length of each elements; N: number of elements
- Fixed size buckets make it consume more space than other sorting algorithms
  - E.g., bubble sort is in-place soring.

Radix sort for any radix values

- Radix = “The base of a number system” (Webster’s dictionary)
- Radix is another term of “base” : number of unique digits, including the digit zero, used to represent numbers
- Radix of numbers:
  - Binary numbers have a radix of 2
  - decimals have a radix of 10
  - hexadecimals have a radix of 16.
- Radix of texts:
  - 26 if only capital letters are considered
  - 36 if capital letters and decimal digits are considered
  - 62 for capital letters + small letters + decimal digits

Radix sort of decimal numbers

Values to be sorted 126, 328, 636, 341, 416, 131, 328

- Sort based on on lower digit:

341, 131, 126, 636, 416, 328, 328

- Sort the result based on next-higher digit:

416, 126, 328, 328, 131, 636, 341

- Sort the result based on highest digit:

126, 131, 328, 328, 341, 416, 636

RadixSorting Strings

- Single characters can be Bucket-Sorted
- Break strings into characters
- Append NULLs to short strings
- Start from the last character, end with the first character.

	5^{th} pass	4^{th} pass	3^{rd} pass	2^{nd} pass	1^{st} pass
String 1	z	i	p	p	y
String 2	z	a	p		
String 3	a	n	t	s	
String 4	f	l	a	p	s

NULLs are treated as character with ASCII code equal to 0

Radix and bit masks when sorting binary data

- Values to be sorted: 126, 328, 636, 341, 416, 131, 328
  - Binary numbers: (0 001 111 110, 0 101 001 000, 1 001 111 100, 0 101 010 101, 0 110 100 000, 0 010 000 011, 0 101 001 000)
  - Octal numbers: (0176, 0510, 1174, 0525, 0640, 0203, 0510)
  - Hexadecimal numbers: (07E, 148, 27C, 1A0, 083, 148)
- Selection is a trade-off between time complexity and space complexity.
- For the above examples
  - how many buckets are needed?
  - how many passes must be made?
  - How to generate masks and how to determine bucket index?

Mask=0xF<<(pass*4)

Bucket_index = (Value & Mask)>>(pass*4)

Radix sort algorithm in a general form

```
radix_sort(A, n, k) {
   /* A: array; n: number of items; k: number of digits */
   create buckets  (buckets can be arrays or lists)
    for (d = 0; d <k; d++) {
         /* sort A using digit position d as the key. */
         for (i = 0; i<n; i++) {
                  p = the d-th digit  (from right) of A[i]
                  Add A[i] to bucket p
        }
       A = Join the buckets
   }
}
```

Order is important

1. Control the passes to move from the least significant part to the most significant part.
2. Enforce the same order when selecting items to move buckets, organizing the items in buckets, and joining the buckets.

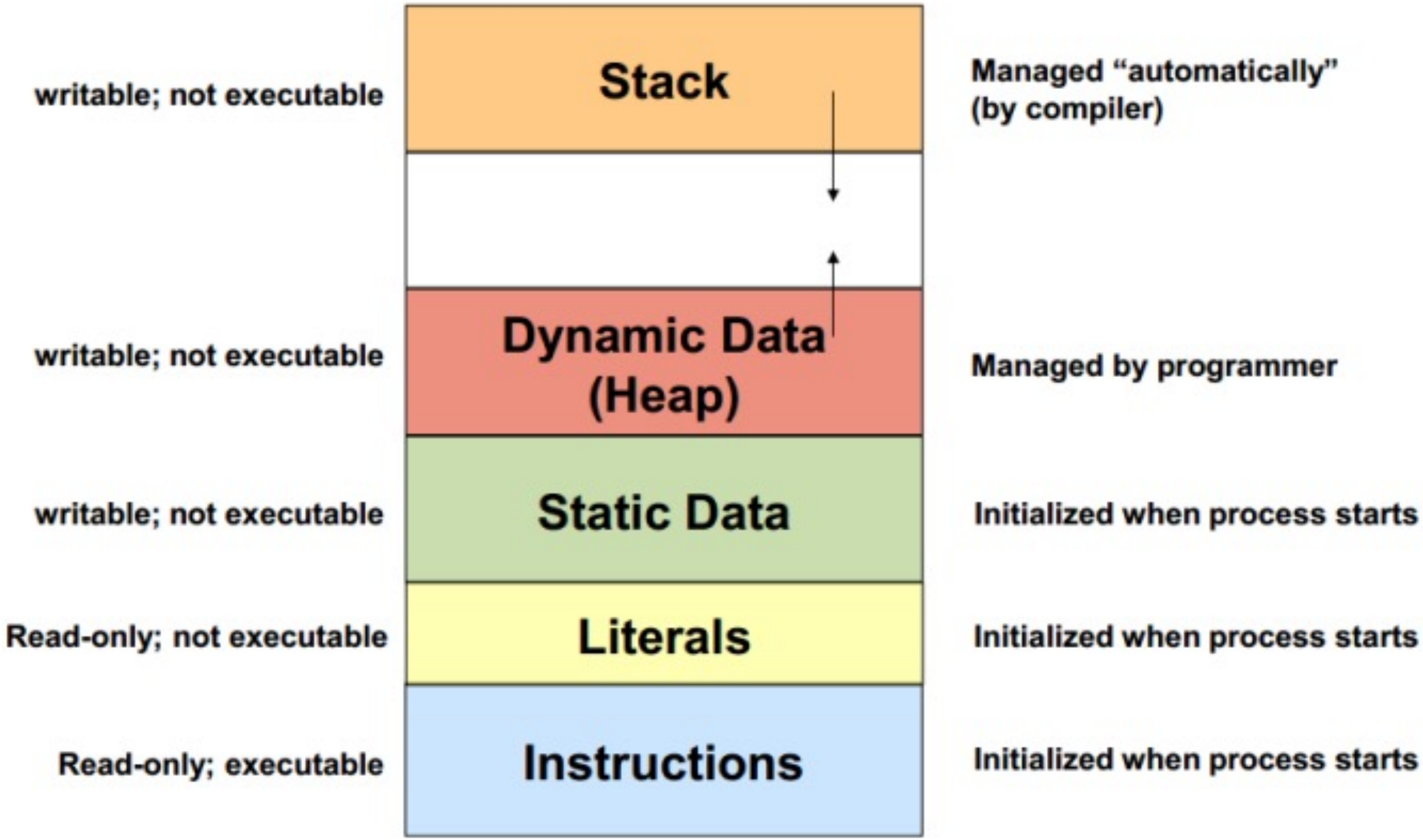
Not magic. It provably works.

- Elements: N-digit numbers, base B
- Claim: after i^{th} sorting, least significant i digits are sorted.
  - e.g. B=2, i=2, elements are 101 and 011. 1<u>01</u> comes before 0<u>11</u> for last 2 bits.
- Proof using induction:
  - base case: i=1. 1 digit is sorted (that wasn't hard!)
  - Induction step
    - assume for i, prove for $i+1$.
    - consider two numbers: X, Y. Say X_i is i^{th} digit of X (from the right)
      - Values are more determined by a higher digit than any lower digits, i.e.,
      - $X_{i+1} > Y_{i+1}$ then $i+1^{th}$ sorting will put them in order
      - $X_{i+1} < Y_{i+1}$, same thing
      - $X_{i+1} = Y_{i+1}$, order depends on last i digits. Induction hypothesis says already sorted for these digits.

CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

Organization of memory space



Understand memory space with a C prog

```
#include <stdio.h>
#include <stdlib.h>
char global_data[]="This is in heap";
int depth=0;
void func() {
   char func_data[20];
   if (depth++>5) return;
   sprintf(func_data, "*#*#*#*#* %d *#*#*#*#", depth);
   printf("func_data (layer %d)@ %p\n",depth,(void *)func_data);
   func();
}
main(int argc, char **argv) {
   char main_data[20],*dynamic_alloc_data;
   strcpy(main_data, "#$#$#$#$#$#$#$#$");
   dynamic_alloc_data=(char *)malloc(50);
   strcpy(dynamic_alloc_data, "Text in allocated mem. space");
   printf("Code: main @ %p, func @ %p\n", (void *)main,(void *)func);
   printf("global_data @ %p\n", (void *)global_data);
   printf("dynamic_alloc_data @ %p\n", (void *)dynamic_alloc_data);
   func();
}
```

Execution results:

```
Code: main @ 0x55a2c31d5788, func @ 0x55a2c31d56fa
global_data @ 0x55a2c33d6010
dynamic_alloc_data @ 0x55a2c3eb9260
func_data (layer 1)@ 0x7ffc08380bd0
func_data (layer 2)@ 0x7ffc08380ba0
func_data (layer 3)@ 0x7ffc08380b70
func_data (layer 4)@ 0x7ffc08380b40
func_data (layer 5)@ 0x7ffc08380b10
func_data (layer 6)@ 0x7ffc08380ae0
```

Instructions

Heap growing from low mem address to high mem address

Stack growing from high mem address to low mem address

Examine code, heap data, and stack using gdb

- Check the code (e.g., `disassemble main`, `disassemble func`)
- Locate and examine the data in memory (e.g., `x/32cb func_data`)
- Monitor the growth of stack (e.g., `x/256cb $sp`)

Basic pointer concepts and pointer operations

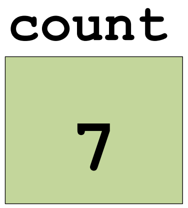
pointers and array, passing pointers to a function, pointers and strings, strtok

Pointers overview

- Pointers save addresses.
- With a pointer, you can locate and access the corresponding data.
- The type of the pointer (e.g., int *, float *) determines how many bytes are interpreted together and how to interpret the data.
- By changing the address in a pointer, you can locate and access other data.
- What you can do with pointers?
  - Controlling the way in which data is interpreted
  - Sharing data by passing pointers (addresses) instead of data
  - Dynamically organizing data into different structures

Normal variables and pointer variables

- Pointer variables
  - Contain memory addresses as their values
  - A normal variable contains a specific value
    - You can consider a variable name as an "alias" of a memory address
  - A pointer variable contains the *address* of another variable
    - The pointer variable is an "alias" of a memory address, in which another memory address is stored.



Pointer variable declarations

- **`*`** used with pointer variables

```
int *myPtr;
/* Declares a pointer to an int
 * Pointer type:  int *
 */
```

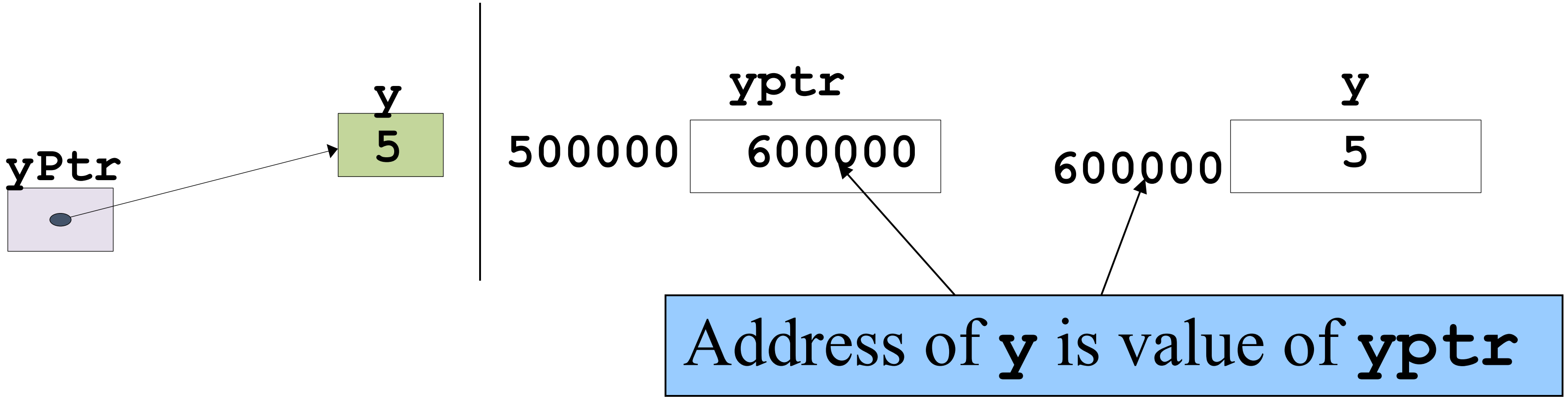
- Multiple pointers, multiple **`*`**

```
int *myPtr1, *myPtr2;
```

- Can declare pointers to any data type, even pointer to point
- Initialize pointers to **`0`**, **`NULL`**, or an address
  - **`0`** or **`NULL`** - points to nothing (**`NULL`** preferred)

& (address operator) returns address of operand

```
int y = 5;
int *yPtr, *yPtr2;
yPtr = &y;
/*yPtr gets address of y, i.e., yPtr "points to " y */
```




```
yPtr2 = yPtr; /*yPtr2 points to y too. */
```

***** (indirection/dereferencing operator)

Returns a synonym/alias of what a pointer *points* to

`yptr` returns the address of **`y`**

`*yptr` returns **`y`** (because **`yptr`** points to **`y`**)

- ***** can be used for assignment

```
*yptr = 7; // changes y to 7
```

- ***** can only be used to dereference pointer variables.

```
*(0x55a2c31d5788)  //invalid
```

`*` and `&` are inverses

`*` and `&` cancel each other out

```
*&yptr  ->  * (&yptr) -> * (address of yptr)->
 returns alias of what operand points to -> yptr
```

```
&*yptr -> &(*yptr) -> &(y) -> returns address of y,
 which is yptr -> yptr
```

```c
/* Using the & and * operators */
#include<stdio.h>

int main(){
int a;        /* a is an integer */
int *aPtr; /* aPtr is a pointer to an integer */

a = 7;
aPtr = &a; /* aPtr set to address of a */

printf("The address of a is %p\n"
  "The value of aPtr is %p", &a, aPtr);

printf("\nThe value of a is %d\n"
  "The value of *aPtr is %d", a, *aPtr);

printf("\n* and & are inverses\n"
  "&*aPtr = %p, *&aPtr = %p\n",
  &*aPtr, *&aPtr);

return 0;
}
```

The address of **a** is the value of **aPtr**.

The ***** operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

Notice how ***** and **&** are inverses

```
The address of a is 0012FF88
The value of aPtr is 0012FF88
The value of a is 7
The value of *aPtr is 7
* and & are inverses
&*aPtr = 0012FF88, *&aPtr = 0012FF88
```

Typecasting using a pointer

Change the type of the pointer to change the way the data is interpreted

```
main(){
 float f=123.45;
 unsigned int *p = (unsigned int *) &f, i, j, value_of_bits4to7;
 i = *p; j=(int)f; value_of_bits4to7 = (*p & 0xF0)>>4;
 printf("%d %d %d\n", i, j, value_of_bits4to7);
}
```

j
123

f
123.45

i
1123477094

p
0x7ffffffe3f8

0100 0010 1111 0110 1110 0110 **0110** 0110

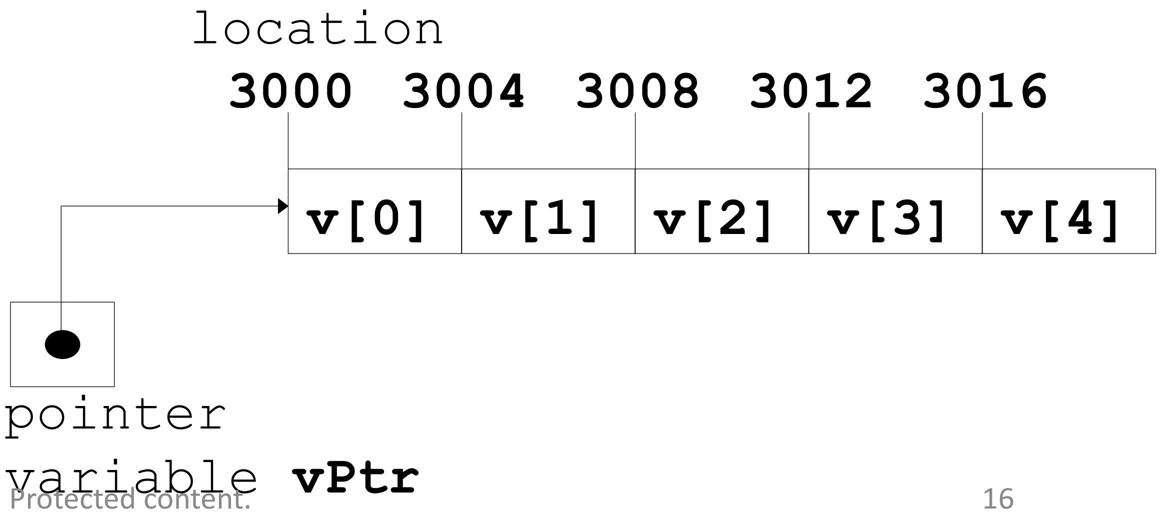
value of bits4to7
6

Two types of type casting

```
#include <stdio.h>
void main(){
  float f=123.45;
  unsigned int i, j, *p;

  /*1st*/
  i=(int)f;

  /*2nd*/
  p=(int *)&f;
  j=*p;

  /* output: 123 1123477094 */
  printf("%d %d\n", i, j);
}
```

Pointer expressions and pointer arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (**`++`** or **`--`**)
  - Add an integer to a pointer(**`+`** or **`+=`** , **`-`** or **`-=`**)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array
- 5-element **`int`** array on machine with 4-byte **`int`**s
  - **`vPtr`** points to first element **`v[0]`** at location **`3000`**. (**`vPtr = v`**)
  - **`vPtr +=2;`** sets **`vPtr`** to **`3008`**
    - **`vPtr`** points to **`v[2]`** (incremented by 2), but machine has 4 byte **`int`**s.



Pointer expressions and pointer arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other.

```
vPtr2 = &v[2];
vPtr = &v[0];
vPtr2 - vPtr == 2.
```

- Pointer comparison (**<**, **==** , **>**)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to **0**
- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used

```
int *ptr1 = &b;
char *ptr2= (char *)ptr1;
char c=*ptr2;
```

Relationship between pointers and arrays

- Array variables and pointers can be used interchangeably in most cases.
  - Array variables save starting addresses of the arrays.
- Pointers can do array subscripting operations

  Declare an array **`b[5]`** and a pointer **`bPtr`**

  `bPtr = b; //`Array name is actually a address of first element

  OR

  `bPtr = &b[0]; //`Explicitly assign **`bPtr`** to address of first element

  Element **`b[n]`** can be accessed by **`*( bPtr + n )`**

  Array itself can use pointer arithmetic.

  `b[3]` same as **`*(b + 3)`**

  Pointers can be subscripted (pointer/subscript notation)

  `bPtr[3]` same as **`b[3]`**
- You can also malloc some memory pointed by a pointer and use it as an array (will introduce later).

Relationship between pointers and arrays

- Array variables are constant pointers and are attached with array size info
  - Array variables save starting addresses of the arrays, and cannot be changed.
  - Array variables cannot be changed
  - sizeof() returns different values.
- **`sizeof()`** returns size of operand in bytes
  - Can be used with variable names (e.g., sizeof(a)), type name (e.g., sizeof(int)), and constant values (e.g., sizeof("hello world!\n").
  - Return value is in `unsigned long` type.
  - For arrays: size of 1 element * number of elements

```
int myArray[10], *p=myArray;
printf("%lu, %lu", sizeof(myArray), sizeof(p)); /* print 40,8 */
```

Memory allocation

- Header file: <stdlib.h>
- `void * malloc(size_t esize)` -- allocate a single block of memory of `esize` bytes
- `void * calloc(size_t num, size_t esize)` -- allocate a block of memory of `num*esize` bytes
- `void * realloc(void * ptr, size_t esize)` -- extend the amount of space (pointed by `ptr`) allocated previously to `esize`
- Returns `void *` if succeed (cast the result to an appropriate type before use).
- Returns NULL if not enough memory available.
- If realloc() cannot extend the current memory block (ptr), it allocates memory from a new location, copies over the data, and frees up the memory pointed by ptr.
- `void free(void *ptr)`

memory pointed by `ptr` is no longer needed. Memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up.

```
float *nums;
int N;
int I;

printf("Read how many numbers:");
scanf("%d",&N);
nums = (float *) calloc(N, sizeof(float));
/* nums is now an array of floats of size N */
for (I = 0; I < N; I++) {
  printf("Please enter number %d: ",I+1);
  scanf("%f",&(nums[I]));
}
/* Calculate average, etc. */

…
```

Allocated with calloc() and used like an array.

```
float *nums;
int I;

nums = (float *) calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */

for (I = 0; I < 5; I++)  nums[I] = 2.0 * I;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */

nums = (float *) realloc(nums,10 * sizeof(float));
/* An array of 10 floating point values is allocated, the
 first 5 floats from the old nums are copied as the first 5
 floats of the new nums, then the old nums is released */
```

Releasing memory (free)

```
void free(void *ptr)
```

- memory at location pointed to by ptr is released (so we could use it again in the future)
- program keeps track of each piece of memory allocated by where that memory starts
- if we free a piece of memory allocated with calloc, the entire array is freed (released)
- results are problematic if we pass as address to free an address of something that was not allocated dynamically (or has already been freed)

Suggested practice: to free the memory in the function where it is allocated

```
void problem() {
  float *nums;  int N = 5;

  nums = (float *) calloc(N, sizeof(float));

  /* But no call to free with nums */
}
```

- When function problem called, space for array of size N allocated.
- When function ends, variable nums goes away, but the space nums points at (the array of size N) does not.
- There is no way to figure out where the space is.
- This problem is called *memory leak.*

Strings are arrays of characters ended with NULL

"abcd" is actually "abcd\0"

```
char str[]="abcd", *p=str;
sizeof("abcd");     /* returns 5 */
strlen("abcd");     /*  returns 4 */
p[0]=='a';          /* returns 1 (true) */
p[4]==0;            /*returns 1 (true)*/
scanf("%s",str);   /* not &str */
```

Is a NULL pointer an empty string?

Passing addresses between functions

- Many library functions use pointers.
- Most useful when you want to
  - **pass an array or a string:** passing an address is more efficient than copying all data; passing all data using one argument;..
  - **have some data updated in a function**, e.g., sorting an array
- Use `*` operators for pointer arguments when defining the function
- Use pointer arguments when calling the function
  - Pass address of argument using `&` operator (e.g., `cube(&mynumber)`)
    - Passing non-pointers (e.g., `cube(2)`) may cause segmentation faults.
  - Arrays are not passed with `&` because the array name is already a pointer

```c
#include <stdio.h>
void cube(int *);

int main(){
  int number = 5;
  printf("Original value of number: %d\n", number);
  cube(&number);
  printf("New value of number: %d\n", number);
  return 0;
}

void cube(int *nPtr ){
*nPtr = (*nPtr) * (*nPtr) * (*nPtr);
}
```

Address of **number** is given because **cube** expects a pointer.

Inside **cube**, ***nPtr** is used (***nPtr** is **number**).

Output:

```
Original value of number: 5
New value of number: 125
```

Bubble-sort using pointers

- bubbleSort() sorts elements in place
- Swap() swaps two array elements

```
#include <stdio.h>

#define SIZE 10
void swap(int *, int *);
void bubbleSort(int *, const int);

int main() {
  int a[SIZE]={2,6,4,8,10,12,89,68,45,37 };
  int i;
  printf("Data items in original order\n" );
  for ( i = 0; i < SIZE; i++)
    printf( "%4d", a[ i ] );
  bubbleSort(a, SIZE);
  printf("\nData items in ascending order\n" );
  for ( i = 0; i < SIZE; i++)
    printf( "%4d", a[ i ] );
  printf( "\n" );
  return 0;
}
```

- **`Bubblesort`** gets passed the address of array **`a`** (pointer).
- When passing an array of values, array size must also be passed.

```
void bubbleSort(int *array, const int size) {
  int pass, j;
  for ( pass = 0; pass < size - 1; pass++ )
    for ( j = 0; j < size - 1; j++ )
      if ( array[ j ] > array[ j + 1 ])
        swap( &array[ j ], &array[ j + 1 ] );
}

void swap(int *element1Ptr, int *element2Ptr) {
  int hold = *element1Ptr;
  *element1Ptr = *element2Ptr;
  *element2Ptr =hold;
}
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Strtok(): splitting a string into tokens

```
#include <string.h>
char * strtok (char *string, const char *delimiters)
```

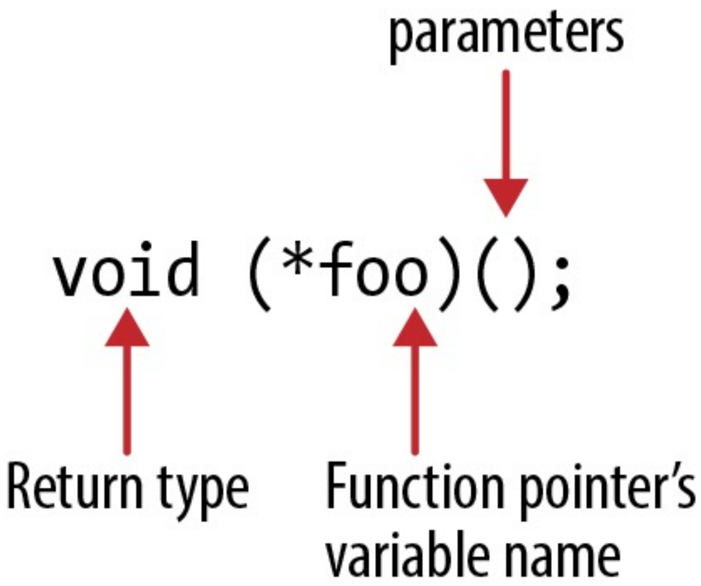
- A string can be split into tokens by making **a series of strtok calls.**
  - return a token on each call.
  - The searching begins at the next character after the token previously found.
  - Return NULL when no other tokens can be found (string end is reached or string contains only delimiters)
  - Contents in `string` may be changed (delimiters replaced with NULL, 1 string to multiple)
- On the first call, the ***string* argument** specifies the string to be split up.
- Subsequent calls, the ***string* argument** must be **null.**
  - If it is not NULL, the searching and splitting will restart from the beginning of the string.
- *delimiters* specifies a set of delimiters (no need to be same in a series of strtok calls)

```c
#include <string.h>
#include <stdio.h>
int main(){
   char address[] =
      "tom@www.auckland.ac.nz:/home/tom/fall2020/cs288/";
   char delimiter[] = ".@:/";
   char *token;
   /* get the first part */
   token = strtok(address, delimiter);
   /* get the rest */
   while( token != NULL ) {
      printf( "%s\n", token );
      token = strtok(NULL, delimiter);
   }
   return(0);
}
```

```
tom
www
auckland
ac
nz
home
tom
fall2020
cs288
```

Function pointers

- A function pointer is a pointer that holds the address of a function.
  - Function name is starting address of the function.
- A important and useful feature in C.
  - Your program can dynamically change which function is to be called.
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers
- Declare a function pointer
  - Similar to declaring a function
  - var name and * in ()




```
int (*f1)(double);       //Passed a double and
                         // returns an int
void (*f2)(char*);       // Passed a pointer to char
                         // and  returns void
double* (*f3)(int, int);  // Passed two integers
                       //returns a pointer to a double
```

Some examples for function pointers.

```
int   *f4();     // a function returns int
int   (*f5)();   // a function pointer returns int
int*  (*f6)();   // a function pointer returns int *
```

```
#include <stdio.h>
int (*fptr1)(int);
int square(int num) {
    return num*num;
}
main(){
    int n = 5;
    fptr1 = square;
    printf("%d squared is %d\n",n, fptr1(n));
}
```

Passing function pointers (using bubblesort as an example)

- Function **`bubble`** takes a function pointer pointing to a helper function
  - **`bubble`** calls this helper function, which determines ascending or descending sorting
- The argument in **`bubble`** for the function pointer:

  `int ( *compare )( int, int )`

  tells **`bubblesort`** to expect a pointer to a function that takes two **`int`**s and returns an **`int`**.
- If the parentheses were left out: **`int *compare( int, int ),`** it is to declares a function that receives two integers and **returns a pointer** to a **`int`**

```c
#include <stdio.h>
#define SIZE 10
int ascending( int a, int b ){
  return b < a;
}
int descending( int a, int b ){
  return b > a;
}
void swap( int *element1Ptr, int *element2Ptr){
  int temp = *element1Ptr;
  *element1Ptr = *element2Ptr;
  *element2Ptr = temp;
}
void bubble(int work[], int size, int (*compare)(int, int ))
{
  int pass, count;
  for ( pass = 1; pass < size; pass++)
    for ( count = 0; count < size - 1; count++ )
      if ( (*compare)(work[count], work[count+1]) )
        swap( &work[count], &work[count+1] );
}
```

```c
int main(){
  int order, counter,
  a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
  printf( "Enter 1 to sort in ascending order, \n"
          "Enter 2 to sort in descending order: ");
  scanf( "%d", &order );
  printf( "\nData items in original order \n" );
  for ( counter = 0; counter < SIZE; counter++)
    printf( "%5d", a[ counter ] );
  if ( order == 1 ){
    bubble( a, SIZE, ascending );
    printf( "\nData items in ascending order\n" );
  }
  else{
     bubble( a, SIZE, descending );
     printf( "\nData items in descending order\n" );
  }

  for ( counter = 0; counter < SIZE; counter++)
    printf( "%5d", a[ counter ] );
  printf( "\n" );
}
```

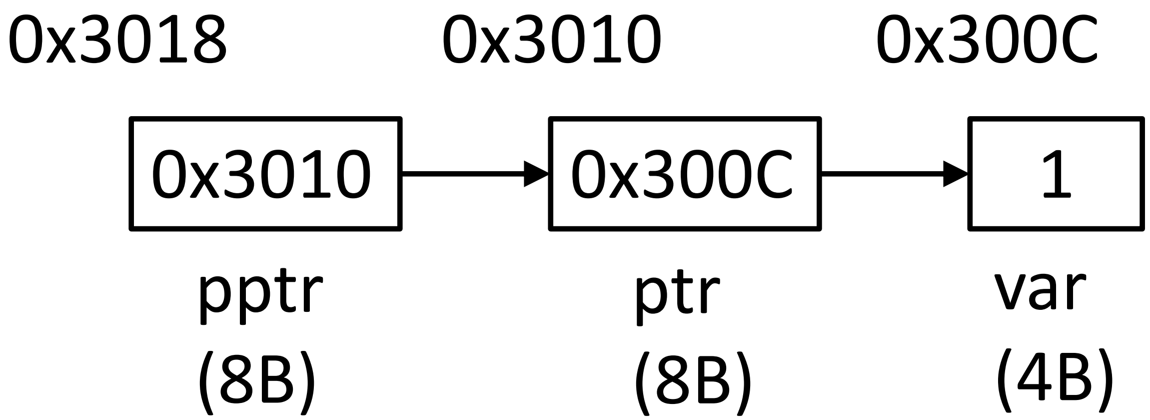
Pointer to pointer and (dynamic) multidimensional arrays

Pointer to pointer, array of pointers and dynamic multi-dimensional arrays, parsing command-line arguments and environment variables,

function pointers

Pointer to pointer: memory address of pointer variable

```
#include <stdio.h>
int main () {
    int  var = 1;
    int  *ptr = &var;
    int  **pptr = &ptr;

    printf("Value of var = %d\n", var );
    printf("Value available at *ptr = %d\n", *ptr );
    printf("Value available at **pptr = %d\n", **pptr);
    return 0;
}
```




```
Value of var = 1
Value available at *ptr = 1
Value available at **pptr = 1
```

- Defined using ** (2nd * denotes that it is a pointer; 1st * denotes that the data pointed by it is pointer).
- Saves the address of a pointer variable.
- Dereferenced using **

Pointer vs. pointer to pointer

- A pointer to pointer is also a pointer
  - A pointer to pointer (e.g., pptr) saves a memory address, as a normal pointer (e.g., ptr) does.
- Different types determine different ways to interpret the data (1s and 0s).
  - e.g., *pptr and *ptr are interpreted differently.
  - Type of *pptr is int *, it is an "alias" of ptr.
  - Type of *ptr is int, it is an "alias" of var.
  - Type of **pptr also int, it is an "alias" of var.

A "weird" program.

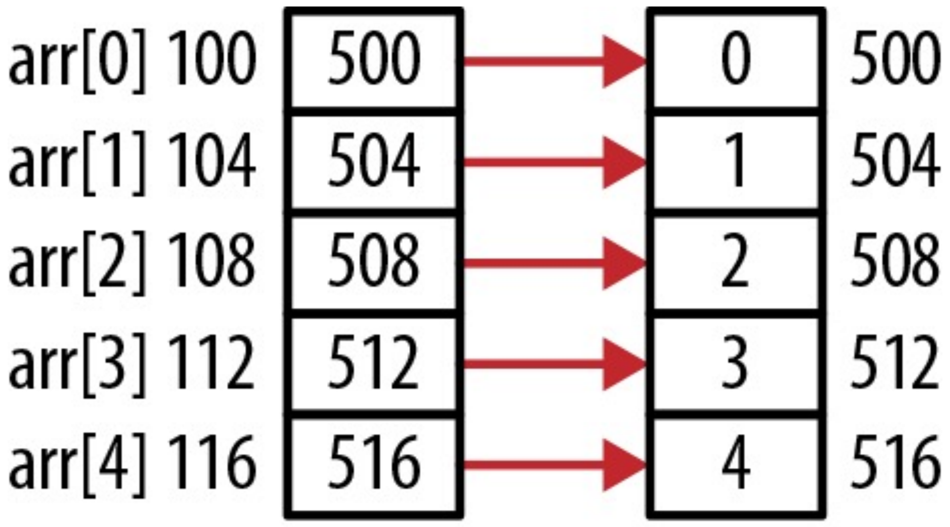
This program is only for help you understand pointer to pointer. It is not a good way to use pointers to pointer.

```
#include <stdio.h>
int main () {
   int  var = 1;
   int  *ptr = &var;
   int  **pptr = &var;

   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr);
   printf("Value available at *pptr = %lu\n", * ((int *)pptr));
   return 0;
}
```

```
Value of var = 1
Value available at *ptr = 1
Value available at *pptr = 1
```

Array of pointers

```
#include <stdio.h>
int main () {
  int* arr[5];
  for(int i=0; i<5; i++) {
    arr[i] = (int*)malloc(sizeof(int));
    *arr[i] = i;
  }
  return 0;
}
```




```
#include <stdio.h>
int main () {
  int* arr[5];
  for(int i=0; i<5; i++) {
    *(arr+i)=(int*)malloc(sizeof(int));
    **(arr+i)=i;
  }
  return 0;
}
```

What are these values?

*arr[0]

**arr

**(arr+1)

arr[0][0]

arr[3][0]

Multi-dimensional arrays

```c
#include <stdio.h>
int main () {
 int matrix[2][5] = {
          {1,2,3,4,5},
          {6,7,8,9,10}};

 for(int i=0; i<2; i++) {
   for(int j=0; j<5; j++) {
     printf("matrix[%d][%d] "
    "Address: %p Value: %d\n",
        i, j, &matrix[i][j],
            matrix[i][j]);
    }
  }
}
```

matrix[0][0]	100	1
matrix[0][1]	104	2
matrix[0][2]	108	3
matrix[0][3]	112	4
matrix[0][4]	116	5
matrix[1][0]	120	6
matrix[1][1]	124	7
matrix[1][2]	128	8
matrix[1][3]	132	9
matrix[1][4]	136	10

```
matrix[0][0] Address: 100 Value: 1
matrix[0][1] Address: 104 Value: 2
matrix[0][2] Address: 108 Value: 3
matrix[0][3] Address: 112 Value: 4
matrix[0][4] Address: 116 Value: 5
matrix[1][0] Address: 120 Value: 6
matrix[1][1] Address: 124 Value: 7
matrix[1][2] Address: 128 Value: 8
matrix[1][3] Address: 132 Value: 9
matrix[1][4] Address: 136 Value: 10
```

- Elements in a multi-dimensional array are saved contiguously in memory.
- Rows/columns must have the same number of elements
- Address of matrix[i][j]=starting address of matrix + i *size_of_row+j*size_of_element.

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?

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

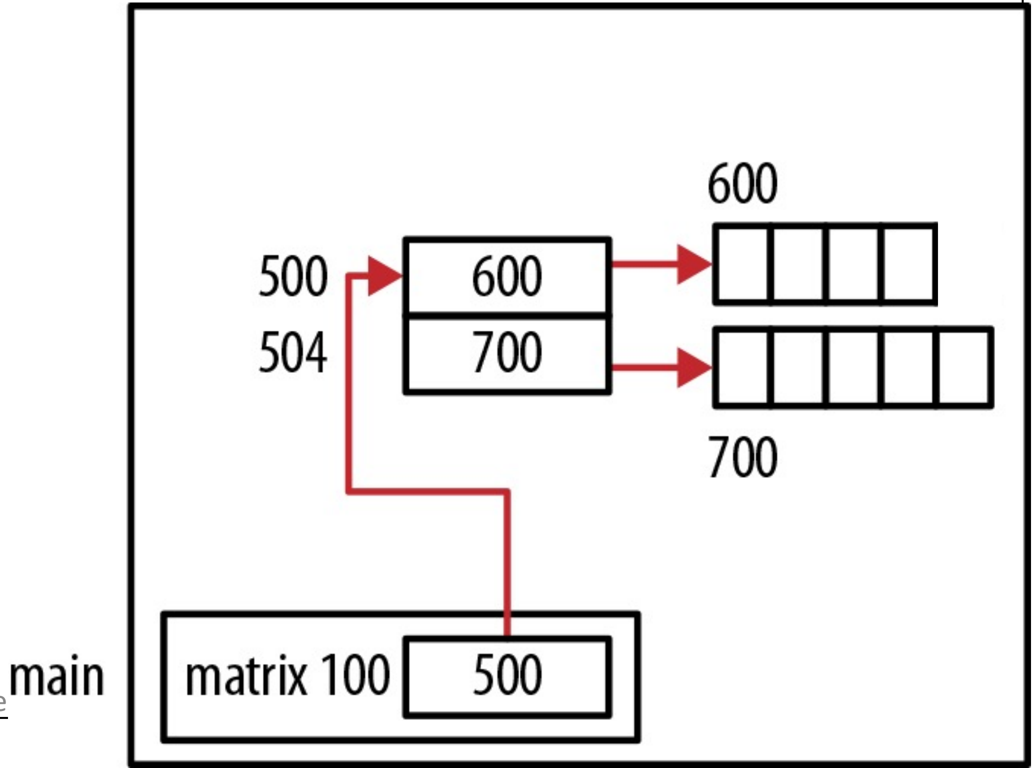
```
#include <stdio.h>
#include <stdlib.h>

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");
}
```

Dynamic multi-dimensional array

- Elements in a multi-dimensional array are saved contiguously in memory.
  - Rows and columns cannot be expanded dynamically.
- Rows/columns in a multi-dimensional array must have the same number of elements.
  - Array cannot be jagged
- What if we want to have more flexibility?
- Create dynamic multi-dimensional array using array of pointers.
  - Typical example is argv parameter of main().

Dynamic multi-dimensional array

```
#include <stdio.h>
int main () {
 int rows=2, min_columns=4, val=1;
 int **matrix = (int **) malloc(rows * sizeof(int *));
 for (int i = 0; i < rows; i++)
   matrix[i] = (int *) malloc((min_columns + i) * sizeof(int));

 for(int i=0; i<2; i++) {
   for(int j=0; j<4+i; j++) {
     matrix[i][j] = val++;
    }
 }
 free(matrix[0]);
 free(matrix[1]);
 free(matrix);
}
```

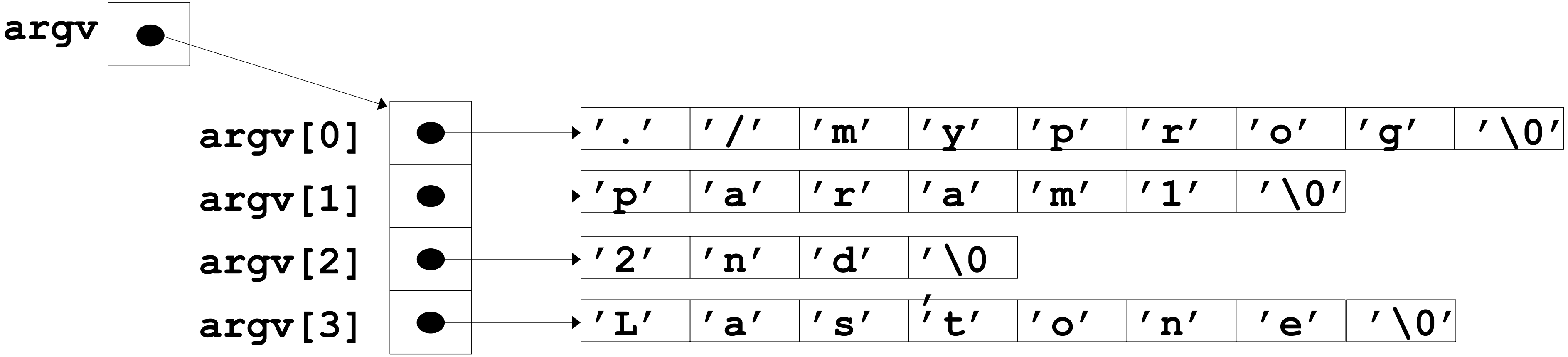


Processing arguments

```
int main(
  int argc,  // specifies # in argv[]
  char * argv[]); // list of parameters

int main(int argc, char **argv);
```

- for argv[], an ancillary data structure is provided: argc
- argc pointers pointing to argc strings, each of which is an argument

./myprog   parameter1   2nd   Lastone



An example: reverse-print command line args

```
// output all command line arguments in reverse order
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char * argv[] ) {
    printf( "%d command line args passed.\n", argc );
    while( --argc > 0 ) { // pre-decrement skips argv[0]
      printf( "arg %d = \"%s\"\n", argc, argv[argc] );
    }
}
```

```
$ ./myprog 3 r 55 ""
5 command line args passed.
arg 4 = ""
arg 3 = "55"
arg 2 = "r"
arg 1 = "3"
```

Parsing command line arguments needs much more work (will introduce later).

What is argv[0]?

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

int main(int argc, char **argv){
  printf("mem. addr. of argc: %p\n", &argc);
  printf("mem. addr. of argv: %p\n", &argv);
  printf("mem. addr. of argv[0]: %p\n", argv);
  printf("mem. addr. in argv[0]: %p\n", argv[0]);
  printf("1st char %c in argv[0]:\n", argv[0][0]);
  return 0;
}
```

```
mem. addr. of argc: 0x7fff91892a0c
mem. addr. of argv: 0x7fff91892a00
mem. addr. of argv[0]: 0x7fff91892af8
mem. addr. in argv[0]: 0x7fff91893774
1st char . in argv[0]:
```

Check how arguments are saved in memory using gdb. Example gdb commands

```
break 10
run param1 param2 param3
x/8xg argv
x/64cb argv[0]
```

POSIX argument rules (IEEE Std 1003.1-2017 Chap 12)

- Followed by most Unix/Linux programs
  - http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html
- General format:
  - utility_name [-a] [-b] [-c option_argument] [-d|-e] [-f [option_argument]] [operand...]
- Three types of arguments
  - Options; option arguments, operand
- Examples
  - ls -l -t -r
  - ls -ltr
  - head -n 5 /etc/passwd
  - rm -f ~/a.tmp
  - gcc -o myprog myprog.c

POSIX argument rules (IEEE Std 1003.1-2017 Chap 12)

- **options**: arguments that consist of '-' characters and single letters or digits
  - The character after '-' is an option character.
  - Every command/tool has a different set of options.
  - Options supported and their meanings are hard-coded in a program
  - The same option may have different meanings in different commands/tools.
    - e.g., -f may means "file", "force" in rm, or "fields" in cut
  - Several options can be combined and put in a single argument
    - e.g., `ls -l -t -r` is the same as `ls -ltr`
  - The order of different options relative to one another should not matter.
    - e.g., `ls -l -t -r` is the same as `ls -t -r -l`
- **Option arguments**: arguments shown separated from their options by <blank> characters
  - when an option-argument is enclosed in the '[' and ']' notation in command line description, it is optional
  - Some options have option arguments, and some do not have.
- **Operands**: arguments other than options and option arguments
  - The order of operands may matter and position-related interpretations should be determined by the program.

```c
/* Parsing command line */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int arg;
    for(arg = 1; arg < argc; arg++) {
        if(argv[arg][0] == '-')
            printf("option: %s\n", argv[arg]+1);
        else
            printf("argument %d: %s\n", arg, argv[arg]);
    }
    exit(0);
}
```

```
%./args -i -lr  'hi there' -f fred.c
option: i
option: lr
argument 3: hi there
option: f
argument 5: fred.c
```

Not easy to extend when a program supports complex options.

Parsing command line arguments using getopt()

```
#include <unistd.h>
int getopt(int argc, char *const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

- The ***getopt*()** function parses the command line arguments.
  - Mainly used to process options and option arguments.
  - Need to be called repeated.
    - Return one option each time called. ***Optarg*** points to the corresponding option argument
- ***optstring*** is a string summarizing the legitimate option characters.
  - If an option character is followed by a colon, the option requires an option argument.
  - ":" being first character has special meaning (next page).
- External variable, ***optind*** is set to the index of the next argument to be process.
- ***operands***
  - arguments in *argv[]* are permuted with all operands are moved to the end, starting at ***argv[optind]***.

Parsing command line arguments using `getopt()`

Possible getopt() return values

- ***-1*** for the end of the option list
- A positive value: **the value is the ASCII code of a character**, which may be
  - ***An option character*** in optstring when the option is found successfully, and
    - the option does not need an option argument, or
    - the option needs an option argument, and the option argument is found
      - ***optarg*** saves the actual option argument
  - '**?**' for an unknown option character, ***optopt*** stores the actual option
  - '**?**' when option argument is missing for an option and first character in optstring is NOT '**:**'
  - '**:**' when option argument is missing for an option and first character in optstring is '**:**'
- getopt() stops scanning when it sees long options started with "--" (e.g., ls --all).
  - use getopt_long() to process long options.

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

int main(int argc, char *argv[])
{
  int opt;
  while((opt=getopt(argc,argv,“:if:lr")) !=-1){
    switch(opt) {
      case 'i':
      case 'l':
      case 'r':
        printf("option: %c\n", opt); break;
      case 'f':
        printf("filename: %s\n", optarg); break;
      case ':':
        printf("option %c needs a value\n", optopt); break;
      case '?':
        printf("unknown option: %c\n", optopt); break;
    }
  }
  for(; optind < argc; optind++)
    printf("argument: %s\n", argv[optind]);
  exit(0);
}
```

```
%./arg -i -lr 'hi there' -f fred.c -q
option: i
option: l
option: r
filename: fred.c
unknown option: q
argument: hi there
```

```
%./arg -i -lr 'hi there' -f
option: i
option: l
option: r
option f needs a value
argument: hi there
```

```
%./arg -i 'hi there' -f -q
option: i
filename: -q
argument: hi there
```

Processing environmental variables

```
int main( int argc,  char * argv[],
  char * envp[])// all environment vars
{ // main
  . . .
} //end main
```

- envp: a set of pointers, each of which points to a string.
- NULL marks the end of the list

Printing out environment variables

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

int main( int argc, char ** argv, char * envp[] )
{
  int index = 0;

  while(envp[index]){
    printf("envp[%d] = \"%s\"\n", index, envp[index]);
    index++;
  }
  printf("Number of environment vars = %d\n", index );
  exit( 0 );
}
```

Sample output

```
envp[0] = "LS_COLORS=rs=0 …"
envp[1] = "SSH_CONNECTION=…"
envp[2] = "LESSCLOSE=/usr/bin/lesspipe %s %s"
envp[3] = "LANG=en_US.UTF-8"
envp[4] = "XDG_SESSION_ID=4120"
envp[5] = "USER=ubuntu"
envp[6] = "PWD=/tmp"
envp[7] = "HOME=/home/ubuntu"
…
envp[20] = "OLDPWD=/bin"
Number of environment vars = 21
```

Library functions for handling env variables

- get an environment variable
  - char *getenv(const char *name);
- change or add an environment variable
  - int setenv(const char *name, const char *value, int overwrite);
- delete an environment variable
  - int unsetenv(const char *name);

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

int main() {
    printf("HOME = %s\n", getenv("HOME"));
}
```