Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Spring 2023

Arrays, Pointers, and Strings I

One-dimensional Arrays (1)

- Array
  - A simple variable with an index, or subscript
  - The brackets `[ ]` are used for array indexing
  - The indexing of array elements always starts from 0
  - (e.g.) `int grade0, grade1, grade2, grade3;` vs. `int grade[4];`

```
#define N   100

int a[N], i, sum = 0;

        ...

for (i = 0; i < N; i++)
    sum += a[i];
```

One-dimensional Arrays (2)

- Array initialization
  - Array may be of storage class automatic, external, or static, but NOT register
  - Arrays can be initialized using an array initializer

```
float f[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
a[100] = {0};                /* initializes all elements of a to zero */
a[] = {2, 3, 5, -7};         /* same as a[4] = {2, 3, 5, -7} */
```

  - If a list of initializers is shorter than the number of array elements, the missing elements are initialized to zero
  - external or static array: If not initialized explicitly, then all elements are initialized to zero by default
  - automatic array: not necessarily initialized

One-dimensional Arrays (3)

- Array subscripting: `a`[*expr*]
  - `a[i]` refers to `i`-th element of the array `a` (starting from 0)
  - No automatic bound check: if `i` has a value outside the range from 0 to N-1, then run-time error occurs or it can silently corrupt the other data (system dependent)

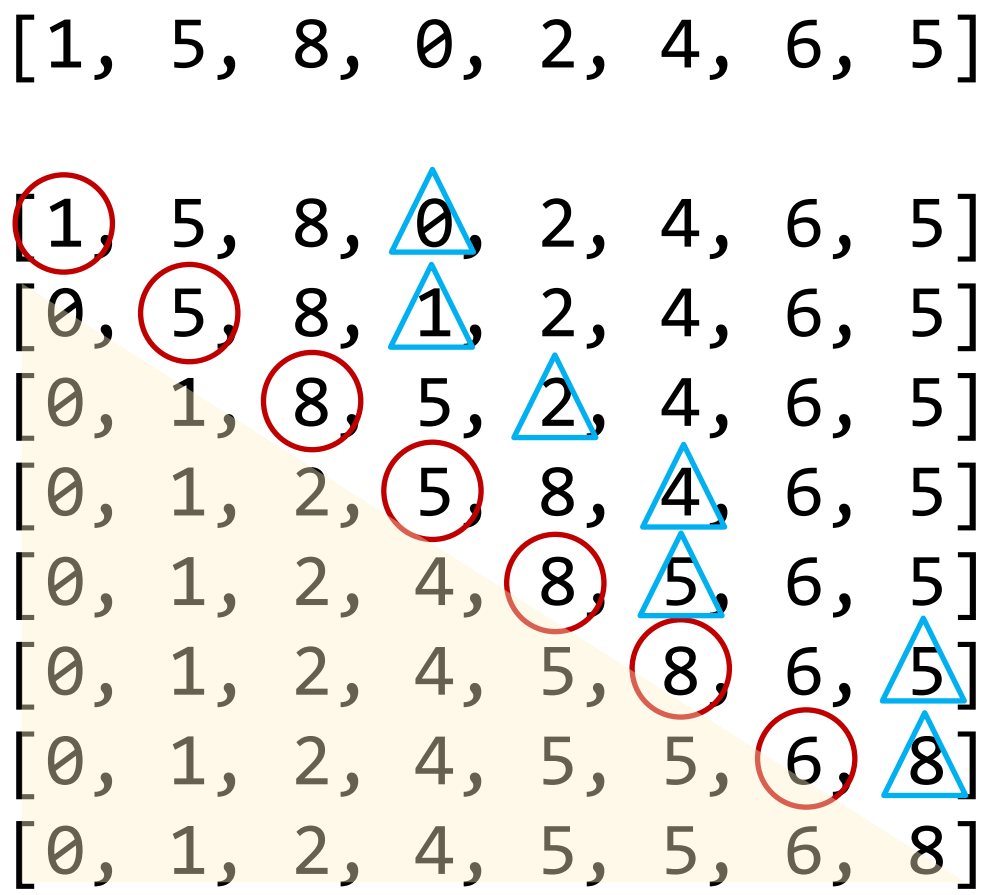
- `()` in function call and `[]` in array subscripting have
  - the highest precedence
  - left to right associativity

Example: Selection Sort

```c
#include <stdio.h>

int main(void)
{
    int a[] = {1, 5, 8, 0, 2, 4, 6, 5};
    int i, j, tmp, min_idx, n;

    n = sizeof(a)/sizeof(int);
    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i + 1; j < n; j++)
            if (a[min_idx] > a[j])
                min_idx = j;
        tmp = a[min_idx];   /* swap a[min_idx], a[i] */
        a[min_idx] = a[i];
        a[i] = tmp;
    }
    printf("After sorting: ");
    for (i = 0; i < n; i++)
        printf("%3d ", a[i]);
    printf("\n");
}
```

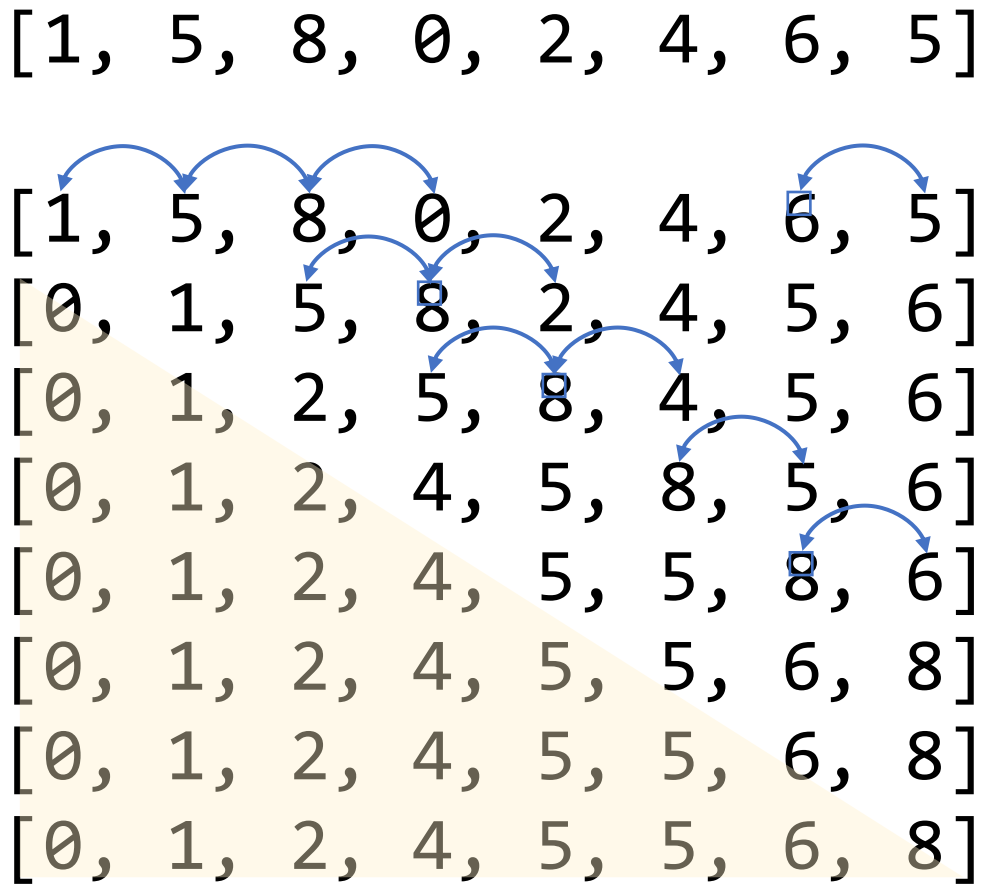


Example: Bubble Sort

```c
#include <stdio.h>

int main(void)
{
    int a[] = {1, 5, 8, 0, 2, 4, 6, 5};
    int i, j, tmp, n;

    n = sizeof(a)/sizeof(int);
    for (i = 0; i < n - 1; i++)
        for (j = n - 1; j > i; j--)
            if (a[j-1] > a[j])
            {
                tmp = a[j-1];   /* swap a[j-1], a[j] */
                a[j-1] = a[j];
                a[j] = tmp;
            }
    printf("After sorting: ");
    for (i = 0; i < n; i++)
        printf("%3d ", a[i]);
    printf("\n");
}
```



Pointers (1)

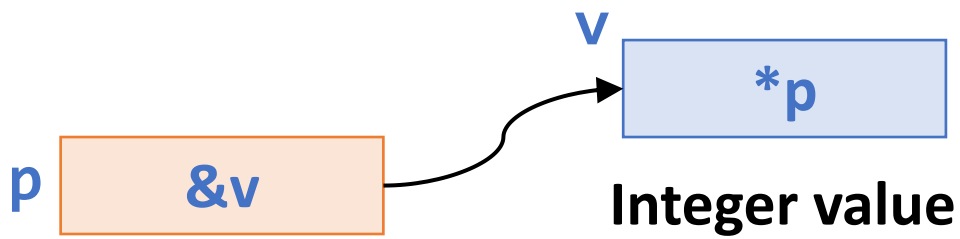
- & operator
  - Unary address operator, right-to-left associativity
  - If v is a variable, then &v is the address (location) in memory space

- Pointer variable
  - A variable which takes addresses as values
  - Can be declared in program
  - When we want to declare p as a pointer variable, we should declare *p like a simple variable
    (e.g.) `int *p;`

Pointers (2)

- * operator
  - Unary "indirection" or "dereferencing" operator, right-to-left associativity
  - If p is a pointer, then then *p is the value of the variable of which p is the address
  - The direct value of p is a memory location
  - *p is the indirect value of p, namely, the value of the memory space of which address is stored in p

```
int  v;
int *p = &v;
```



Pointers (3)

- A legal value of pointer variable
  - A special address 0 (or `NULL` in `<stdio.h>`)
  - Positive integers being interpreted as machine addresses

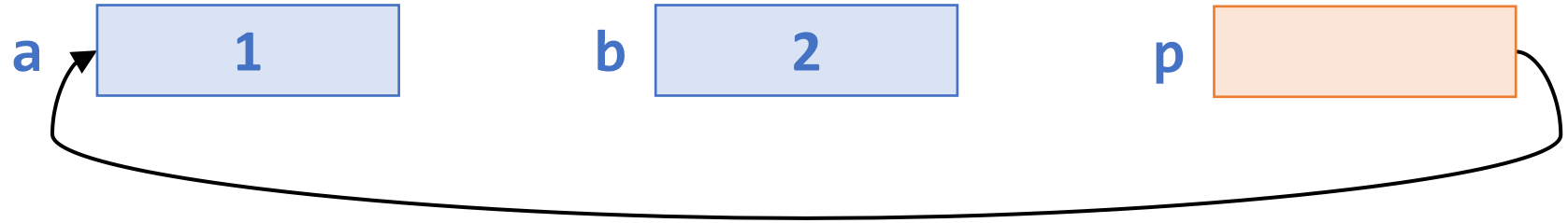
```
p = 0;
p = NULL;                /* equivalent to p = 0; */
p = &i;
p = (int *) 1776;        /* an absolute address in memory */
```

Pointers (4)

```
int a = 1, b = 2, *p;
```

Since a value of p has not been assigned, we do not know yet what it points to

`p = &a;` "p is assigned the address of a"



`b = *p;` "b is assigned the value of storage pointed to by p"

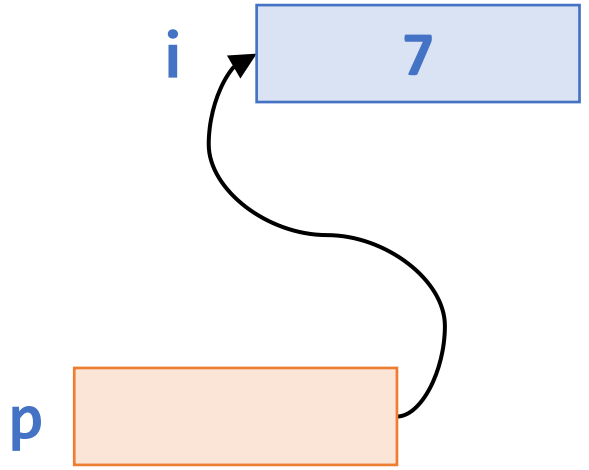
`b = *p;` ⇔ `b = a;`

Pointers (5)

```c
#include <stdio.h>

int main(void)
{
    int i = 7;
    int *p = &i;

    printf("Value of i: %d\n", *p);
    printf("Location of i: %p\n", p);
    return 0;
}
```




```
Value of i: 7
Location of i: 0x7fffe995c7ec
```

Pointers (6)

Operator	Associativity
`()` `[]` `++` (*postfix*) `--` (*postfix*)	Left to right
`+` (*unary*) `-` (*unary*) `++` (*prefix*) `--` (*prefix*) `!` `&` (*address*) `*` (*dereference*)	Right to left
`*` `/` `%`	Left to right
`+` `-`	Left to right
`<` `<=` `>` `>=`	Left to right
`==` `!=`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Right to left
`=` `+=` `-=` `*=` `/=` `%=` *etc.*	Right to left
`,` (*comma operator*)	Left to right

Pointers (7)

Declarations and initializations

```
int i = 3, j = 5, *p = &i, *q = &j, *r;
double x;
```

Expression	Equivalent expression	Value
`p == &i`	`p == (&i)`	1
`**&p`	`*(*(&p))`	3
`r = &x`	`r = (&x)`	/* illegal */
`7 * *p / *q + 7`	`((7 * (*p)) / (*q)) + 7`	11
`*(r = &j) *= *p`	`(*(r = (&j))) *= (*p)`	15

Pointers (8)

- Conversions during assignment between different pointer types are allowed
  - When one of the type is a pointer to `void`
  - When the right side is the constant `0`

Declarations and initializations	
`int *p;  float *q;  void *v;`	
Legal assignments	**Illegal assignments**
`p = 0;`	`p = 1;`
`p = (int *) 1;`	`v = 1;`
`p = v = q;`	`p = q;`
`p = (int *) q;`	

Call-by-Reference

- "Call-by-value": parameter passing in C
  - The values of variables in the calling environment are unchanged

- "Call-by-reference" mechanism
  - For changing the values of variables in the calling environment
    1. Declaring a function parameter to be a pointer
    2. Using the dereferenced pointer in the function body
    3. Passing an address as an argument when calling the function

Call-by-Value vs. Call-by-Reference

call-by-value

```c
#include <stdio.h>

void swap(int p, int q)
{
    int tmp = p;
    p = q;
    q = tmp;
}

int main(void)
{
    int i = 3, j = 5;
    swap(i, j);
    printf("%d %d\n", i, j);
    return 0;
}
```

call-by-reference

```c
#include <stdio.h>

void swap(int *p, int *q)
{
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main(void)
{
    int i = 3, j = 5;
    swap(&i, &j);
    printf("%d %d\n", i, j);
    return 0;
}
```

Arrays and Pointers (1)

- An array name by itself is an address
- Arrays and pointers
  - Both can be subscripted
  - A pointer variable can take different addresses as values
  - An array name is a fixed address (constant pointer)

- `a[i]` ⇔ `*(a+i)`
- `*(p+i)` ⇔ `p[i]`
- `p = a;` ⇔ `p = &a[0];`
- `p = a + 1;` ⇔ `p = &a[1];`

```
int a[100], *p;
```

Arrays and Pointers (2)

```
int a[100], *p;
```

- `a + i`
  - A pointer arithmetic
  - Points to the `i`-th element of the array (counting from 0)
  - Has as its value the `i`-th offset from the base address of the array, `a`

- `p + i`
  - Represents the `i`-th offset from the value of `p`
  - The actual address produced by such an offset depends on the type that `p` points to

Arrays and Pointers (3)

```
#define     N   100
int a[N], i, *p, sum = 0;
```

```
for (i = 0; i < N; i++)
    sum += a[i];
```

```
for (i = 0; i < N; i++)
    sum += *(a+i);
```

```
for (p = a; p < &a[N]; p++)
    sum += *p;
```

```
for (p = a, i = 0; i < N; i++)
    sum += p[i];
```

- Note that because `a` is a constant pointer, the following expressions are illegal
  - `a = p`
  - `++a`
  - `a += 2`
  - `&a`

Pointer Arithmetic and Element Size

- Pointer arithmetic
  - If the variable `p` is a pointer to a particular type,
    `p + 1` `p + i` `++p` `p += i`

```
double a[2], *p, *q;
p = a;                                  /* points to base of array */
q = p + 1;                              /* equivalent to q = &a[1]; */
printf("%ld\n", q - p);                 /* 1 is printed */
printf("%ld\n", (long) q - (long) p);   /* 8 is printed */
```

- `q - p`
  - yields the int value representing the number of array elements between p and q

Arrays as Function Arguments

- In a function definition, a formal parameter that is declared as an array is actually a pointer
  - When an array is passed as an argument to a function, the base address of the array is passed "call-by-value"

```
double sum(double a[], int n) /* <=> double sum(double *a, int n) */ {
    int i;
    double s = 0.0;
    for (i = 0; i < n; i++)
        s += a[i];
    return s;
}
int main(void) {
    double v[100];
    ...
}
```

Various ways that `sum()` might be called

Invocation	What gets computed and returned
`sum(v, 100)`	v[0] + v[1] + … +v[99]
`sum(v, 88)`	v[0] + v[1] + … +v[87]
`sum(&v[7], k-7)`	v[7] + v[8] + … +v[k-1]
`sum(v+7, 2*k)`	v[7] + v[8] + … +v[2*k + 6]

Example: Bubble Sort (Revisited)

```c
#include <stdio.h>

void swap(int *p, int *q)
{
    int tmp;
    tmp = *p, *p = *q, *q = tmp;
}

void bubble(int d[], int n)    /* void bubble(int *d, int n) */
{
    int i, j;

    for (i = 0; i < n-1; i++)
        for (j = n-1; j > i; j--)
            if (d[j-1] > d[j])     /* if (*(d+j-1) > *(d+j)) */
                swap(&d[j-1], &d[j]);    /* swap(d+j-1, d+j) */
}
```

```c
int main(void)
{
    int a[] = {1, 5, 8, 0, 2, 4, 6, 5};
    int i, n = sizeof(a)/sizeof(int);

    bubble(a, n);
    printf("After sorting: ");
    for (i = 0; i < n; i++)
        printf("%3d", a[i]);
    printf("\n");
}
```

Dynamic Memory Allocation (1)

- `#include <stdlib.h>`
- `void *malloc(size_t size);` /* memory allocation */
- `void *calloc(size_t n, size_t size);` /* contiguous allocation */

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

int main(void)
{
    int *a;                         /* to be used as an array */
    int n;                          /* the size of the array */
    ...
    a = calloc(n, sizeof(int));     /* get space for a */
    if (a != NULL)
        ...
}
```

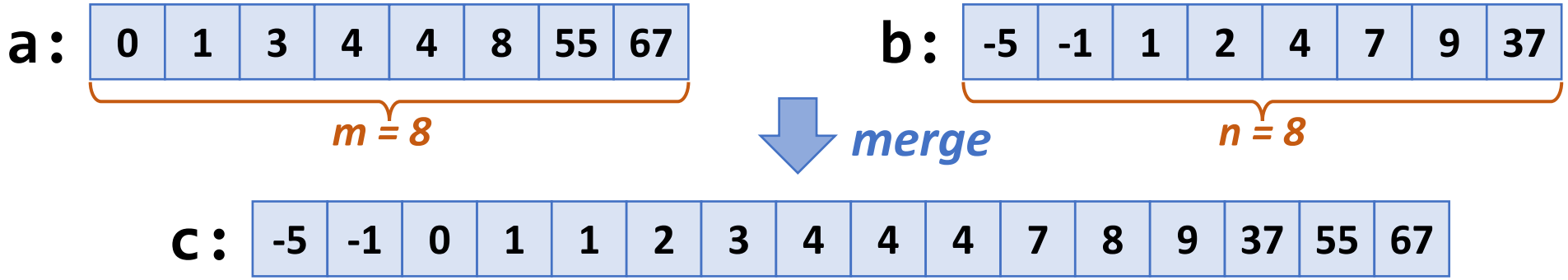
Dynamic Memory Allocation (2)

- `ptr = calloc(n, sizeof(int));`
  - The allocated memory is initialized with all bits set to zero
- `ptr = malloc(n * sizeof(int));`
  - The allocated memory space is not initialized

- Space having been dynamically allocated MUST be returned to the system upon function exit
- `void free(void *`*ptr*`);`
  - *ptr* must be the base address of space previously allocated

Example: Merge Sort (1)

- How to merge two pre-sorted arrays in a sorted order?



merge.c

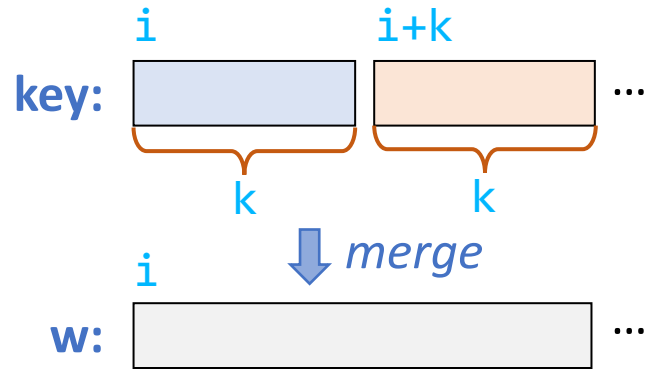
```c
#include "mergesort.h"

void merge(int a[], int b[], int c[], int m, int n) {
    int i = 0, j = 0, k = 0;
    while (i < m && j < n)
        c[k++] = (a[i] < b[j])? a[i++] : b[j++];
    while (i < m)  c[k++] = a[i++];
    while (j < n)  c[k++] = b[j++];
}
```

Example: Merge Sort (2)

k = 1	4	3	1	67	55	8	0	4	-5	37	7	4	2	9	1	-1
k = 2	3	4	1	67	8	55	0	4	-5	37	4	7	2	9	-1	1
k = 4	1	3	4	67	0	4	8	55	-5	4	7	37	-1	1	2	9
k = 8	0	1	3	4	4	8	55	67	-5	-1	1	2	4	7	9	37
	-5	-1	0	1	1	2	3	4	4	4	7	8	9	37	55	67

```
for (i = 0; i < n; i += 2*k)
    merge(key+i, key+i+k, w+i, k, k);
for (i = 0; i < n; i++)  key[i] = w[i];
```



Example: Merge Sort (3)

mergesort.c

```c
#include "mergesort.h"

void mergesort(int key[], int n)
{
    int i, k, m, *w;

    for (m = 1; m < n; m *= 2);
    if (n < m)  exit(1);
    w = calloc(n, sizeof(int));     /* allocate temporary storage */
    assert(w != NULL);
    for (k = 1; k < n; k *= 2)
    {
        for (i = 0; i < n; i += 2*k)
            merge(key+i, key+i+k, w+i, k, k);
        for (i = 0; i < n; i++)  key[i] = w[i];
    }
    free(w);                        /* free temporary storage */
}
```

Example: Merge Sort (4)

main.c

```c
#include "mergesort.h"

int main(void)
{
    int sz;
    int key[] = {4, 3, 1, 67, 55, 8, 0, 4, -5, 37, 7, 4, 2, 9, 1, -1};

    sz = sizeof(key)/sizeof(int);
    printf("Before mergesort:\n");
    printkey(key, sz);
    mergesort(key, sz);
    printf("After mergesort:\n");
    printkey(key, sz);

    return 0;
}
```

Example: Merge Sort (5)

printkey.c

```c
#include "mergesort.h"

void printkey(int key[], int n)
{
    int i;

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

```
$ gcc -o mergesort mergesort.c merge.c
printkey.c main.c
$ ./mergesort
```

mergesort.h

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

void merge(int a[], int b[], int c[], int m, int n);
void mergesort(int key[], int n);
void printkey(int key[], int n);
```