

CS2311 Computer Programming

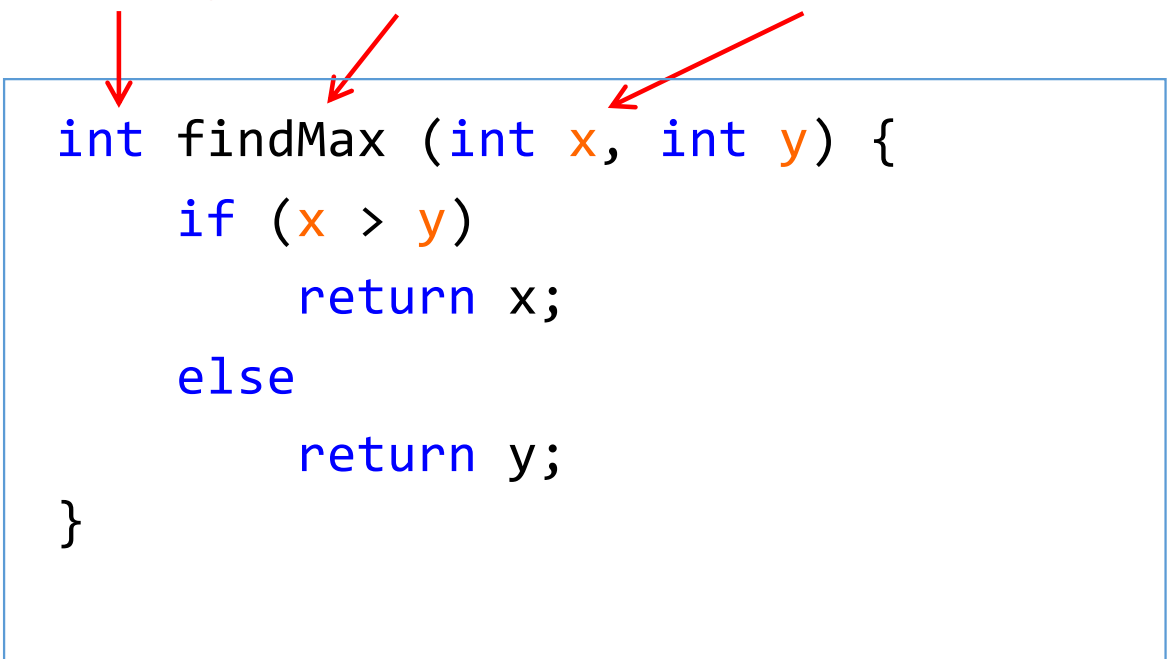
LT06: Array

Computer Science, City University of Hong Kong

Semester B 2022-23

Review: Defining a Function

return_type function_name parameter_list



```
int findMax (int x, int y) {  
    if (x > y)  
        return x;  
    else  
        return y;  
}
```

- **return_type**: A function may return some value.
A return_type is the data type of the value the function returns. Some functions do not return any value. In this case, the return_type is **void**.
- **function_name**: The actual name of the function.
- **parameter_list**: The input arguments. The parameter list refers to the type and order of parameters. A function may contain no parameters.

Review: Calling a Function

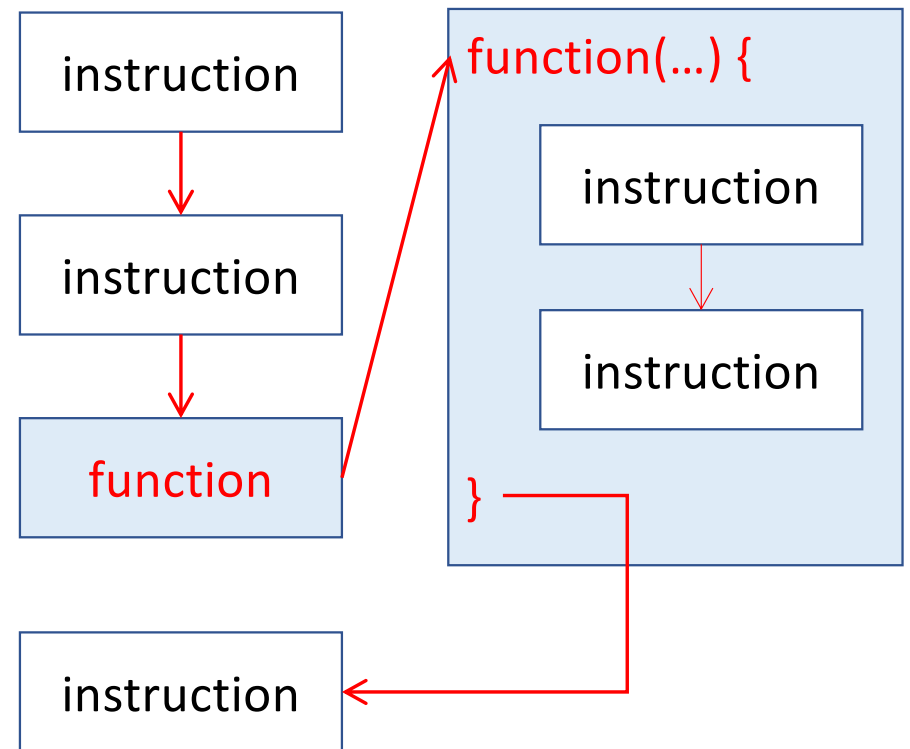
- To make a function call, we only need to specify a function name and provide argument(s) in a pair of ()

```
void main() {  
    int x=4;  
    printHello(x);  
    cout << "bye";  
}
```

```
void printHello(int n) {  
    int i;  
    for (i=0; i<n; i++)  
        cout << "Hello\n";  
}
```

Review: Flow of Function Call

- During program execution, when **a function name followed by parentheses** is encountered, the function is invoked, and the program control is passed to that function;
- when the function ends, program control is returned to the statement immediately after the function call in the original function



Review: Function in Memory Stack

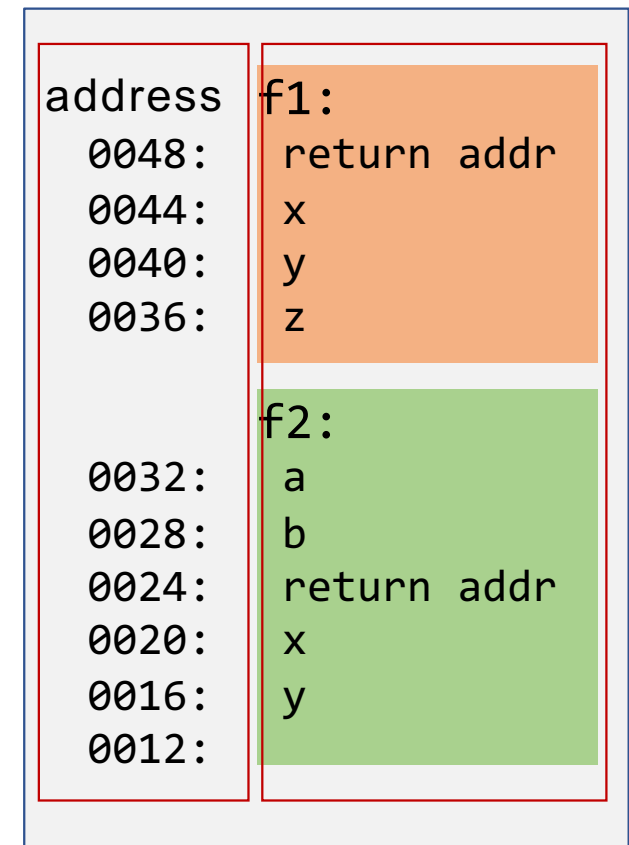
```
void f1() {  
    int x = 5;  
    int y = 6;  
    int z = f2(x, y);  
    cout << x << " " << z;  
}
```

```
int f2(int a, int b) {  
    int x = a + b;  
    int y = b - a;  
    a++;  
    return x*y;  
}
```

Each function has its own memory space which stores

- data (i.e., local variables and parameters)
- return address (address of the next instruction in the calling function)

Memory Stack



Review: Function Prototype

- A function should be defined **before** use

```
// CORRECT
```

```
int findMax(int x, int y) {  
    return (x > y) ? x:y;  
}  
void main() {  
    cout << findMax(3, 4);  
}
```

```
// SYNTAX ERROR
```

```
void main() {  
    cout << findMax(3, 4); // findMax undefined  
}  
int findMax(int x, int y) {  
    return x>y ? x:y;  
}
```

- Suppose we have 3 functions, where func1 calls func2, func2 calls func3, and func3 calls func1. In what order should the functions be defined?
- C++ allows us to bypass this problem using *function prototypes*

Review: Function Prototype

- C++ allows us to *declare a function* and then call the function before defining it
- The declaration of the function is called *function prototype*, which
 - specifies the function name, parameters and return type
 - for example, the following statement declares foo as a function, there is no input and no return value

```
void foo (void);
```

Review: Function Prototype

```
// SYNTAX ERROR
void main() {
    cout << findMax(3, 4); // findMax undefined
}

int findMax(int x, int y) {
    return x>y ? x:y;
}
```

```
// CORRECT
int findMax(int, int);

void main() {
    cout << findMax(3, 4);
}

int findMax(int x, int y) {
    return x>y ? x:y;
}
```

- `int findMax(int, int)` declares `findMax` as a function name, the return type is `int`, and there're two parameters and their types are `int`
- Another way to declare the prototype is: `int findMax(int n1, int n2);`
- However, the variable names are optional, and you can use different parameter names in the actual function definition

Review: Function Prototype

- In a large program, a function f may be used by many other functions written in different source files
 - where to declare function f ?
- In C++, function prototype and definition can be stored separately
- Header file (.h):
 - With extension **.h**, e..g, `stdio.h`, `string.h`
 - Contain function prototype only
 - To be included in the program that will call the function
- Implementation file (.cpp)
 - Contain function implementation (definition)

Review: Function Prototype Example

main.cpp

```
#include "mylib.h"

void main() {
    int x, y=2, z=3;
    x = calMin(y, z);
}
```

mylib.h

```
int calMin(int, int)
```

mylib.cpp

```
int calMin(int a, int b) {
    if (a>b)
        return b;
    else
        return a;
}
```

Review: Argument vs Parameter

- **Parameter**: (a.k.a, *formal parameter*)
 - identifier that appears in function declaration
- **Argument**: (a.k.a, *actual parameter*)
 - expression that appears in a function call

```
void printHello(int n) {  
    int i;  
    for (i=0; i<n; i++)  
        cout << "Hello\n";  
}
```

```
void main() {  
    int x=4;  
    printHello(x);  
    cout << "bye";  
}
```

Review: Parameter Passing in C++

- There're different ways in which arguments can be passed into the called function
- Three most common methods
 - Pass-by-Value
 - Pass-by-Reference
 - Pass-by-Pointer (later)

Review: Pass-by-Value

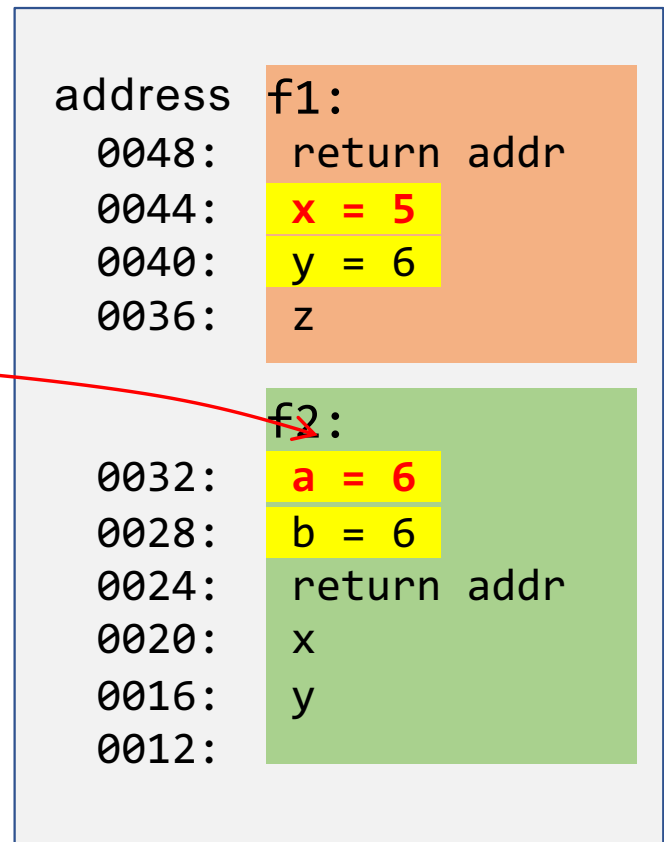
- the value of argument is *copied* to parameter

```
void f1() {  
    int x = 5;  
    int y = 6;  
    int z = f2(x, y);  
    cout << x << " " << z;  
}
```

```
int f2(int a, int b) {  
    int x = a + b;  
    int y = b - a;  
    a++;  
    return x*y;  
}
```

- when `a++` is executed in `f2`, only the memory storage of `f2` is modified

Memory Stack



Review: Pass-by-Value

```
void func(int y) {  
    y=4; // modify y in func(), not the one in main()  
}  
  
void main(){  
    int y=3;  
    func(y);  
    cout << y << endl; // print 3, y remains unchanged  
}
```

- `y` and `y` are two different variables stored in *different* places in memory
 - `y` (parameter) is a local variable in `func`
 - `y` (argument) is a local variable in `main`
- => Modifying `y` in `func` doesn't affect `y`

Review: Pass-by-Reference

- Argument *address* is passed to the parameter
- *Argument can be updated inside the function*
- Add '&' in front of the parameter that to be pass by reference

```
void swap(int &a, int &b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}
```

```
void main() {  
    int x=1, y=3;  
    swap(x, y);  
    cout << "x:" << x << ", y:" << y << endl;  
}
```

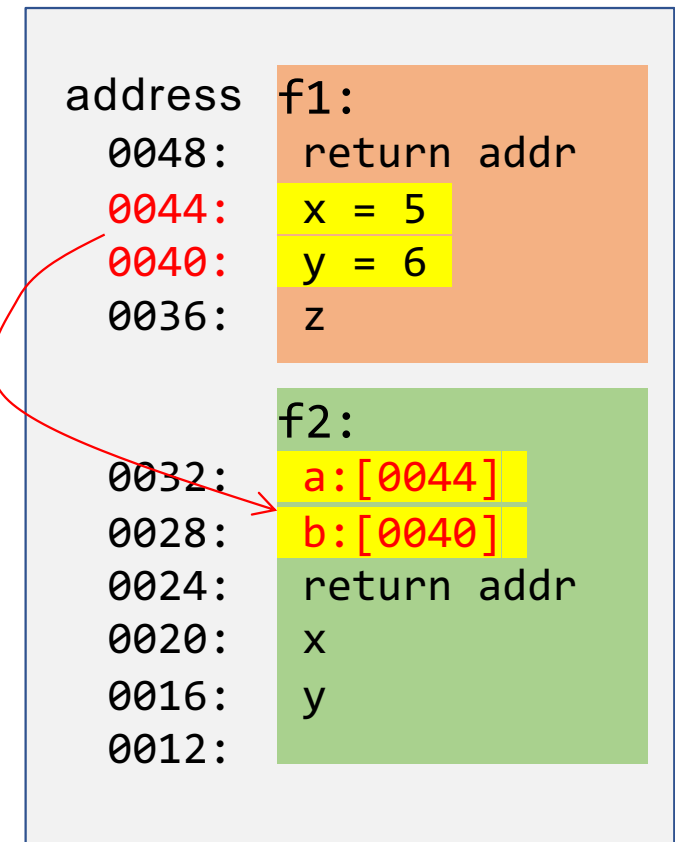
Review: Pass-by-Reference

```
void f1() {  
    int x = 5;  
    int y = 6;  
    int z = f2(x, y);  
    cout << x << " " << z;  
}
```

```
int f2(int &a, int &b) {  
    int x = a + b;  
    int y = b - a;  
    a++;  
    return x*y;  
}
```

- the address of x (0044) is passed to f2
- when a++ is executed in f2, the value stored in 0044 (i.e., x in f1) is modified

Memory Stack



Exercise 1

```
#include <iostream>
using namespace std;
char mystery(int firstParameter, int secondParameter);
int main( )
{
    cout << mystery(10, 9) << "ow" << endl;
    return 0;
}

char mystery(int secondParameter, int firstParameter)
{
    if (firstParameter >= secondParameter)
        return 'W';
    else
        return 'H';
}
```

What is the output produced by this program?

Exercise 2

What are the outputs produced by this program?

```
#include <iostream>
using namespace std;
void friendly( );
void shy(int audienceCount=0);
int main( )
{
    friendly( );
    shy(6);
    shy();
    friendly( );
    return 0;
}

void friendly( ) {
    cout << "Hello" << endl;
}

void shy(int Count) {
    if (Count < 5)
        return;
    for (int i = 0; i < Count; i++);
    cout << "Goodbye" << endl;
}
```

Exercise 3

What are the outputs?

10	20	30
1	2	3
1	20	3

```
#include <iostream>
using namespace std;
void figureMeOut(int& x, int y, int& z);

int main( ) {
    int a, b, c;
    a = 10;
    b = 20;
    c = 30;
    figureMeOut(a, b, c);
    cout << a << " " << b << " " << c << endl;
    return 0;
}

void figureMeOut(int& x, int y, int& z) {
    cout << x << " " << y << " " << z << endl;
    x = 1;
    y = 2;
    z = 3;
    cout << x << " " << y << " " << z << endl;
}
```

Recursions

- One basic problem solving technique is to break the task into subtasks
- If a subtask is a smaller version of the original task, you can solve the original task using a recursive function
- A recursive function is one that **invokes itself**, either directly or indirectly
- Example: Factorial

- The factorial of n is defined as

$$0! = 1$$

$$n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1, \text{ for } n > 0$$

- A recurrence relation: (induction)

$$n! = n \cdot (n-1)!, \quad \text{for } n > 0$$

- e.g., $3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 \cdot 0! = 3 \cdot 2 \cdot 1 \cdot 1$

Factorial: Iterative vs Recursive

Iterative

```
int factorial(int n) {  
    int i, fact=1;  
    for (i=1; i<=n; i++) {  
        fact = i*fact;  
    }  
    return fact;  
}
```

Recursive

```
int factorial(int n) {  
    if (n==0)  
        return 1;  
    return n*factorial(n-1);  
}
```

Example: Vertical Number

- Input: one (non-negative) integer
- Output: integer with one digit per line
- How to break down a number into separated digits?

```
void printDigit(int n) {  
    do {  
        cout << n%10 << endl;  
        n/=10;  
    } while (n>0);  
}
```

Input	Output
7894	4 9 8 7

Example: Vertical Number

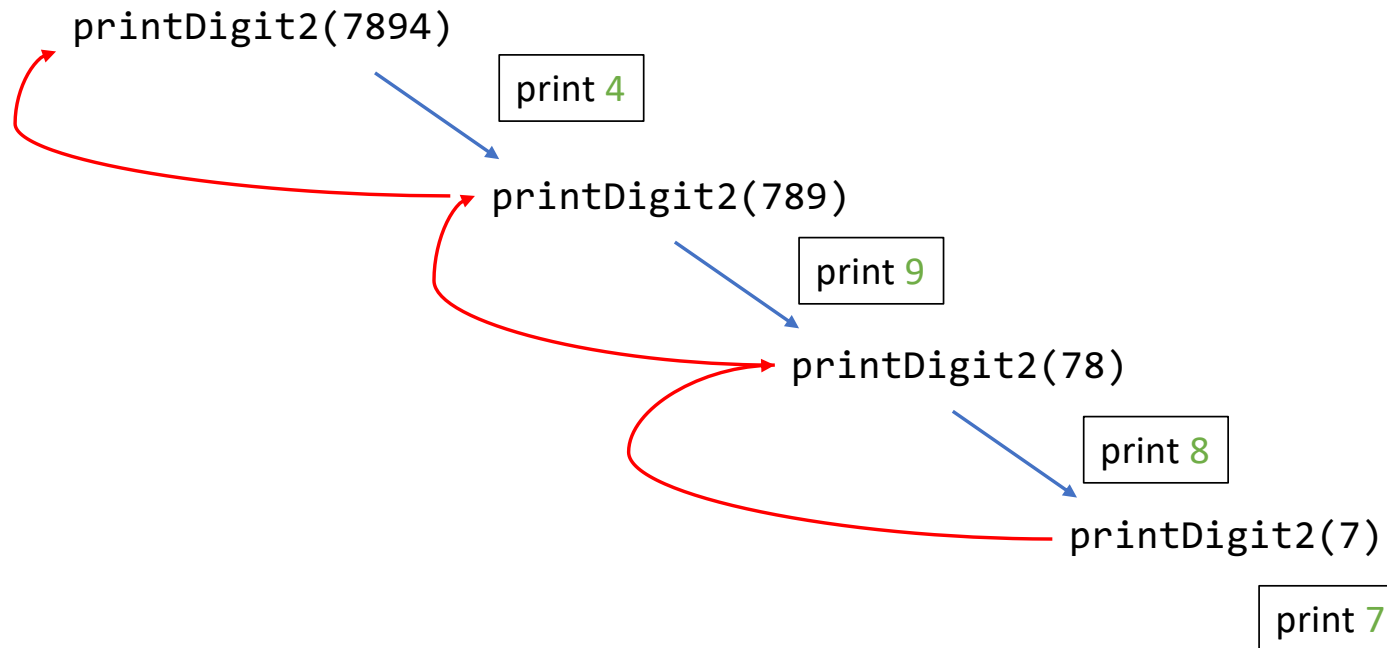
- How to break down a number into separated digits?

```
void printDigit(int n) {  
    do {  
        cout << n%10 << endl;  
        n/=10;  
    } while (n>0);  
}
```

```
void printDigit2(int n) {  
    cout << n%10 << endl;  
    if (n>=10){  
        printDigit2(n/10);  
    }  
}
```

Input	Output
7894	4 9 8 7

Recursive: Leaving



Guidelines for Recursive Functions

- Identify the **parameters**
 - e.g., n in the factorial problem
- Find out a recurrence relation between the current problem and **smaller versions** (in terms of smaller parameters) of the current problem
 - e.g., $\text{factorial}(n) = n * \text{factorial}(n-1)$
- Find out the **base cases** and their solutions
 - e.g., $\text{factorial}(0) = 1$
 - Omitting the base case is one of the common mistakes in writing recursive functions

Checkpoints

1. There is no infinite recursion (check **exit condition**)
2. The break down of the problem works correctly
3. For each of cases that involve recursion, if all recursive calls perform their actions correctly, then the entire case performs correctly.

Checkpoints

	Factorial	Vertical Number
Exit condition	$n == 0$	$n < 10$
Problem break down	$\text{factorial}(n) = n * \text{factorial}(n-1)$ e.g. $n=2 \rightarrow 2! = 2 * 1!$ $n=3 \rightarrow 3! = 3 * 2!$ $n=4 \rightarrow 4! = 4 * 3!$	$\text{cout} << n \% 10;$ $\text{printDigits}(n/10);$ e.g. $n=78 \rightarrow \text{print } 8 \text{ \& call for } 7$ $n=789 \rightarrow \text{print } 9 \text{ \& call for } 78$ $n=7894 \rightarrow \text{print } 4 \text{ \& call for } 789$
If all stopping case are correct	$n!$ is returned	n digits are printed

Efficiency of Recursion

- Generally speaking, non-recursive versions will execute more efficiently (**time/space**)
 - Overhead involved in entering and exiting blocks is avoided in non-recursive solutions.
 - Also have a number of local variables and temporaries that do not have to be saved and restored via a stack.
- There are conflicts between
 - Machine efficiency and
 - Programmer efficiency

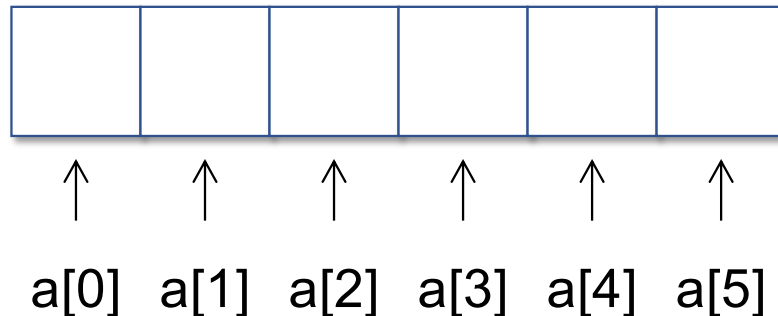
What's an Array?

- Sequence of data items of the same type

`data_type array_name[size]`

- stored continuously
- can be accessed by `index`, or `subscript`

`int a[6];`



What's an Array?

- Sequence of data items of the same type

`data_type array_name[size]`

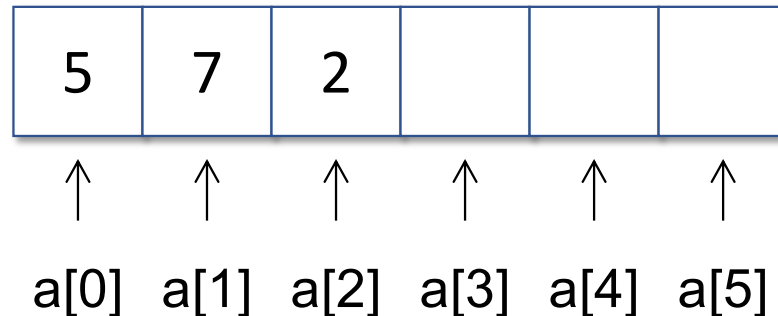
- stored continuously
- can be accessed by `index`, or `subscript`

```
int a[6];
```

```
a[0] = 5;
```

```
a[1] = 7;
```

```
a[2] = 2;
```

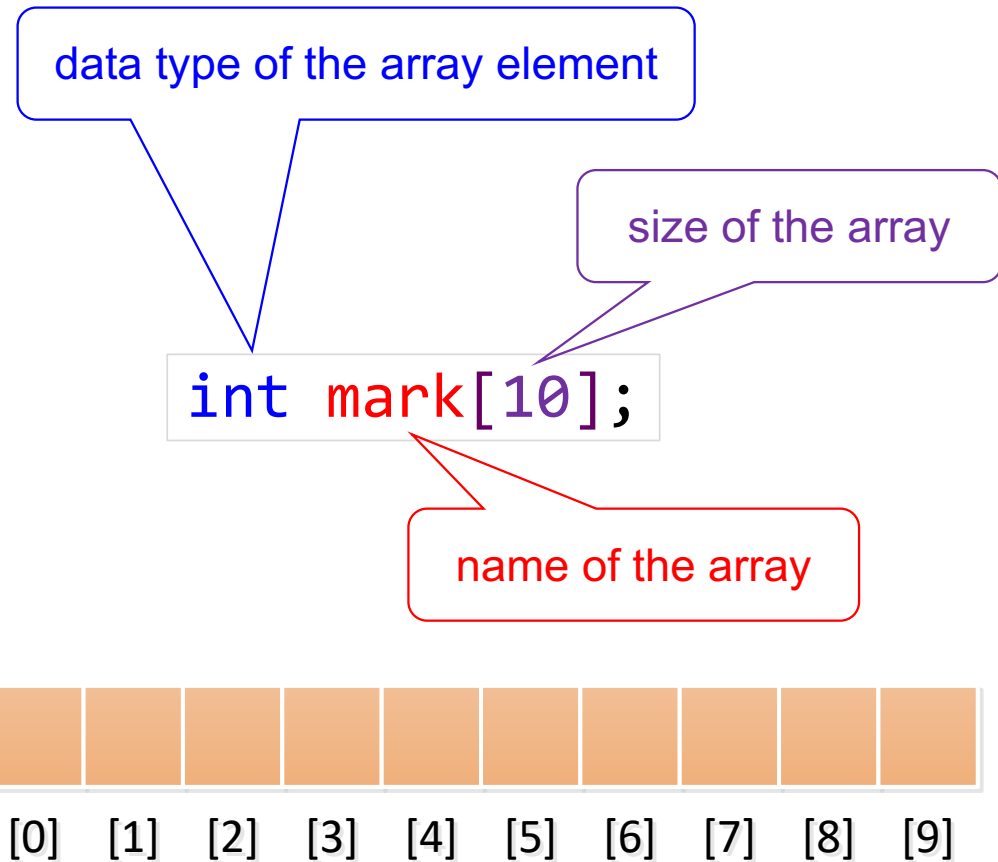


Today's Outline

- Array definition
- Array initialization
- Passing array to functions
- Array operations
- Multi-dimensional array

Array Definition

- There're ten elements in this array
mark[0], mark[1], ..., mark[9]
- the *i*-th array element is mark[*i*-1]
- the range of subscript *i* ranges from 0 to array_size-1
- mark[10] is invalid: array out of bound!



Array Definition

- Set array size

- `const int n = 0;`
`int mark[n];`

- `int mark[50*50];`

- `int n = 0;`
`int mark[n];`

- `int n; cin >> n;`
`int mark[n];`

Using #define to Set Array Size

- `#define` is a C++ predefined **macro** keyword

Usage: `#define A B`

- which globally replaces all occurrences of A to B in the ENTIRE source code listing (all .cpp and all .h files)

- Examples:

```
#define N 100
```

```
#define SIZE 10
```

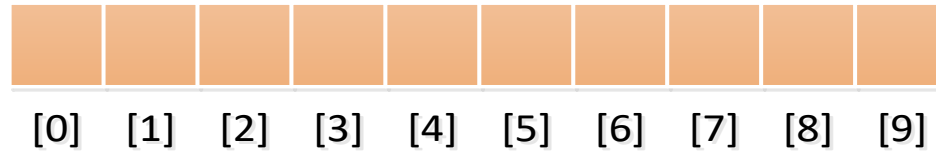
- Using `#define` to set array size

```
#define N 100
```

```
int mark[N];
```

Assign Values to Array Elements

```
int mark[10]
```



- Suppose the mark of the first student is 30: `mark[0] = 30;`
- Input the marks of the i-th student `cin >> mark[i];`
- Input the marks for all 10 students

```
for (int i=0; i< 10; i++)  
    cin >> mark[i];  
  
for (int i=1; i<=10; i++)  
    cin >> mark[i-1];
```

Assign Values to Array Elements

```
int mark[10]
```



[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

- Suppose the mark of the first student is 30: `mark[0] = 30;`
- Input the marks of the i-th student
- Input the marks for all 10 students

```
cin >> mark[i-1];
```

```
for (int i=0; i< 10; i++)  
    cin >> mark[i];
```

```
for (int i=1; i<=10; i++)  
    cin >> mark[i-1];
```

Retrieve Values of Array Elements

- Print the mark of the i-th student

```
cout << mark[i-1];
```

- Print the sum of the marks of all students

```
for (int i = 0; i < 10; i++) {  
    sum += mark[i];  
}  
cout << sum;
```

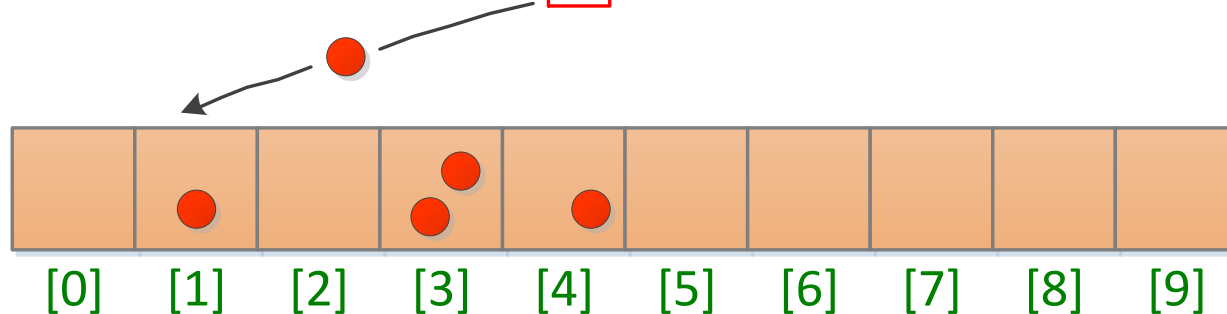
Example 1: Read and Write Array

```
#define N 10
int main() {
    int marks[N], sum=0;
    for (int i=0; i<N; i++)
        cin >> mark[i];
    cout << "The mark of the students are: ";
    for (int i=0; i<N; i++) {
        cout << mark[i];
        sum = sum + mark[i];
    }
    float average = (float)sum/N;
    cout << "Average mark=" << average << endl;
}
```

Example 2: Counting Digits

- Input a sequence of digits (each digit is in $[0, 9]$), which is terminated by -1
- Count the **occurrence** of each digit
- Use an integer array count of 10 elements
 - `count[i]` stores the number of occurrence of digit `i`

Input sequence: 3 4 1 3 1 3 -1



Example 2: Buggy Version

```
int count[10]; // number of occurrence of digits
int digit; // input digit
```

```
do { // read the digits
    cin >> digit;
    if (digit>=0 && digit<=9) // necessary to avoid out-of-bound
        count[digit]++;
} while (digit!=-1); //stop if the input number is -1
```

```
for (int i=0; i<10; i++) { // print the occurrences
    cout << "Frequency of " << i << " is " << count[i] << endl;
}
```


Example 2: The Actual Output

3 4 1 3 1 3 -1

Frequency of 0 is 2089878893

Frequency of 1 is 2088886165

Frequency of 2 is 1376256

Frequency of 3 is 3

Frequency of 4 is 1394145

Frequency of 5 is 1245072

Frequency of 6 is 4203110

Frequency of 7 is 1394144

Frequency of 8 is 0

Frequency of 9 is 1310720

Example 2: Correct Solution

```
int count[10]; // number of occurrence of digits
int digit; // input digit
for (int i=0; i<10; i++)
    count[i] = 0;
do { // read the digits
    cin >> digit;
    if (digit>=0 && digit<=9) // necessary to avoid out-of-bound
        count[digit]++;
} while (digit != -1); //stop if the input number is -1
for (int i=0; i<10; i++) { // print the occurrences
    cout << "Frequency of " << i << " is " << count[i] << endl;
}
```

// It's a good practice to initialize arrays. Otherwise, array element values will be unpredictable

Array Initialization

```
int a[3]={1,2,3};    // a has type int[4] and holds 1, 2, 3
int b[5]={1,2,3};    // b has type int[5] and holds 1, 2, 3, 0, 0
int c[4]={1};        // c has type int[4] and holds 1, 0, 0, 0
int d[3]={0};        // d has type int[3] and holds all zeros
```

Array Initialization

<code>int a[3]={1,2,3};</code>	<code>// a has type int[4] and holds 1, 2, 3</code>
<code>int b[5]={1,2,3};</code>	<code>// b has type int[5] and holds 1, 2, 3, 0, 0</code>
<code>int c[4]={1};</code>	<code>// c has type int[4] and holds 1, 0, 0, 0</code>
<code>int d[3]={0};</code>	<code>// d has type int[3] and holds all zeros</code>
<code>int e[] = {1,2,3};</code>	<code>// e has type int[3] and holds 1, 2, 3</code>
<code>int f[2]={1,2,3};</code>	<code>// it's an error to provide more elements than array size</code>
<code>int g[];</code>	<code>// it's an error to declare an array without specifying size</code>

Count Digits: Correct Solution II

```
int count[10] = {0}; // number of occurrence of digits
int digit; // input digit

do { // read the digits
    cin >> digit;
    if (digit>=0 && digit<=9) // necessary to avoid out-of-bound
        count[digit]++;
} while (digit!=-1); //stop if the input number is -1

for (int i=0; i<10; i++) { // print the occurrences
    cout << "Frequency of " << i << " is " << count[i] << endl;
}
```

Array Initialization Summary

(a) Basic Initialization

```
int numbers[5] = {3, 7, 12, 24, 45};
```



(b) Initialization without Size

```
int numbers[ ] = {3, 7, 12, 24, 45};
```



(c) Partial Initialization

```
int numbers[5] = {3, 7};
```



The rest are filled with 0s

(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```



All filled with 0s

Initialization of Char Arrays

- `char str[3]={ 'a', 'b', 'c' };`
- `char str[3]="abc";` *// str has type char[3] and holds 'a', 'b', 'c'*
- `char str[] = "abc";` *// str has type char[4] and holds 'a', 'b', 'c', '\0'*
 // More details in the string lecture

Exercise 1

a. `int x[4] = { 8, 7, 6, 4, 3 };`

b. `int x[] = { 8, 7, 6, 4 };`

c. `const int SIZE = 4;`
`int x[SIZE];`

Which array definition is generally correct?

Exercise 2

```
double a[3] = {1.1, 2.2, 3.3};  
  
cout << a[0] << " " << ++a[1] << " " << a[2] << endl;  
  
a[1] = a[2]++;  
  
cout << a[0] << " " << a[1] << " " << a[2] << endl;
```

What are the outputs?

Exercise 3

```
int i, temp[10];

for (i = 0; i < 10; i++)
    temp[i] = 2*i;

for (i = 0; i < 10; i=i+2) {
    cout << temp[i] << " ";
    addOne(i);
}
```

```
void addOne(int &a) {
    a++;
}
```

What are the outputs?

Passing Arrays to Function

- To pass an array to a function, we only need to specify the array name
- Array is **passed by pointer**

the size of the array is optional, e.g.,
you can write `void f(int a[])`

```
void func(int a[3]){  
    cout << a[0] << endl; // print 1  
    a[0]=10;  
}  
  
void main () {  
    int a[3]={1,2,5};  
    >func(a);  
    cout << a[0] << endl; // print 10  
}
```

only need to input
the array name!

if the content of `a[i]` is modified in func, the modification will persist even after the function returns (**Call by pointer, more in later lectures**)

Passing Arrays to Function

- To pass an array to a function, we only need to specify the array name
- Array is **passed by pointer**

the size of the array is optional, e.g.,
you can write `void f(int a[])`

```
void func(int a[3]){  
    cout << a[0] << endl; // print 1  
    a[0]=10;  
}  
  
void main () {  
    int a[3]={1,2,5};  
    >func(a);  
    cout << a[0] << endl; // print 10  
}
```

only need to input
the array name!

if the content of `a[i]` is modified in func, the modification will persist even after the function returns (**Call by pointer, more in later lectures**)

Passing Arrays to Function

- The following program is **invalid**

```
void f(int x[20]) {  
    ...  
}  
  
void main() {  
    int y[20];  
    f(y[0]); // invalid, type mismatch  
}
```

Today's Outline

- Array definition
- Array initialization
- Passing array to functions
- Array operations
 - sizeof
 - Compare two arrays
 - Sort
 - Search
- Multi-dimensional array

sizeof

- Recall: sizeof(**data_byte**) gives the number of bytes of the **data_type**

```
cout << sizeof(int); // will print 4
```

- sizeof Array gives the total number of bytes occupied by that array

```
int a[4];  
cout << sizeof(a); // will print 16
```

- How to calculate number of elements of an array?

Compare Two Arrays

- We have two integer arrays, each with 5 elements

```
int array1[5] = {10, 5, 3, 5, 1};
```

```
int array2[5]; // will be entered by the user
```

- Compare whether array1 and array2 are equal
 - **array equality**: two arrays are equal if all of their elements are equal.
 - you have to compare all array elements **one by one**
 - the following code will generate **wrong** result

```
if (array1 == array2)  
    cout << "the arrays are equal\n";
```


Compare Two Arrays

```
void main() {  
    int array1[5] = {10, 5, 3, 5, 1};  
    int array2[5];  
    cout << "input 5 elements to array2\n";  
    for (int i=0; i<5; i++)  
        cin >> array2[i];  
    bool arrayEqual = true;  
    for (int i=0; i<5 && arrayEqual; i++) {  
        if (array1[i] != array2[i])  
            arrayEqual = false;  
    }  
    if (arrayEqual)  
        cout << "The arrays are equal\n";  
    else  
        cout << "The arrays are not equal\n";  
}
```

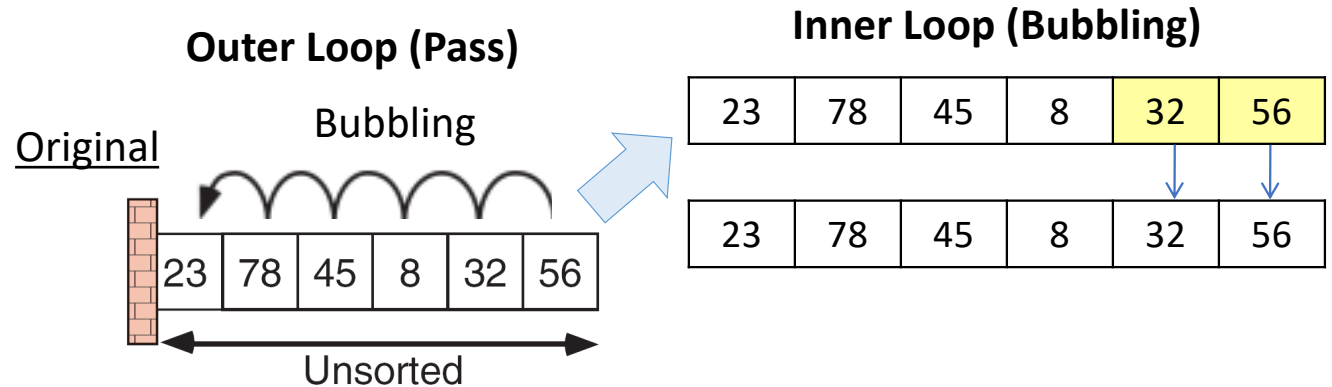
Sorting

- One of the most common application in CS
 - arranging data by their values: {1, 5, 3, 2} -> {1, 2, 3, 5}
- Many algorithms for sorting

Selection sort	}	Slow but easy to code	Quick sort	}	Faster but more complex to code
Bubble sort			Merge sort		
Insertion sort			Heap sort		
- Based on iteratively swapping two elements in the array so that eventually the array is ordered
 - sorting algorithms differ in how they choose the two elements

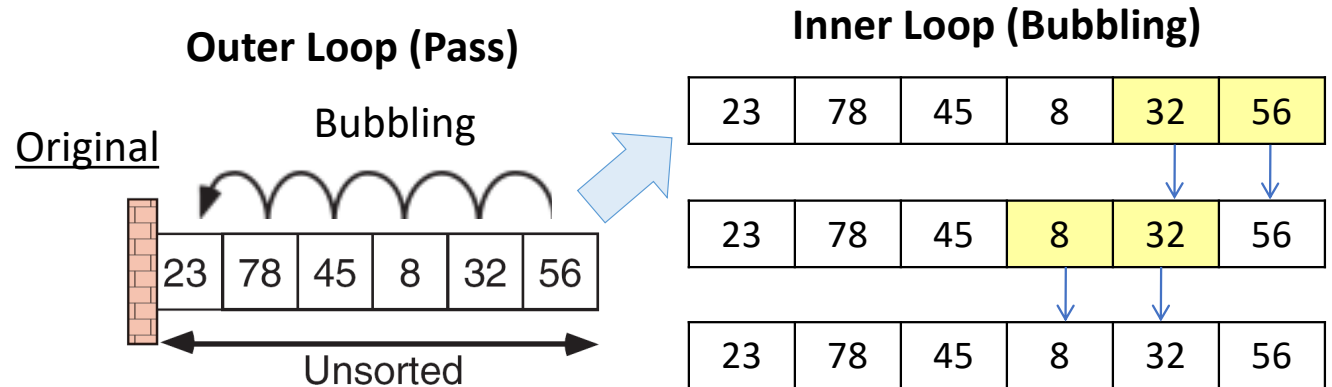
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



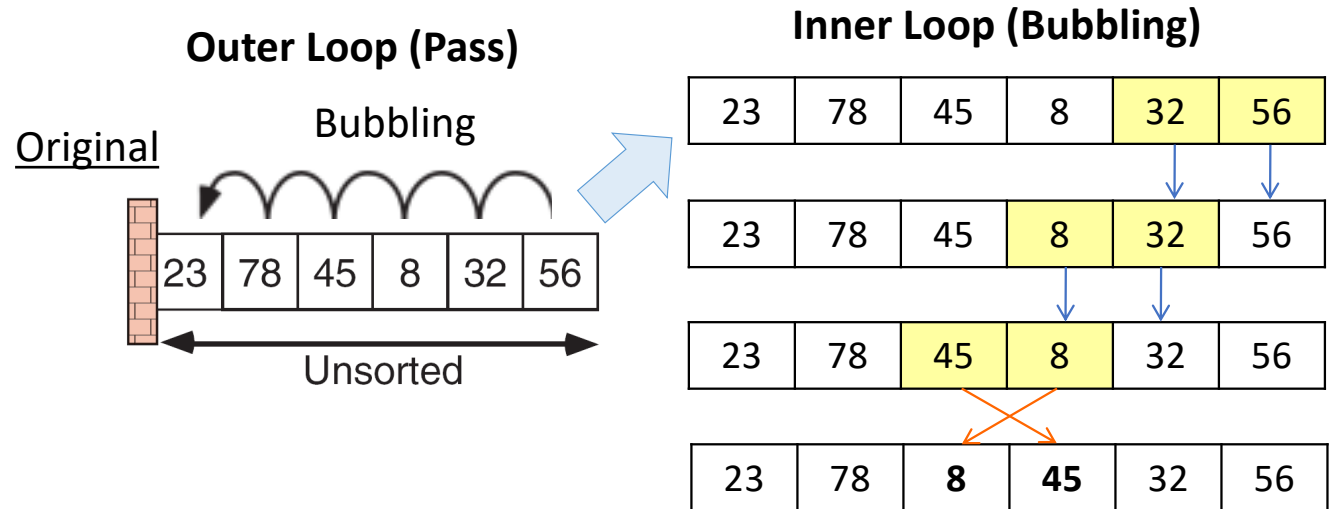
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



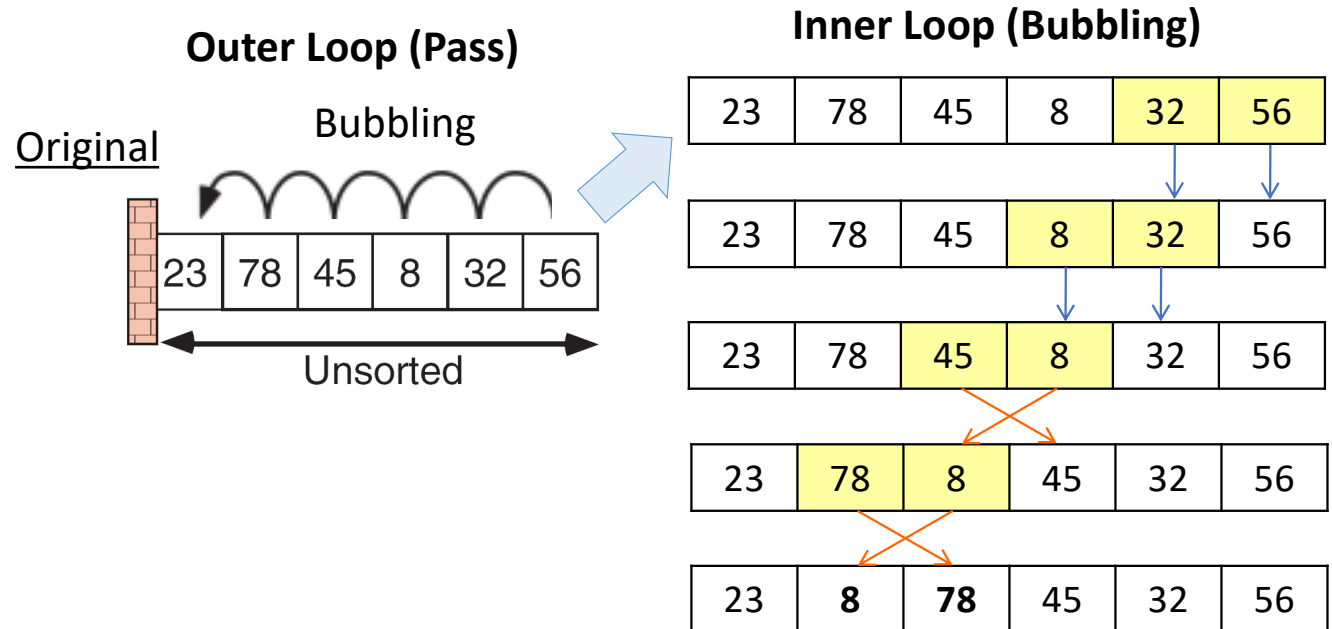
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



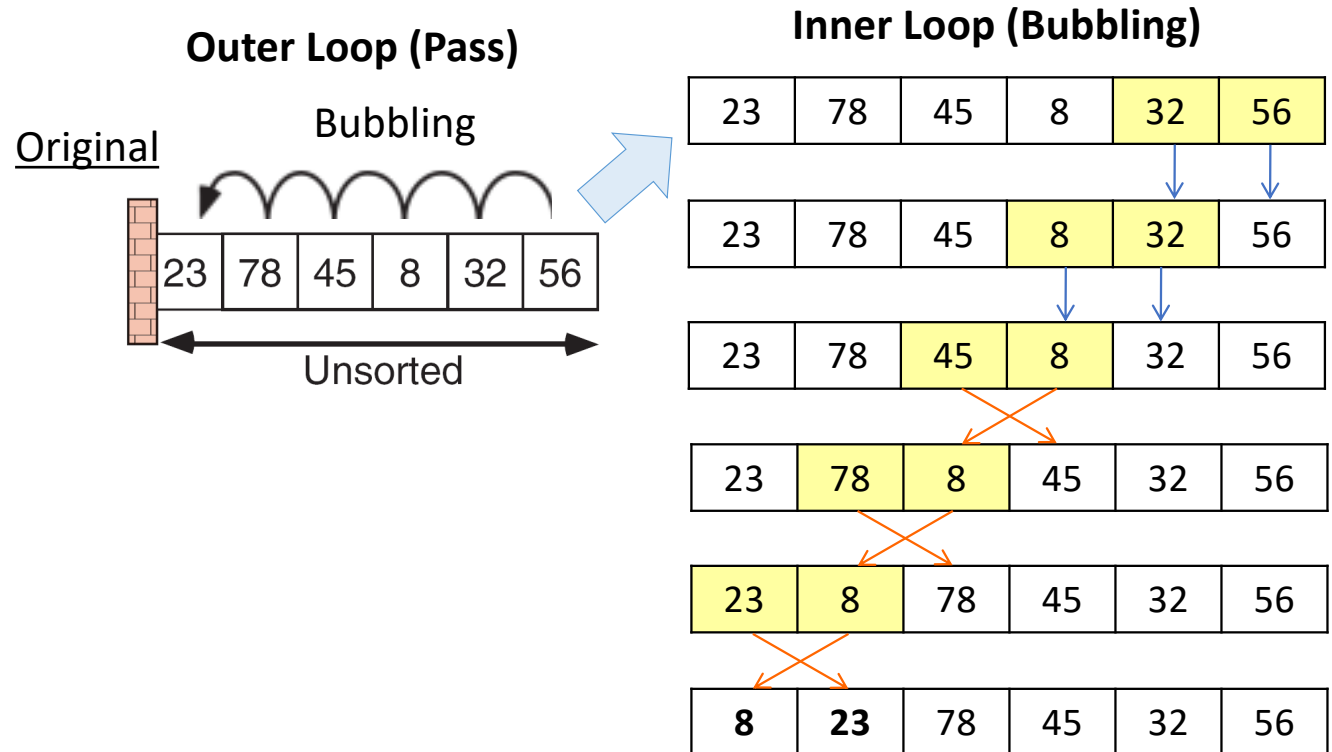
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



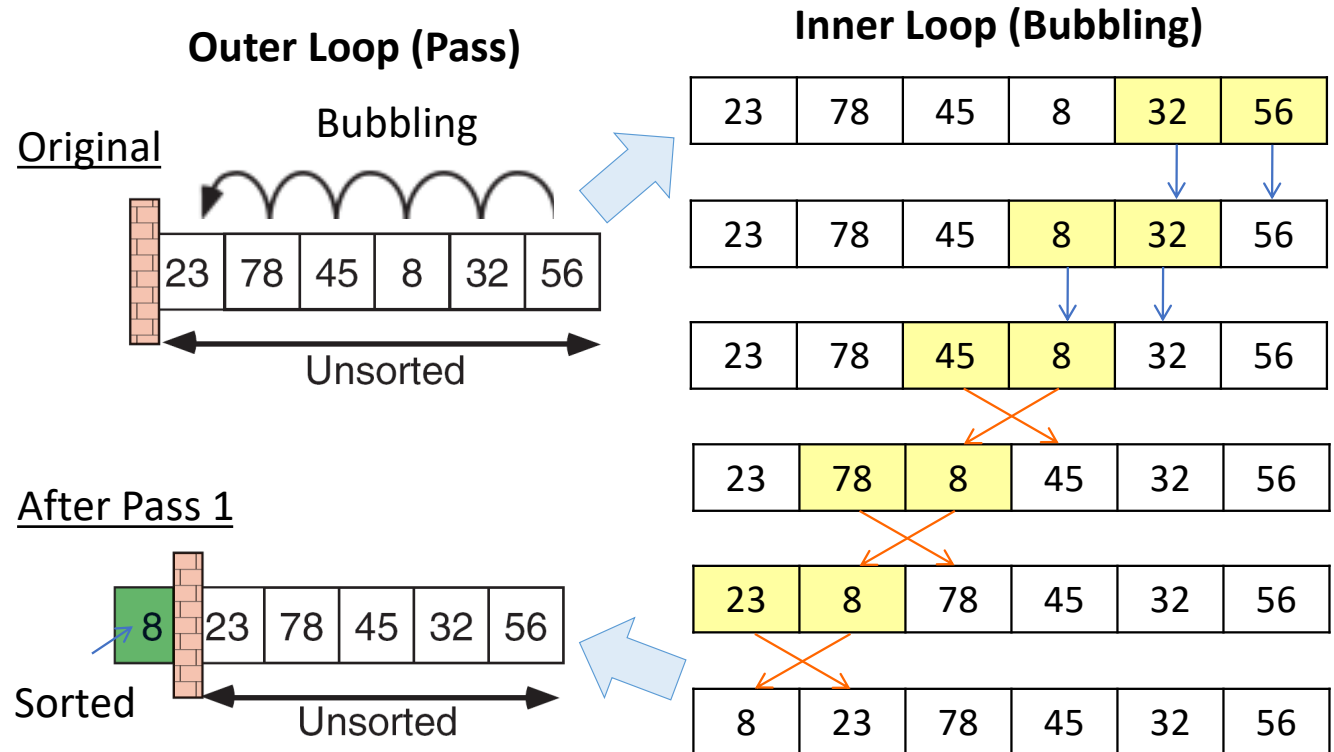
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence
- After the pass, the smallest element is moved ("bubbled up") to the front of the array



Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence
- After the pass, the smallest element is moved ("bubbled up") to the front of the array

```
const int n = 6;
int a[n] = {23, 78, 45, 8, 32, 56};
int k, tmp;

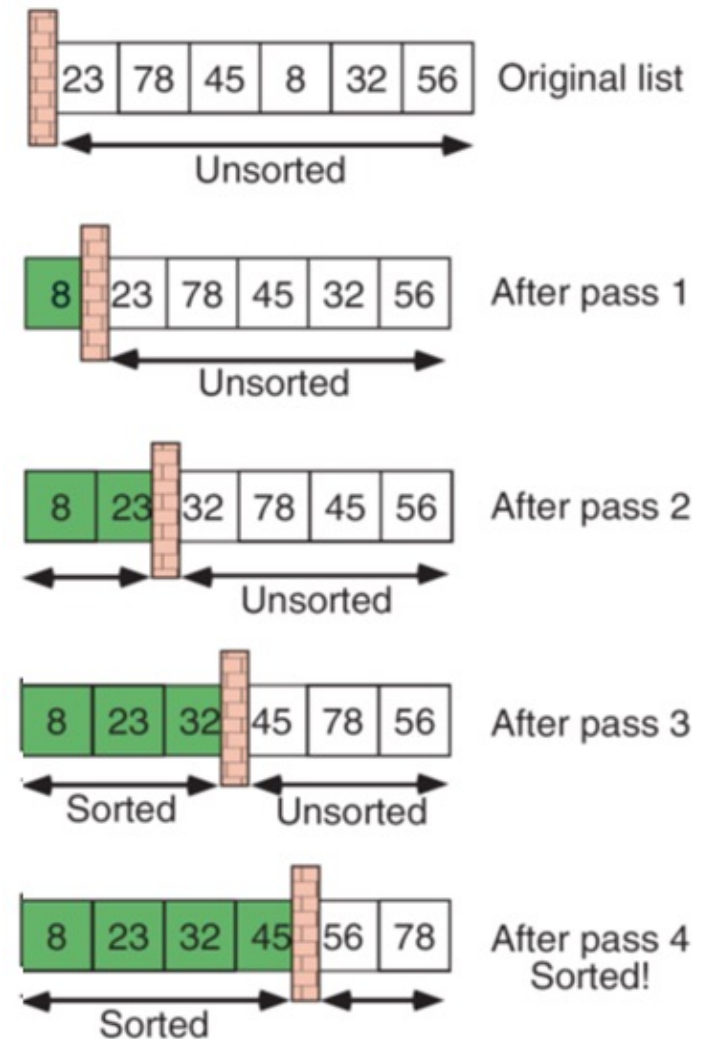
for (k=n-1; k>0; k--) { // bubbling
    if (a[k]<a[k-1]) {
        tmp    = a[k];    // swap
        a[k]    = a[k-1];
        a[k-1] = tmp;
    }
}
```

Bubble Sort

- Repeat bubbling up for n rounds, where n is array size
- After round j (start from round 0), the array is divided into two parts:
 - sorted (green): from $a[0]$ to $a[j]$
 - unsorted: from $a[j+1]$ to $a[n-1]$

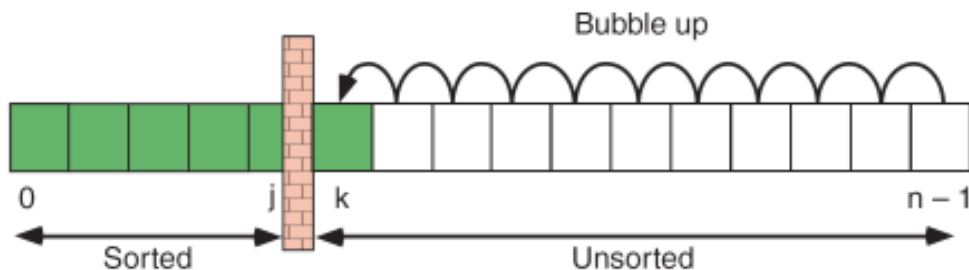
bubble-sort dance: <http://youtu.be/lyZQPjUT5B4>

insert-sort dance: <http://youtu.be/ROalU379l3U>



Bubble Sort

- Repeat bubbling up for n rounds, where n is array size
- After round j (start from round 0), the array is divided into two parts:
 - sorted (green): from $a[0]$ to $a[j]$
 - unsorted: from $a[j+1]$ to $a[n-1]$



```
const int n = 6;
int a[n] = {23, 78, 45, 8, 32, 56};
int j, k, tmp;

for (j=0; j<n-1; j++) {    // outer loop
    for (k=n-1; k>j; k--) { // bubbling
        if (a[k]<a[k-1]) {
            tmp    = a[k];    // swap
            a[k]    = a[k-1];
            a[k-1] = tmp;
        }
    }
}

cout << "sorted: ";
for (j=0; j<n; j++)
    cout << a[j] << ' ';
cout << "\n";
```

Bubble Sort

- Repeat bubbling up for n rounds, where n is array size
- After round j (start from round 0), the array is divided into two parts:
- sorted (green): from $a[0]$ to $a[j]$
- unsorted: from $a[j+1]$ to $a[n-1]$
- **Early stop**: stop when the array is already sorted, no need to go through all $n-1$ passes

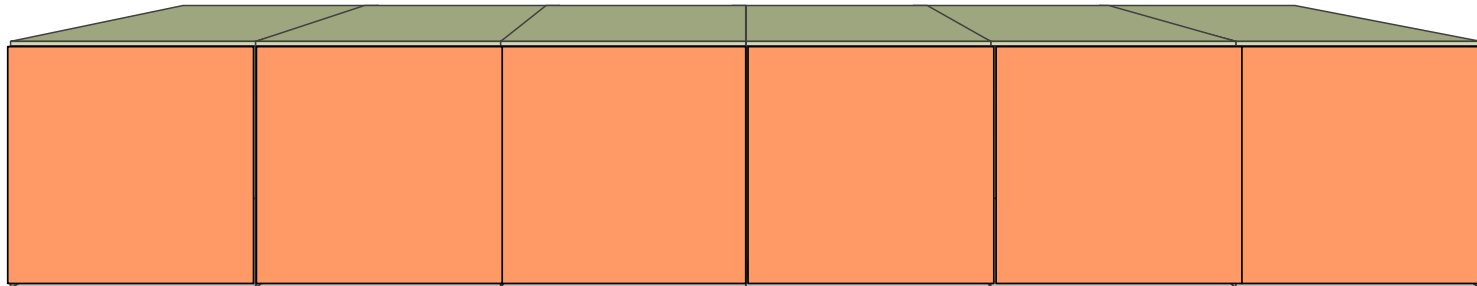
```
const int n = 6;
int a[n] = {23, 78, 45, 8, 32, 56};
int j, k, tmp;
bool sorted = false;
for (j=0; j<n-1 && !sorted; j++) {
    sorted = true;
    for (k=n-1; k>j; k--) { // bubbling
        if (a[k]<a[k-1]) {
            tmp = a[k]; // swap
            a[k] = a[k-1];
            a[k-1] = tmp;
            sorted = false;
        }
    }
}
cout << "sorted: ";
for (j=0; j<n; j++)
    cout << a[j] << ' ';
cout << "\n";
```

Searching

- Search: check if an element is in an array
- Example:
 - read 10 numbers from the user and store them in an array
 - user input another number x
 - write a program to check if x is an array element of the array
 - if yes, output the index of the element
 - if no, output -1

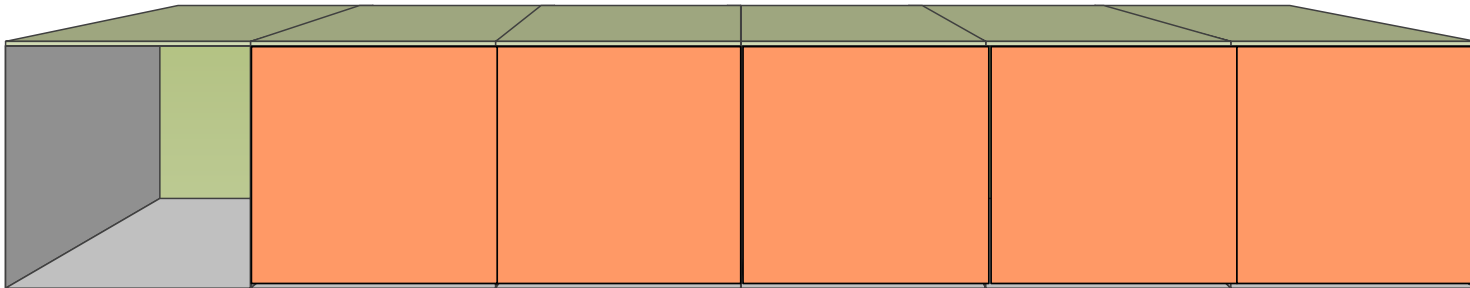
Searching for the Rabbit (case 1)

- Search **sequentially** for rabbit



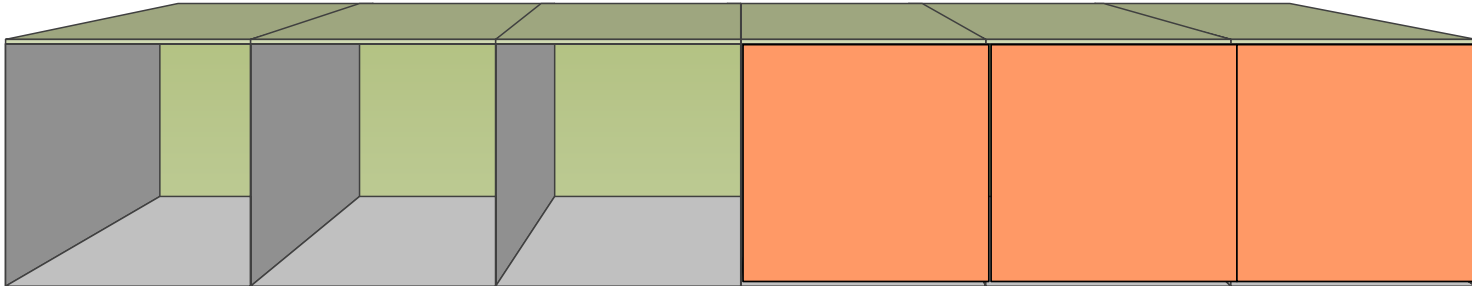
Searching for the Rabbit (case 1)

- Search **sequentially** for rabbit



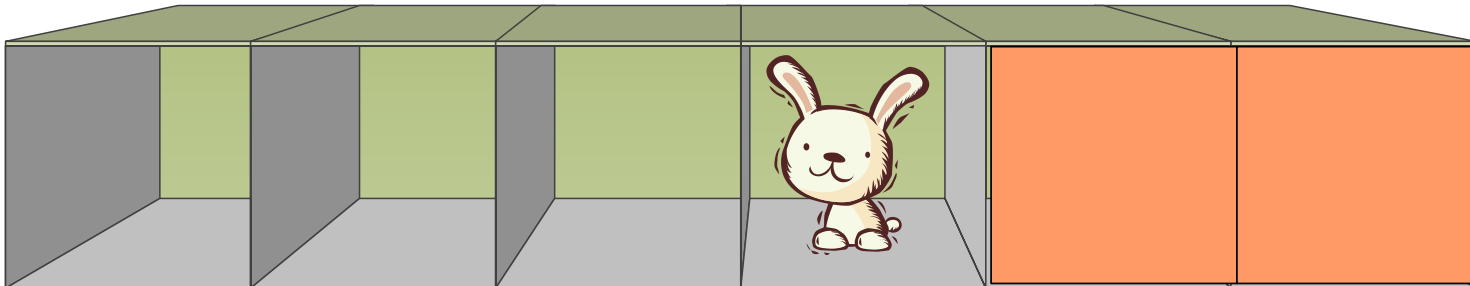
Searching for the Rabbit (case 1)

- Search **sequentially** for rabbit



Searching for the Rabbit (case 1)

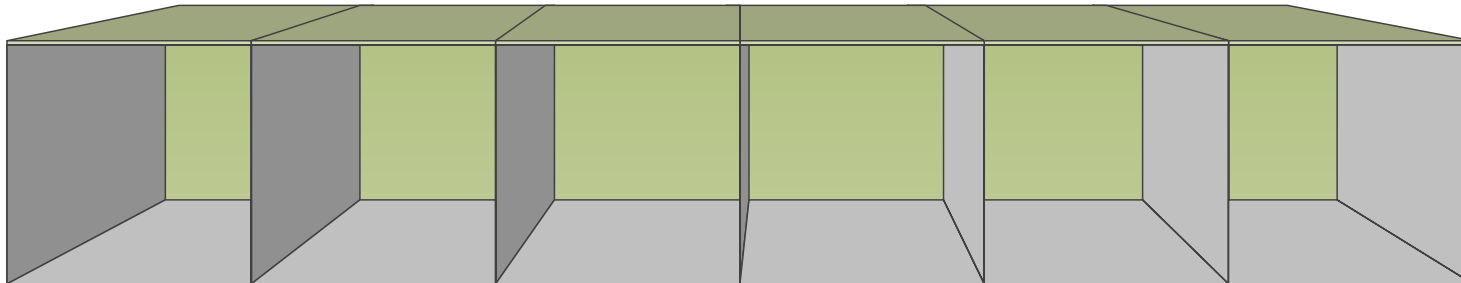
- Search **sequentially** for rabbit



If found, skip the rest

Searching for the Rabbit (case 2)

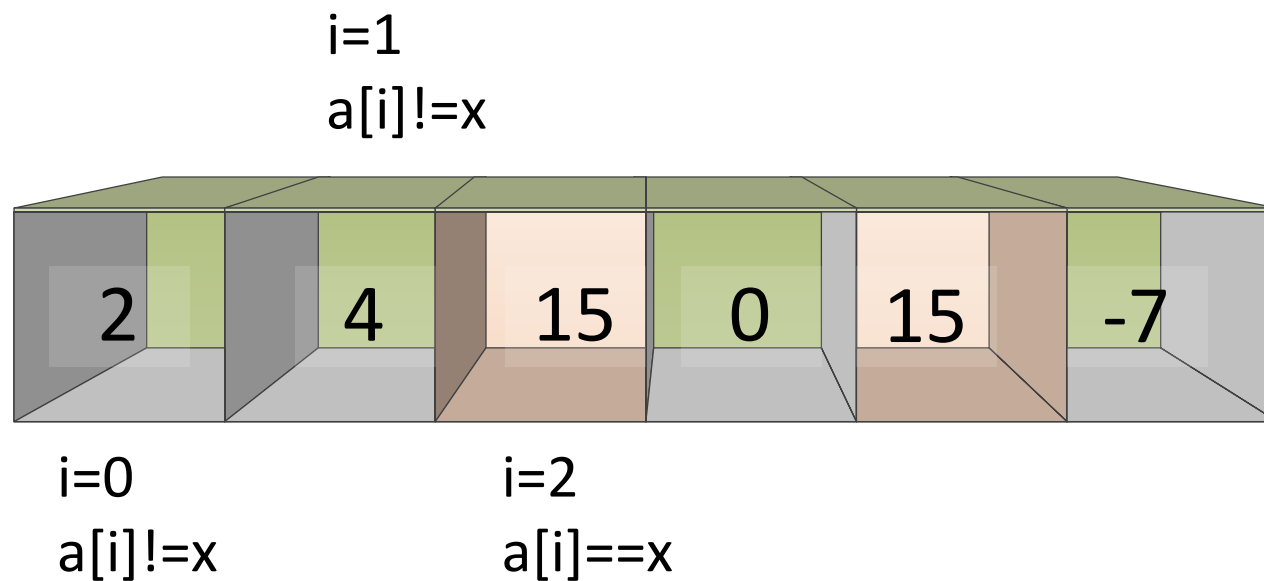
- Search **sequentially** for rabbit



No rabbit found, return -1

Searching for Element (case 1)

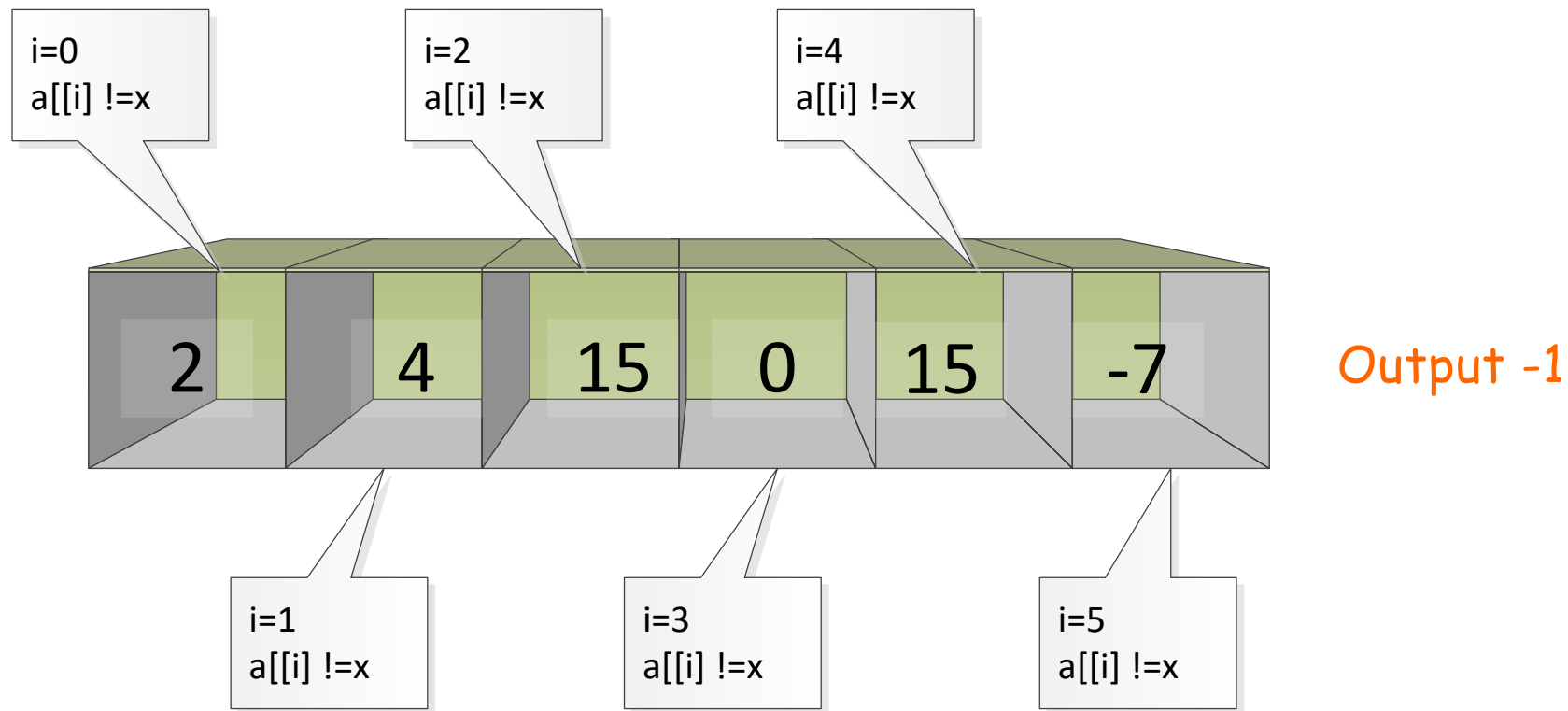
- Search **sequentially** for $x=15$



Output $i=2$,
break out of
the loop

Searching for Element (case 2)

- Search **sequentially** for **x=8**



Searching

- Search **sequentially**

```
#define N 6
int sequentialSearch(int target, int a[N]) {
    for (int i=0; i<N; i++) {
        if (a[i] == x)
            return i;
    }
    return -1;
}
```

Searching in Sorted Arrays

- Sequential search in an array of size N takes $(N+1)/2$ rounds in average
 - $(1 + 2 + \dots + N-1 + N)/N = (N+1)/2$
- Assume the array is already sorted, e.g.,

1	2	4	5	8	9	11	22	23	28	32
---	---	---	---	---	---	----	----	----	----	----

- Suppose the target is 22, can you do faster than sequential search?

Binary Search

- Assume the array is sorted (in ascending order)
- Compare the target with the **middle** element of the array
- If smaller, search in left
- If larger, search in right
- If equal, found

Search Key: 22

Step 1

1	2	4	5	8	9	11	22	23	28	32
---	---	---	---	---	---	----	----	----	----	----

Binary Search

- Assume the array is sorted (in ascending order)
- Compare the target with the **middle** element of the array
- If smaller, search in left
- If larger, search in right
- If equal, found

Search Key: 22

Step 1

1	2	4	5	8	9	11	22	23	28	32
---	---	---	---	---	---	----	----	----	----	----

Step 2

1	2	4	5	8	9	11	22	23	28	32
---	---	---	---	---	---	----	----	----	----	----

Binary Search

- Assume the array is sorted (in ascending order)
- Compare the target with the **middle** element of the array
- If smaller, search in left
- If larger, search in right
- If equal, found

Search Key: 22

Step 1

1	2	4	5	8	9	11	22	23	28	32
---	---	---	---	---	---	----	----	----	----	----

Step 2

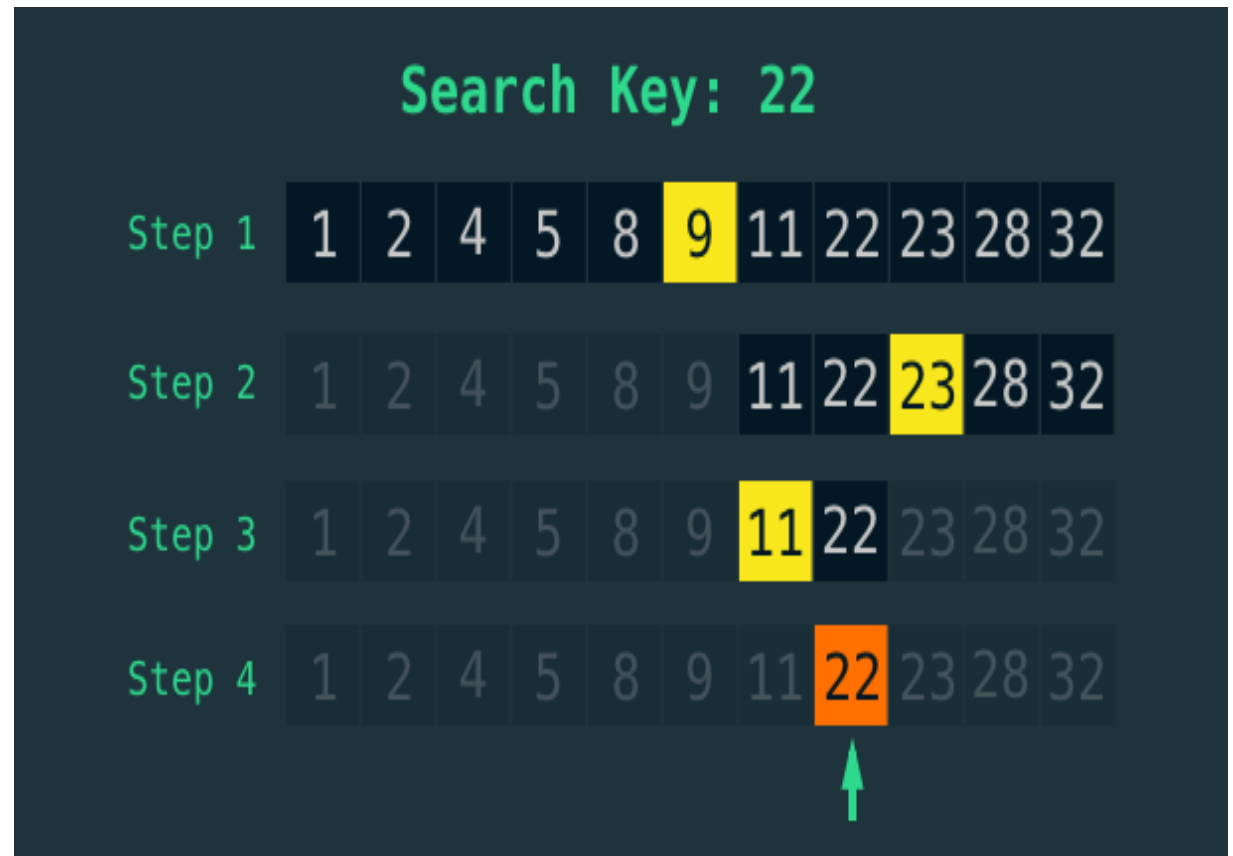
1	2	4	5	8	9	11	22	23	28	32
---	---	---	---	---	---	----	----	----	----	----

Step 3

1	2	4	5	8	9	11	22	23	28	32
---	---	---	---	---	---	----	----	----	----	----

Binary Search

- Assume the array is sorted (in ascending order)
- Compare the target with the **middle** element of the array
- If smaller, search in left
- If larger, search in right
- If equal, found



Binary Search

- In each round, binary search eliminates $N/2$ elements
- In comparison, sequential search eliminates only one



Binary Search

```
int binarySearch(int target, int a[N]) {  
    int first=0, last=N-1, mid;  
    while (target>=a[first] && target<=a[last]) {  
        mid = (first+last)/2;  
        if (target==a[mid])  
            return mid;  
        else if (target>a[mid])  
            first = mid+1;  
        else  
            last = mid;  
    }  
    return -1;  
}
```

Sort + Search

- Assume that you have a huge amount of data (out-of-order) and you need to frequently search in the database for different elements
- What should you do?
- What about if you only need to search once?

Today's Outline

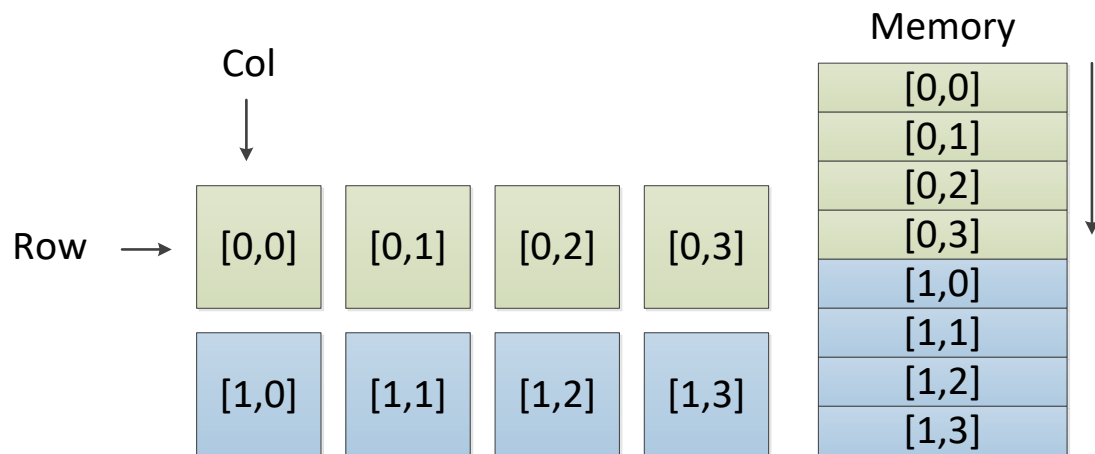
- Array definition
- Array initialization
- Passing array to functions
- Array operations
- Multi-dimensional array

Multi-dimensional Array

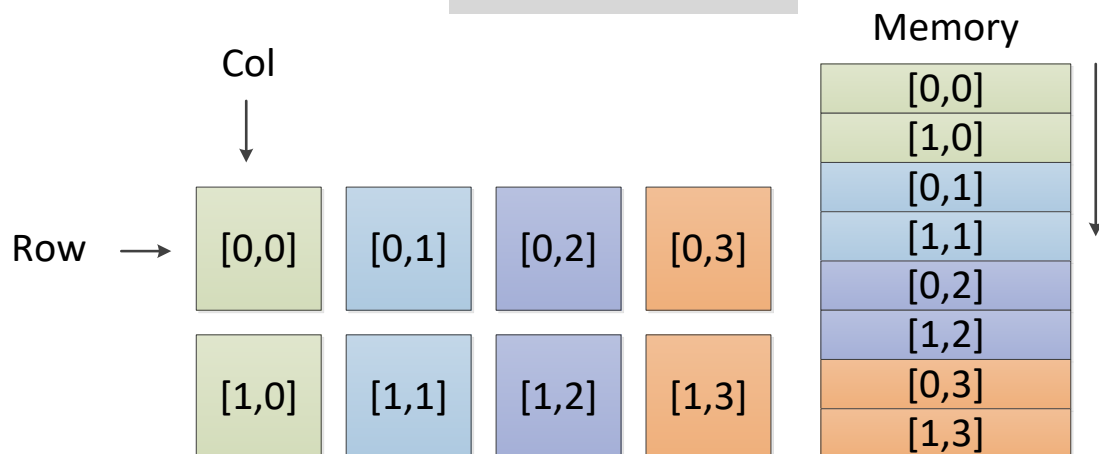
- Array with **more than one index**
 - on physical storage, multi-dimensional array is same as 1d array (*stored continuously in memory space*)
 - logical representation
- To define a 2d array, we specify the **size** of **each** dimension as follows

```
int page[2][3]; // [row][column]  12 34 11
                                   13 56 99
```

- Stored in the "**row-major**" order, i.e.,
 - see next slide



Row-Major



Col-Major

2D Array Initialization

- Assign initial values row by row

```
int page[2][3] = {{1,2,3},{4,5,6}};
```

- Assign initial values to the elements in the order they are arranged:

```
int page[2][3] = {1,2,3,4,5,6};
```

- Only assign initial values to some elements:

```
int page[2][3] = {{1},{4,5}};
```

1	0	0
4	5	0

- If all elements are assigned initial values, the length of the first dimension can be left unspecified:

```
int page[][3] = {1,2,3,4,5,6};
```

```
int page[2][] = {1,2,3,4,5,6}; X
```

Passing 2D Array to Function

- The way to pass a 2D array is similar as the 1D array
- For example: define a function which reads a 2D array as the input and sort each row of the input 2D array

```
void sort2D(int x[][10]) {
```

```
    ...
```

```
}
```


```
void main() {
```

```
    int y[20][10];
```

```
    sort2D(y);
```

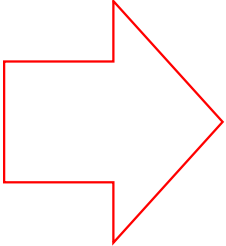
```
}
```

the size of the first dimension
is optional, while the size of
the second dimension must
be given



Example: Swapping Row and Column

- Swap elements of rows and columns of a 2D array with 2 rows and 3 cols
- Output the swapped array (please control the precision of each element, so that it contains only one digit in the decimal part)

1.2	1.34	2.564		1.2	3.0
3	4.123	4		1.3	4.1
				2.6	4.0

Example: Swapping Row and Column

```
// the original array
double array1[2][3] = {{1.2,1.34,2.564},{3,4.123,4}};
// the swapped array
double array2[3][2];
// swap elements of rows and columns
for (int i = 0; i < 2; ++i){
    for (int j = 0; j < 3; ++j){
        array2[j][i] = array1[i][j];
    }
}
// output the swapped array
for (int i = 0; i < 3; ++i){
    for (int j = 0; j < 2; ++j){
        cout << fixed << setprecision(1) << array2[i][j] << "    ";
    }
    cout << endl;
}
```

Example: BMI Program

```
void main() {  
    const int N=10;  
    double data[N][2]; // N records, each holds weight and height  
    int i, position;  
    for (i=0; i<N; i++){  
        cout << "Weigth(kg) Height(m):";  
        cin >> data[i][0];  
        cin >> data[i][1];  
    }  
    for (i=0; i<count; i++) {  
        cout << "BMI for " << i+1 << "is: ";  
        cout << data[i][0]/(data[i][1]*data[i][1]) << endl;  
    }  
}
```

Summary

- Array is a sequence of variables of the *same* data type
- Array elements are *indexed* and can be *accessed* by the use of subscripts, e.g. `array_name[1]`, `array_name[4]`
- Array elements are *stored contiguously* in memory space
- Array Declaration, Initialization, Searching and Sorting

Exercise 1: Calculate Number Power Using Recursion

Calculate the power of the number using the recursion approach

E.g., $5^{10} = \text{power}(5, 10) = 5 * \text{power}(5, 9) = \dots = 5 * \dots * \text{power}(5, 0)$

```
int calculate(int, int);
int main() {
    int base, power, result;
    cout << "Enter base number: ";
    cin >> base;
    cout << "Enter power number: ";
    cin >> power;
    result = calculate(base, power);
    cout << base << "^" << power << " = " << result;
    return 0;
}
```

```
int calculate(int base, int power) {
    if (???) // your codes

        ??? // your codes
    else
        ??? // your codes
}
```

Exercise 2

What is the output produced by the following code?

```
int myArray[4][4], index1, index2;
for (index1 = 0; index1 < 4; index1++)
    for (index2 = 0; index2 < 4; index2++)
        myArray[index1][index2] = index2;

for (index1 = 0; index1 < 4; index1++) {
    for(index2 = 0; index2 < 4; index2++)
        cout << myArray[index1][index2] << " ";
    cout << endl;
}
```


Exercise 3

Consider the following function definition:

```
void too2(int a[], int howMany) {  
    for (int index = 0; index < howMany; index++)  
        a[index] = 2;  
}
```

Which of the following are acceptable function calls?

```
int main(){  
  
    int myArray[29];  
    // function call here  
}
```

- a. `too2(myArray, 29);`
- b. `too2(myArray, 10);`
- c. `too2(myArray, 55);`
- d. `too2(myArray[3], 29);`

Application 1: Maze

- A maze can be defined as a 2D array: `bool maze[H][W];`
 - where H is the height and W is the width
 - `maze[0][0]` is the entry and `maze[H-1][W-1]` is the exit
 - there's a Wall at grid (y, x) if `maze[y][x] == true`, otherwise space.
- 1. Generate a random maze where the ratio of walls is less than R
 - Hint: use `(rand()%100)/100.0` to generate a random number between [0, 1)
- 2. Print the maze. You may use "`char Wall = 177;`" to represent a wall, and "`char space = ' ';`" to represent a space

Application 1: Maze (cont'd)

3. Find a path from entry $(0, 0)$ to exit $(H-1, W-1)$ (Hint: use recursion)

Problem: find a path from (y_0, x_0) to the exit

Application 1: Maze (cont'd)

3. Find a path from entry (0, 0) to exit (H-1, W-1) (Hint: use recursion)

Problem: find a path from (y0, x0) to the exit

Base case?

Recursive problem representation?

How to avoid loops?

How to remember the path when returning from recursion?

Application 2: Snake

- Animation on console
- Define and move a snake
- A snake that can eat beans
- A smarter snake

```

#define H 16 // height
#define W 16 // width

void showGame(int pt[2]) {
    int x, y;
    for (y = 0; y < H; y++) {
        for (x = 0; x < W; x++) {
            if (x == pt[0] && y == pt[1])
                cout << 'o';
            else
                cout << ' ';
        }
        // print row numbers as right border
        cout << y%10 << "\n";
    }
    // print column numbers as bottom border
    for (x = 0; x < W; x++)
        cout << x%10;
}

```

```

int main() {
    int pt[2] = {W/2, H/2}; // init pt at center
    while (true) {
        switch (rand()%4) { // make a random move
            case 0: // to left
                if (pt[0]-1 >= 0) pt[0]--;
                break;
            case 1: // to right
                if (pt[0]+1 < W) pt[0]++;
                break;
            case 2: // to up
                if (pt[1]-1 >= 0) pt[1]--;
                break;
            case 3: // to down
                if (pt[1]+1 < H) pt[1]++;
                break;
        }
        system("cls"); // refresh screen
        showGame(pt); // need #include <Windows.h>
        Sleep(100); // for windows
    }
    return 0;
}

```

```

#define H 16 // height
#define W 16 // width

#define X 0
#define Y 1

void showGame(int pt[2]) {
    int x, y;
    for (y = 0; y < H; y++) {
        for (x = 0; x < W; x++) {
            if (x == pt[X] && y == pt[Y])
                cout << 'o';
            else
                cout << ' ';
        }
        // print row numbers as right border
        cout << y%10 << "\n";
    }
    // print column numbers as bottom border
    for (x = 0; x < W; x++)
        cout << x%10;
}

```

```

int main() {
    int pt[2] = {W/2, H/2}; // init pt at center
    while (true) {
        switch (rand()%4) { // make a random move
            case 0: // to left
                if (pt[X]-1 >= 0) pt[X]--;
                break;
            case 1: // to right
                if (pt[X]+1 < W) pt[X]++;
                break;
            case 2: // to up
                if (pt[Y]-1 >= 0) pt[Y]--;
                break;
            case 3: // to down
                if (pt[Y]+1 < H) pt[Y]++;
                break;
        }
        system("cls"); // refresh screen
        showGame(pt); // need #include <Windows.h>
        Sleep(100);    // for windows
    }
    return 0;
}

```

Define a Snake

```
#define MAX_LEN H*W
```

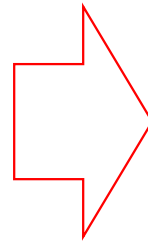
```
int snake[MAX_LEN][2];
```

```
snake[0][X] = 0; // initial x of snake head
```

```
snake[0][Y] = 0; // initial y of snake head
```

```
int len = 1; // initial snake length
```


Move a Snake



Move a Snake

```
bool moveSnakeTo(int snake[MAX_LEN][2], int len, int dst[2]) {
    if (dst[X] < 0 || dst[X] >= W || dst[Y] < 0 || dst[Y] >= H)
        return false;
    if (onSnakeBody(snake, len, dst))
        return false;
    // dst not adjacent to head
    if (abs(dst[X]-snake[0][X]) + abs(dst[Y]-snake[0][Y]) != 1)
        return false;

    for (int i = len-1; i > 0; i--) {
        snake[i][X] = snake[i-1][X];
        snake[i][Y] = snake[i-1][Y];
    }
    snake[0][X] = dst[X];
    snake[0][Y] = dst[Y];
    return true;
}
```

```
// check if a point is on snake body
```

```
bool onSnakeBody(int snake[MAX_LEN][2], int len, int pt[2]) {  
    for (int i = 0; i < len; i++) {  
        if (pt[X] == snake[i][X] && pt[Y] == snake[i][Y])  
            return true;  
    }  
    return false;  
}
```

```
// generate a bean at a random position
```

```
void generateBean(int snake[MAX_LEN][2], int len, int bean[2]) {  
    do {  
        bean[X] = rand()%W;  
        bean[Y] = rand()%H;  
    } while (onSnakeBody(snake, len, bean));  
}
```

Find a Path to Bean

- Again: use recursion
- **Problem**: given the current snake position, find a path to the bean
- the smallest problem?
- recursive problem representation?
- avoid loop?
- remember path?

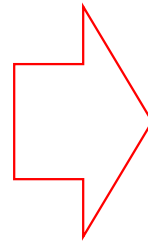
```

bool findPathTo1(int snake[MAX_LEN][2], int len, int dst[2], bool visit[H][W], int path[MAX_LEN][2], int step) {
    if (step == MAX_LEN) return false;
    // basic case of recursion, dst is snake head
    if (dst[X] == snake[0][X] && dst[Y] == snake[0][Y]) {
        path[step][X] = dst[X];
        path[step][Y] = dst[Y];
        return true;
    }
    // check if dst is valid
    if (dst[X] < 0 || dst[X] >= W || dst[Y] < 0 || dst[Y] >= H || onSnakeBody(snake, len, dst))
        return false;
    // we have four possible move directions
    int next_pt[4][2];
    next_pt[1][X]=snake[0][X]-1; next_pt[1][Y]=snake[0][Y]; // left
    next_pt[2][X]=snake[0][X]+1; next_pt[2][Y]=snake[0][Y]; // right
    next_pt[0][X]=snake[0][X]; next_pt[0][Y]=snake[0][Y]-1; // top
    next_pt[3][X]=snake[0][X]; next_pt[3][Y]=snake[0][Y]+1; // bottom
    for (int i = 0; i < 4; i++) {
        // make a copy of the original snake
        int tmp[MAX_LEN][2];
        for (int j = 0; j < len; j++) {
            tmp[j][X] = snake[j][X];
            tmp[j][Y] = snake[j][Y];
        }
        if (!moveSnakeTo(tmp, len, next_pt[i]) || visit[next_pt[i][Y]][next_pt[i][X]])
            continue;
        visit[next_pt[i][Y]][next_pt[i][X]] = true;
        if (findPathTo1(tmp, len, dst, visit, path, step + 1)) {
            path[step][X] = snake[0][X];
            path[step][Y] = snake[0][Y];
            return true;
        }
    }
    return false;
}

```

Eat A Bean

		o	0				
			1	2			
				3			
				4	5	6	
						7	



		0	1				
			2	3			
				4			
				5	6	7	
						8	