



# **EE2331 Data Structures and Algorithms**

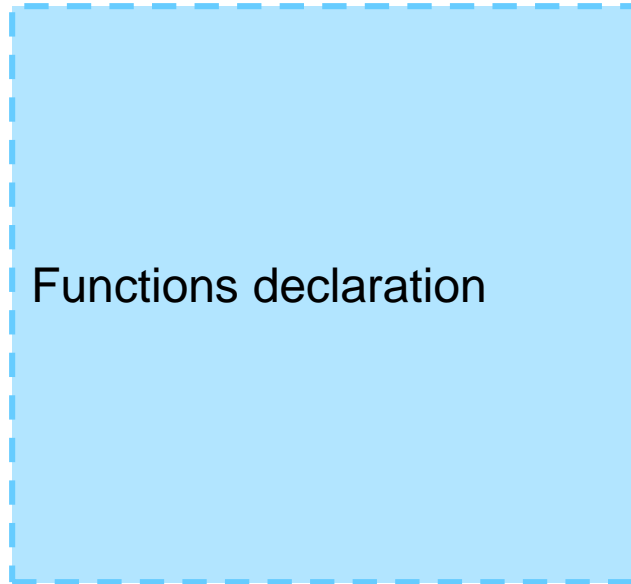
## **C++ Programming Review**

# Outline

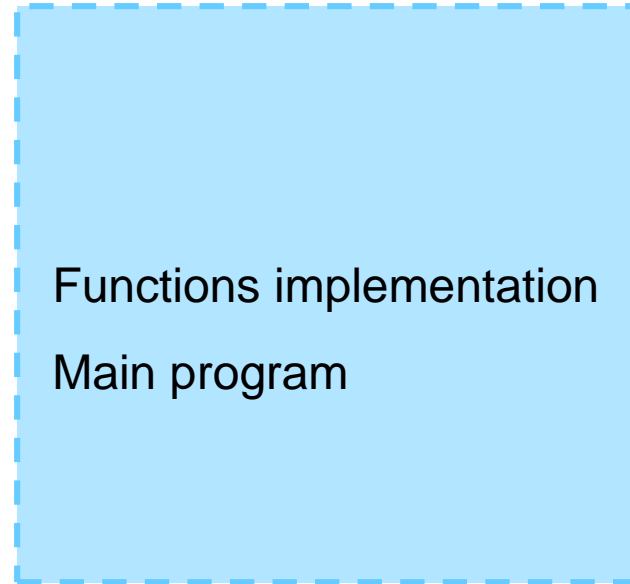
- Standard Libraries
- Basic Data Types
- Arithmetic, Bitwise, Logical Operators
- Control Structures
- Pointers
- Arrays
- Composite Structures
- Parameter Passing in Functions
- Standard I/O
- Pseudo Code
- Suggestion for Good Programming Practice

# C++ File Structure

Header file (.h)



Program file (.cpp)



A header file commonly contains forward declarations of subroutines. Programmers who wish to declare functions in more than one source file can place such declaration in a single header file, which other code can then **include** whenever the header contents are required.

# A Basic C++ Program

```
#include <cstdio>           //include directive(s)
```

```
int z = 0;                  //global variable(s) declaration & initialization
```

```
void hello(int a) {        //function(s) declaration & implementation  
    printf("%d\n", a + z);  
}
```

```
int main(int argc, char *argv[]) {  
    int x = 10;             //local variable(s) declaration & initialization  
    hello(x);               //function call  
    return 0;               //return to O.S. (0 = successful completion)  
}
```

# Main Function

- There are two declarations of main that must be allowed:
  - `int main()` // without arguments
  - `int main(int argc, char** argv)` // with arguments
- The **return type** of main must be **int**.
  - Return **zero** to indicate **success** and **non-zero** to indicate **failure**.
  - You are not required to explicitly write a return statement in `main()`. If you let `main()` return without an explicit return statement, it's the same as if you had written `return 0;`.
    - `int main() { }` // equivalent to the next line
    - `int main() { return 0; }`
  - There are two macros, **EXIT\_SUCCESS** and **EXIT\_FAILURE**, defined in `<cstdlib>` that can also be returned from `main()` to indicate success and failure, respectively.

# Command Line Arguments

 C:\> assign1.exe dat1.txt data2 ... xxx

Where's the  
location of  
your  
compiled  
program?

↑                      ↑                      ↑                      ↑  
Your compiled program    1<sup>st</sup> argument    2<sup>nd</sup> argument    ...     $n^{\text{th}}$   
  
argv[0]                      argv[1]                      argv[2] ... argv[n]

Total no. of arguments (i.e. **argc** =  $n + 1$ )

```
int main(int argc, char *argv[]) {  
    ...  
}
```

**argc**: count  
argv: value

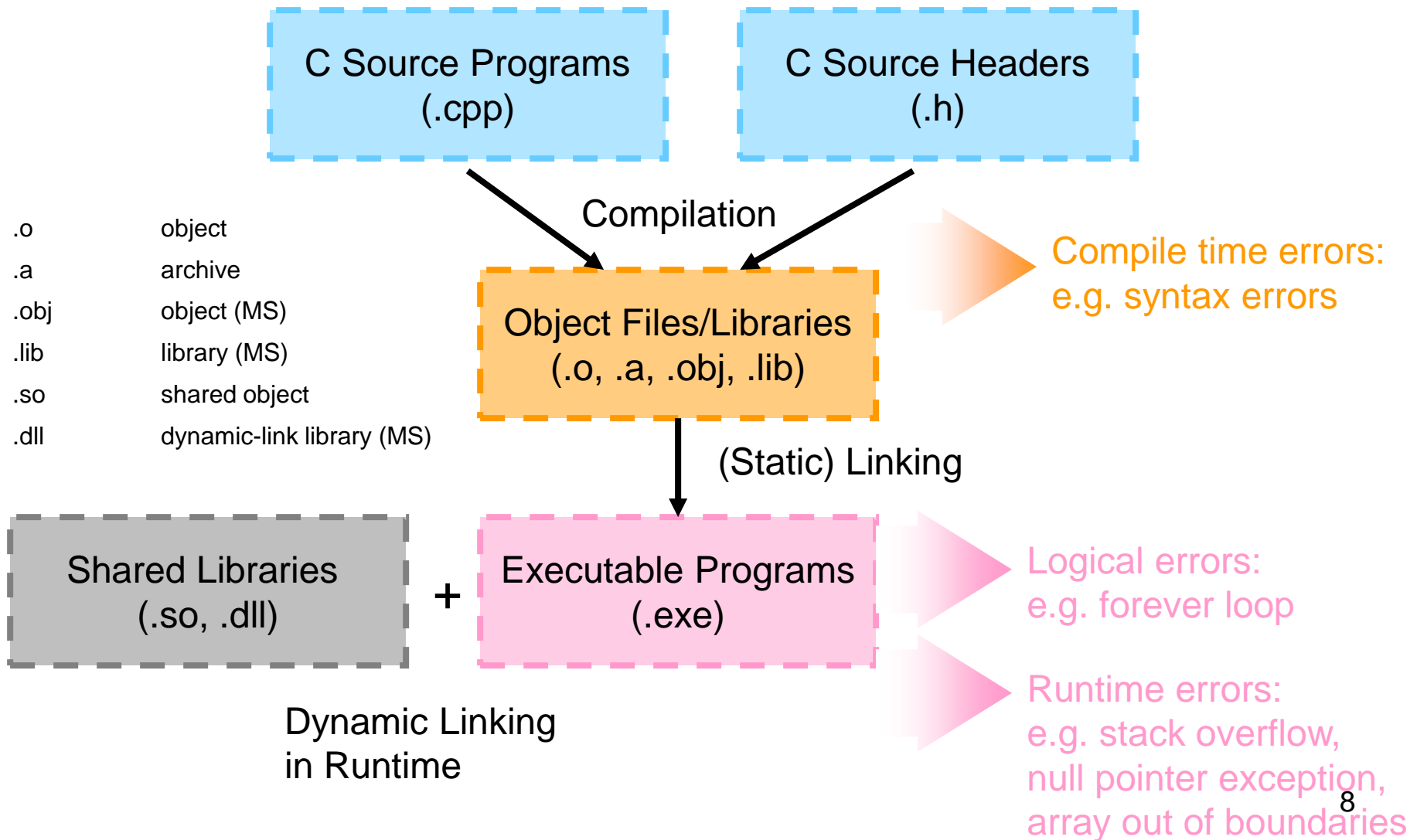
# Command Line Arguments

```
int main(int argc, char *argv[]) {  
    printf("argc = %d\n", argc);  
    for (int i = 0; i < argc; i++)  
        printf("argv[%d] = %s\n", i, argv[i]);  
}
```

```
>ex1_2.exe 123 abc
```

```
argc = 3  
argv[0] = ex1_2.exe           //name of the program  
argv[1] = 123                 //string, not integer  
argv[2] = abc
```

# The Building Process





# Common Standard Library Header

## ■ `<cstdio>`

- Standard I/O facilities: `printf()`, `scanf()`, `getchar()`, `fopen()`, `fclose()`, etc

## ■ `<cstdlib>`

- Standard utility functions: `malloc()`, `free()`, `rand()`, etc

## ■ `<cstring>`

- String functions: `strcpy()`, `strcmp()`, `memset()`, etc

## ■ `<iostream>`

- Perform both input and output operations with the stream objects: **`cin`** and **`cout`**

# Comments

```
/* Block comment 1 */
```

```
/*  
 * Block comment 2  
 */
```

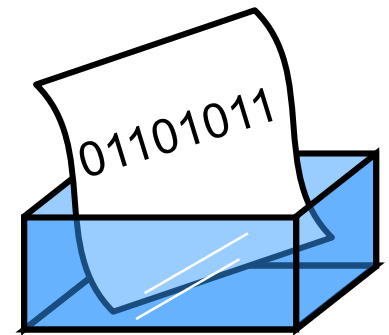
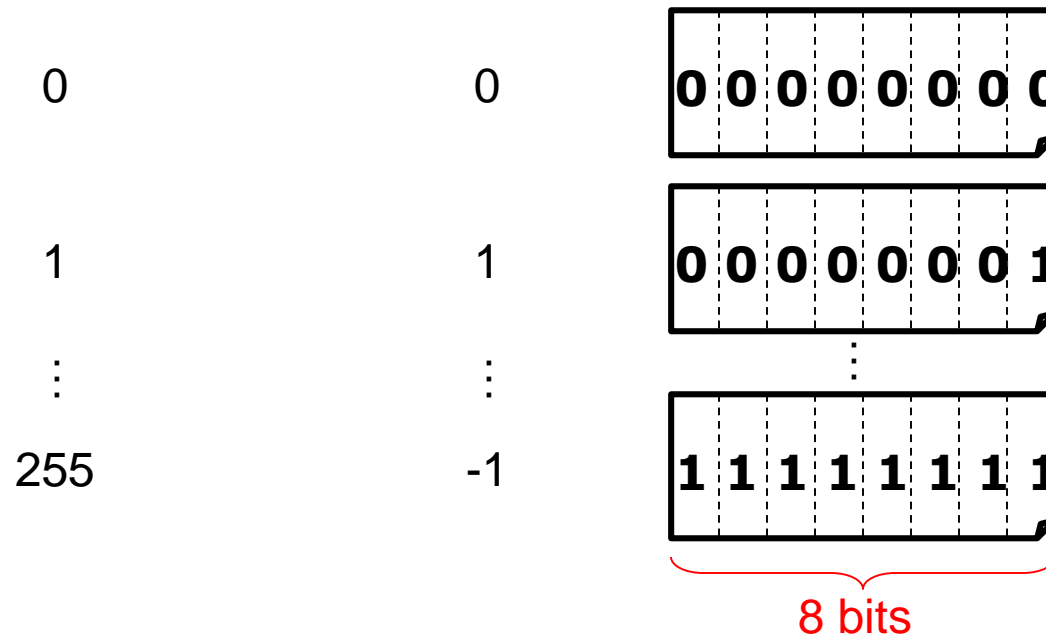
```
// Line comment
```

# Basic Data Type

- Data itself does not have meaning. Its meaning depends on how you interpret the data.
- Char
  - 1 Byte (8 bits)

Unsigned

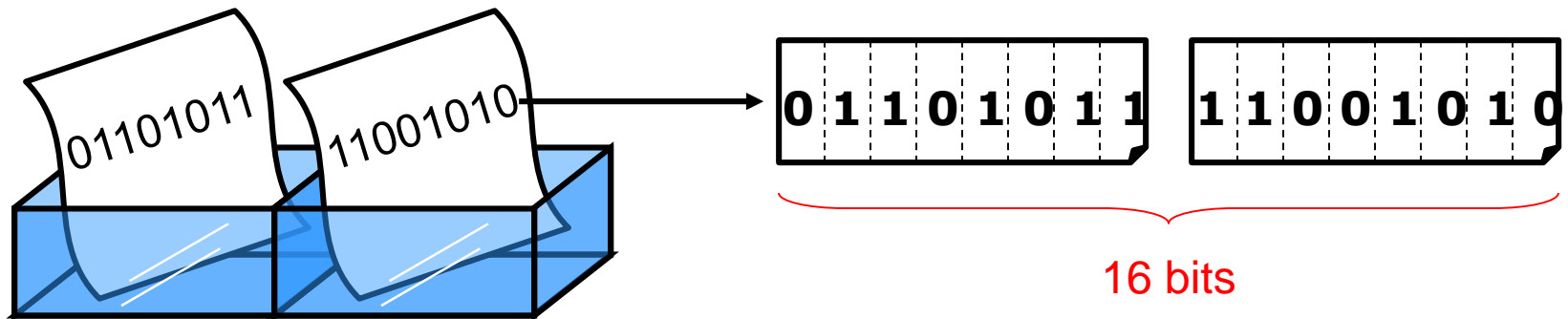
Signed (2's complement)



# Basic Data Type

## ■ Short Int

■ 2 Bytes (16 bits)

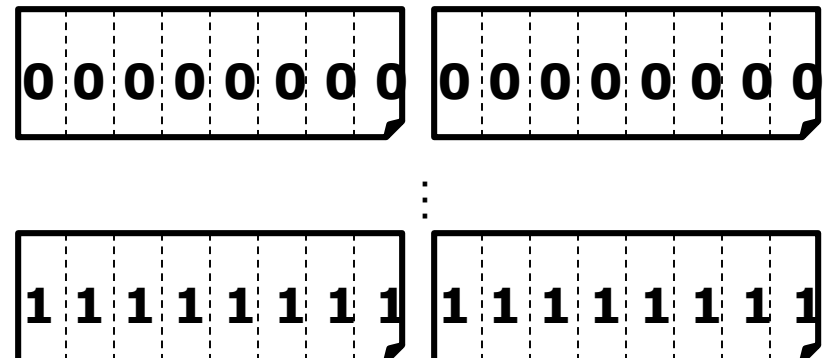


Unsigned

Signed (2's complement)

0  
:  
65535

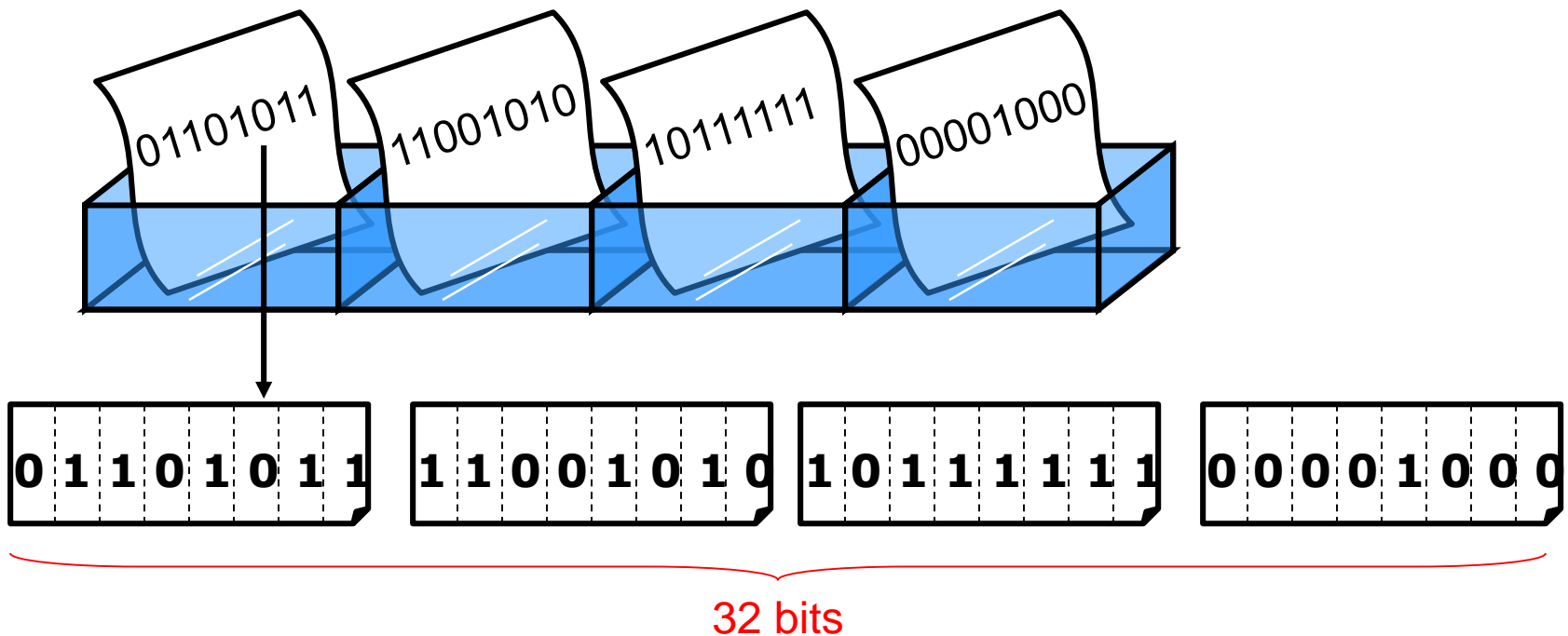
0  
:  
-1



# Basic Data Type

- Int / Long Int

- 4 Bytes (32 bits)



What are the largest and smallest numbers can an “int” be represented?

# Primitive Data Types in C++

| Data type          | Size (byte) | Interpretation/representation     | Range of values   |
|--------------------|-------------|-----------------------------------|---|
| bool               | 1           | Boolean (not available in C)      | false or true   |
| char               | 1           | signed number (2's complement)    | -128 to 127   |
| unsigned char      |             | unsigned number                   | 0 to 255  |
| int                | 4           | signed number (2's complement)    | $-2^{31}$ to $2^{31}-1$                                 |
| unsigned int       |             | unsigned number                   | 0 to $2^{32}-1$   |
| short              | 2           | signed number (2's complement)    | $-2^{15}$ to $2^{15}-1$                                 |
| unsigned short     |             | unsigned number                   | 0 to $2^{16}-1$   |
| long               | 4           | signed number (2's complement)    | $-2^{31}$ to $2^{31}-1$                                 |
| unsigned long      |             | unsigned number                   | 0 to $2^{32}-1$   |
| long long          | 8           | signed number (2's complement)    | $-2^{63}$ to $2^{63}-1$                                 |
| unsigned long long |             | unsigned number                   | 0 to $2^{64}-1$   |
| float              | 4           | IEEE 32-bit floating point number | $\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$   |
| double             | 8           | IEEE 64-bit floating point number | $\pm 5 \times 10^{-324}$ to $\pm 1.798 \times 10^{308}$ |
| <b>pointer</b>     | <b>4</b>    | <b>memory address</b>             | <b>0 to <math>2^{32}-1</math></b>                       |

# Operators in C++

| Operator              | Symbol                | Description   |
|-----------------------|-----------------------|---|
| Assignment            | =                     |   |
| Arithmetic            | +, -, *, /, %         |   |
| Increment, decrement  | ++, --                |   |
| Unary minus           | -                     |   |
| Comparison            | ==, !=, <, <=, >, >=  |   |
| Logical               | !, &&,                |   |
| Bitwise               | ~, &,  , ^, <<, >>    |   |
| insertion, extraction | cout << s<br>cin >> i | insertion to an output stream<br>extraction from an input stream  |
| Member and pointer    | x[i]                  | subscript (x is an array or a pointer)  |
|                       | *x                    | indirection, dereference (x is a pointer)   |
|                       | &x                    | reference (address of x)  |
|                       | x->y                  | <b>structure dereference</b><br>(x is a pointer to object/struct; y is a member of the object/struct pointed to by x) |
|                       | x.y                   | <b>structure reference</b> (x is an object or struct; y is a member of x)   |

# Use of Variables

## ■ Declaration

- Given an identifier (variable name), you specify the data type of it and hence implicitly reserve the required memory space.

## ■ Initialization

- Variables must be initialized before being used.
- The following code will cause **compilation error**.

```
int a;  
cout << a;
```



# Arithmetic Operators

## ■ Addition

```
int a, b, c;  
a = 1;  
b = 2;  
c = a + b;  
printf("%d\n", c);
```

## ■ Mind the **overflow** problem

```
int a, b, c;  
a = b = 2147483647; //the largest value of signed int  
c = a + b;  
printf("%d\n", c);
```

# Arithmetic Operators

## ■ Subtraction

```
int a, b, c;  
a = 1;  
b = 2;  
c = a - b;  
printf("%d\n", c);
```

## ■ Mind the **underflow** problem

```
int a, b, c;  
a = -2147483648; //the smallest value of signed int  
b = 2147483647;  //the largest value of signed int  
c = a - b;  
printf("%d\n", c);
```

# Arithmetic Operators

## ■ Division

```
int a, b, c;  
a = 5;  
b = 2;  
c = a / b;  
printf("%d\n", c);    // output is 2
```

■ Integer truncation occurs

■ What is the result of a float divided by an int?

# Arithmetic Operators

## ■ Remainder (Modulus Operator)

```
int a, b, c;  
a = 5;  
b = 2;  
c = a % b;  
printf("%d\n", c);
```

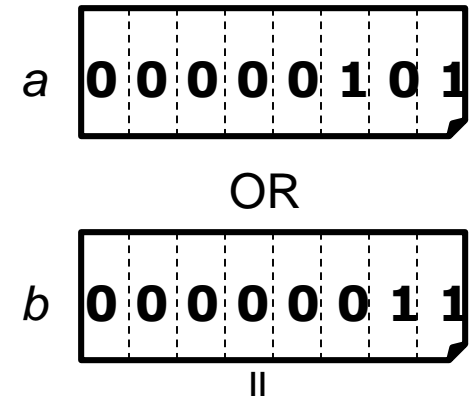
## ■ When to use it?

- Generate periodic values
- To wrap around the array index (in Queue)
- To determine the hash key (in Hash table)

# Bitwise & Logical Operators

## ■ Bitwise OR

```
int a, b, c;  
a = 5;  
b = 3;  
c = a | b;  
printf("%d\n", c);
```



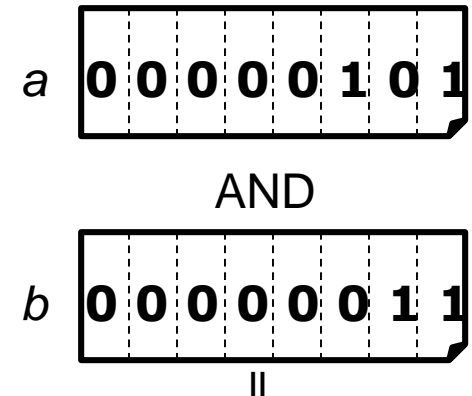
## ■ Logical OR

```
int a, b, c;  
a = 5;  
b = 3;  
c = a || b;  
printf("%d\n", c);
```

# Bitwise & Logical Operators

## ■ Bitwise AND

```
int a, b, c;  
a = 5;  
b = 3;  
c = a & b;  
printf("%d\n", c);
```



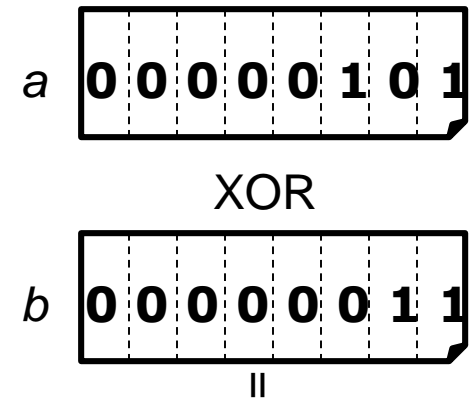
## ■ Logical AND

```
int a, b, c;  
a = 5;  
b = 3;  
c = a && b;  
printf("%d\n", c);
```

# Bitwise Operators

## ■ Exclusive OR

```
int a, b, c;  
a = 5;  
b = 3;  
c = a ^ b;  
printf("%d\n", c);
```



## ■ When to use it?

- Interchange two variables (in Bubble Sort)

# Bitwise Operators

## ■ Left Shift (x2)

```
int a, b;  
a = 5;  
b = a << 1;  
printf("%d\n", b);
```

*a*

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

*b*

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

## ■ Right Shift (/2)

```
int a, b;  
a = 5;  
b = a >> 1;  
printf("%d\n", b);
```

*b*

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|



# Bitwise Operators

## ■ 1's complement

```
int a, b;  
a = 5;  
b = ~a;  
printf("%d\n", b);
```

*a*

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

*b*

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

*Note: both a and b  
should be 4-byte long*

# Variable Assignments

## ■ Example 1

```
int a, b, c;  
a = b = c = 5;  
printf("%d\n", a);
```

## ■ Example 2

```
int a = 5, b = 5, c = 5;  
a = b == c;  
printf("%d\n", a);
```

# Variable Assignments

## ■ Example 3

```
int a = 5, b = 5, c = 5;  
a = b > c;  
printf("%d\n", a);
```

## ■ Example 4

```
int a = 5, b = 5, c = 5;  
a = ++b > c;  
printf("%d %d %d\n", a, b, c);
```

## ■ Example 5

```
int a = 5, b = 5, c = 5;  
a = b++ > c;  
printf("%d %d %d\n", a, b, c);
```

# Typecasting

## ■ Example 1 - Implicit

```
int a;  
float b = 10.5;  
a = b; // precision loss with warning  
printf("%d %f\n", a, b); // 10 10.5
```

## ■ Example 2 - Explicit

```
int a;  
float b = 10.5;  
a = (int) b; // still precision loss but NO warning  
printf("%d %f\n", a, b); // 10 10.5
```

# Typecasting

## ■ Example 3

```
int a = 3;  
int b = 2;  
int c = 4;  
cout << a / b * c << endl; // output is 4 !!  
cout << a * c / b << endl; // output is 6
```

- The resultant type of an arithmetic operation will be promoted to the type of the operators with larger precision.
  - $\text{int} / \text{int} \rightarrow \text{int}$
  - $\text{float} / \text{int} \rightarrow \text{float}$

# Octal and Hex. Assignment

## ■ Example 1 - Octal

```
int a;  
a = 022;           // leading zero  
printf("%d\n", a); // 18
```

## ■ Example 2 - Hex

```
int a;  
a = 0x22;  
printf("%d\n", a); // 34
```

# Control Structures

# If-then-else

■ `? :` (equivalent to if-then-else)

■ *expression `?` true instruction `:` false instruction;*

```
if (a < b)
    min = a;
else
    min = b;
```

```
min = a < b ? a : b;
```



# For-Loop and While-Loop

- for-loop and while-loop are interchangeable

```
for (initialization; loop_test; loop_counting) {  
    //loop-body  
}
```

```
initialization;  
while (loop_test) {  
    //loop-body  
    loop_counting;  
}
```

# Jump Statements

- Jump statements allow the early termination of loops
- These cause unconditional branches
  - `goto` is bad practice and will not be dealt with
  - `break` will exit the inner most loop
  - `continue` will force the next iteration
  - `return` will return to the calling function
  - `exit` will quit the program

# Breaking Out Loops Early

```
for (i = 0; i < n; i++) {  
    ...  
    if (...) break;           //to break out the for-loop  
    ...  
}
```

```
while (...) {  
    ...  
    if (...) continue;       //to skip the rest part of current iteration,  
                             //and continue for next iteration  
    ...  
}
```

# Bad Styles of Loop

```
// DON'T use != (Not equal) to test the end of a range
```

```
for (i = 1; i != n; i++) {  
    //loop body  
}
```

```
// How does the loop behave if n happens to be zero or negative?
```

```
// DON'T modify the value of the loop-counter inside the loop body of a for-loop.
```

```
for (i = 1; i <= n; i++) {  
    //main body of the loop  
    if (testCondition)  
        i = i + displacement;
```

```
    //i++ is executed before going back to top of the loop
```

```
}
```

# Breaking Out Functions Early

```
void func(...) {  
    ...  
    if (...) return;           //to break out the function  
    ...  
}
```

```
int func(...) {  
    ...  
    if (...) return 0;         //to break out the function, and  
                                //return a value to the calling function  
    ...  
}
```

# Breaking Out Programs Early

```
void func(...) {  
    ...  
    if (...) exit(0);           //to terminate the program, and  
                                //return normal exit value 0 to operating  
    ...                         //system!  
}
```

```
int main(...) {  
    ...  
    if (...) exit(1);           //to terminate the program, and  
                                //return abnormal exit value 1 to  
    ...                         //operating system!  
    return 0;                   //normal completion of the program  
}
```

# Loop Design

- Find the maximum value in an array of integers.
- Any mistake in this program?

```
int max(int a[], int n) {           //n = no. of elements in a[]
    int m = 0;                     // variable to store the max value
    for (int i = 0; i < n; i++)
        if (a[i] > m)
            m = a[i];
    return m;
}
```

# Loop Design

- **Precondition/Postcondition** is a condition (predicate) that must always be true just prior/after to the execution of some section of code
  - Often, preconditions/postconditions are simply included in the documentation of the affected section of code.
  - If a precondition is violated, the effect of the section of code becomes **undefined** and thus may or may not carry out its intended work.
- **Loop invariant** is a condition that is necessarily true immediately before and after each iteration of a loop.
  - An appropriate invariant should also **present the goal** of the loop such that it is used to help prove the correctness of an algorithm.



# Loop Design

- Find the maximum value in an array of integers.

```
// precondition:  $n > 0$  and  $a[]$  is unordered
int max(int a[], int n) {
    int m = a[0];
    // m equals the maximum value in  $a[0...0]$ 
    int i = 1;
    while (i < n) {
        // invariant: m equals the maximum value in  $a[0...i-1]$ 
        if (m < a[i])
            m = a[i];        // m equals the maximum value in  $a[0...i]$ 
        i++;
        // invariant: m equals the maximum value in  $a[0...i-1]$ 
    }
    // postcondition: m equals the maximum value in  $a[0...i-1]$ , and  $i == n$ 
    return m;
}
```

# Pointers and Arrays

# Pointers

*Note: The actual size of integers and pointers are 4-byte long*

```
① int a, *p;  
② a = 5;  
③ p = &a;
```

a: value of *a* (i.e. 5)

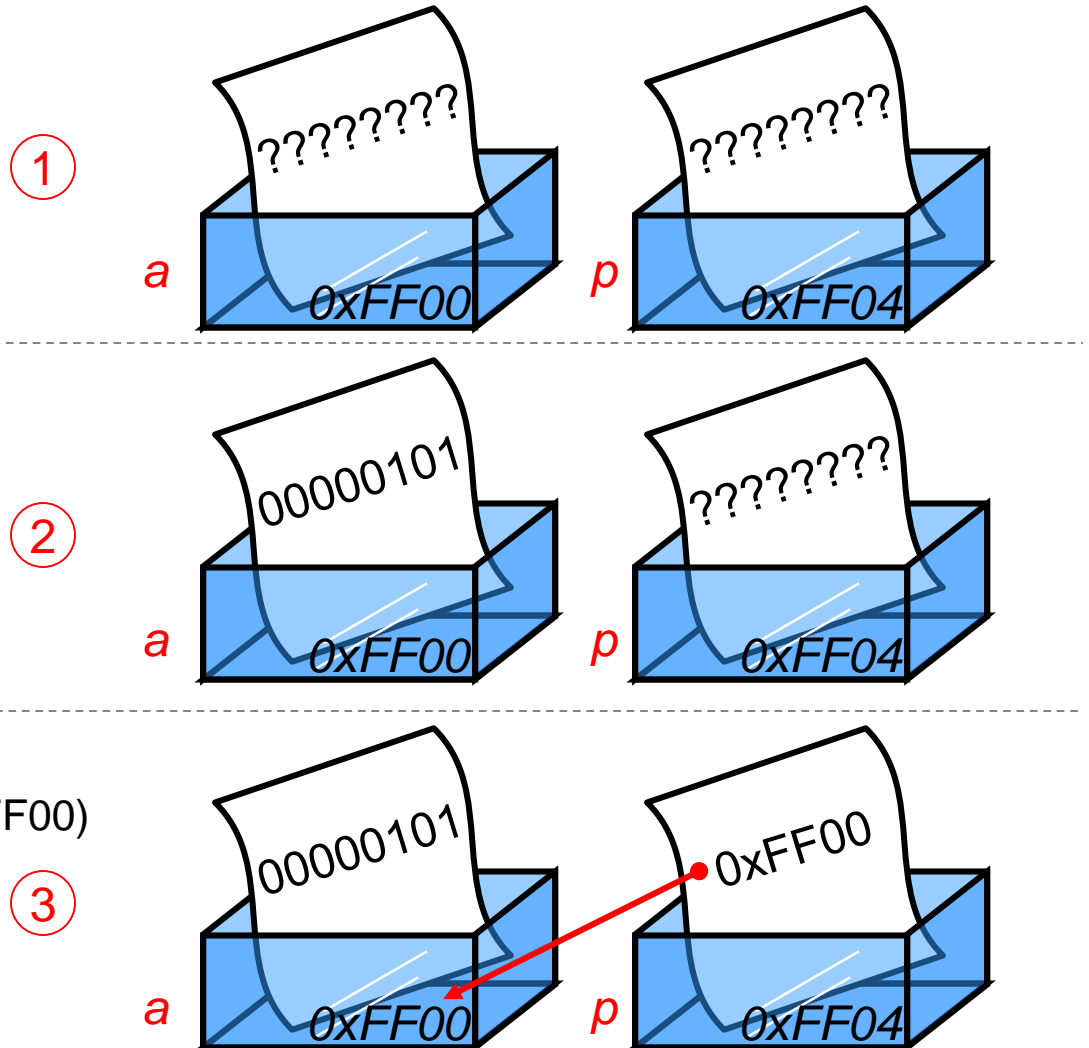
&a: address of *a* (i.e. 0xFF00)

\*a: ?

p: value of *p* == address of *a* (i.e. 0xFF00)

&p: address of *p* (i.e. 0xFF04)

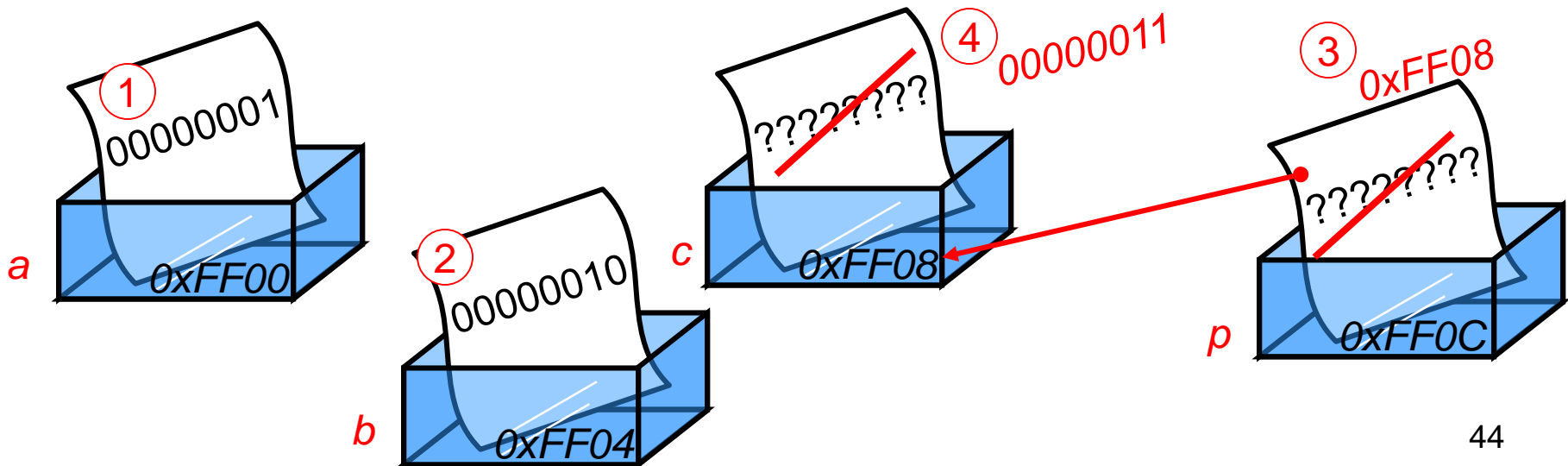
\*p: value pointed by *p* (i.e. 5)



# Pointers

```
int a, b, c, *p;
```

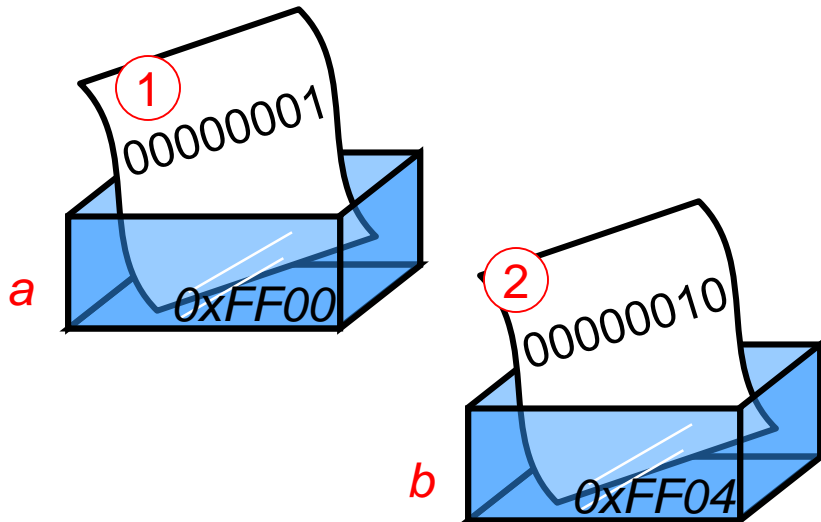
- ① `a = 1;`
- ② `b = 2;`
- ③ `p = &c;`
- ④ `*p = a + b;`  
`printf("%d %d\n", c, *p);`



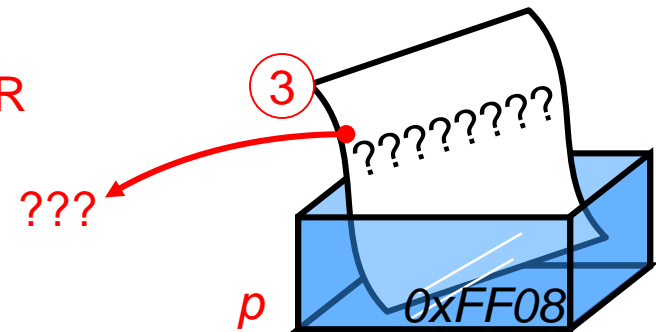
# Pointers

```
int a, b, *p;
```

```
① a = 1;  
② b = 2;  
③ *p = a + b;  
printf("%d\n", *p);
```



ERROR



# Pointers

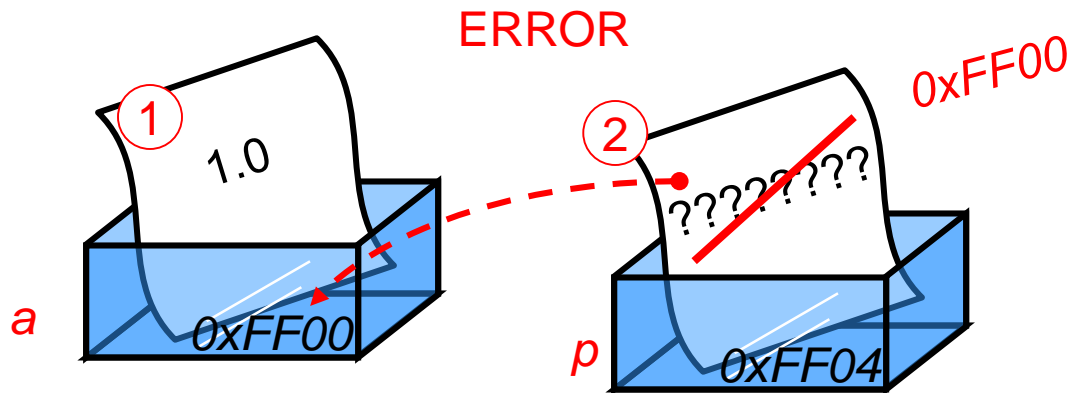
```
float a;
```

```
int *p;
```

① a = 1.0;

② p = &a;

③ printf("%d\n", \*p);

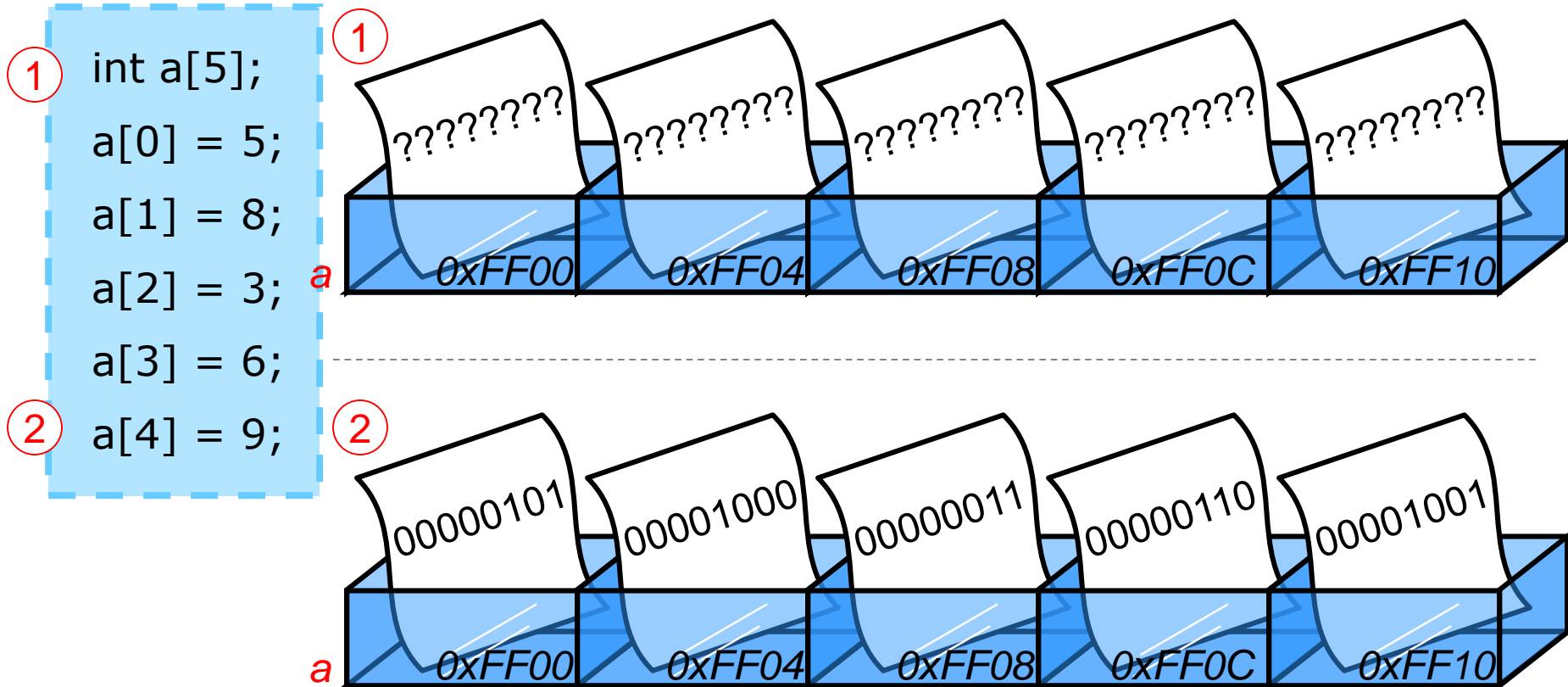


*Note: this is not the actual bit representation of floating numbers. It is simplified for easy drawing*

# Pointers Example

```
int x = 1, y = 2;  
int *a, *b, *c;  
  
a = &x;  
b = &y;  
printf("%d %d %d %d\n", x, y, *a, *b);  
  
c = a;           // swap a with b  
a = b;  
b = c;  
printf("%d %d %d %d\n", x, y, *a, *b);
```

# Creation of Array



*Note: the elements of integer array should be 4-byte long.*

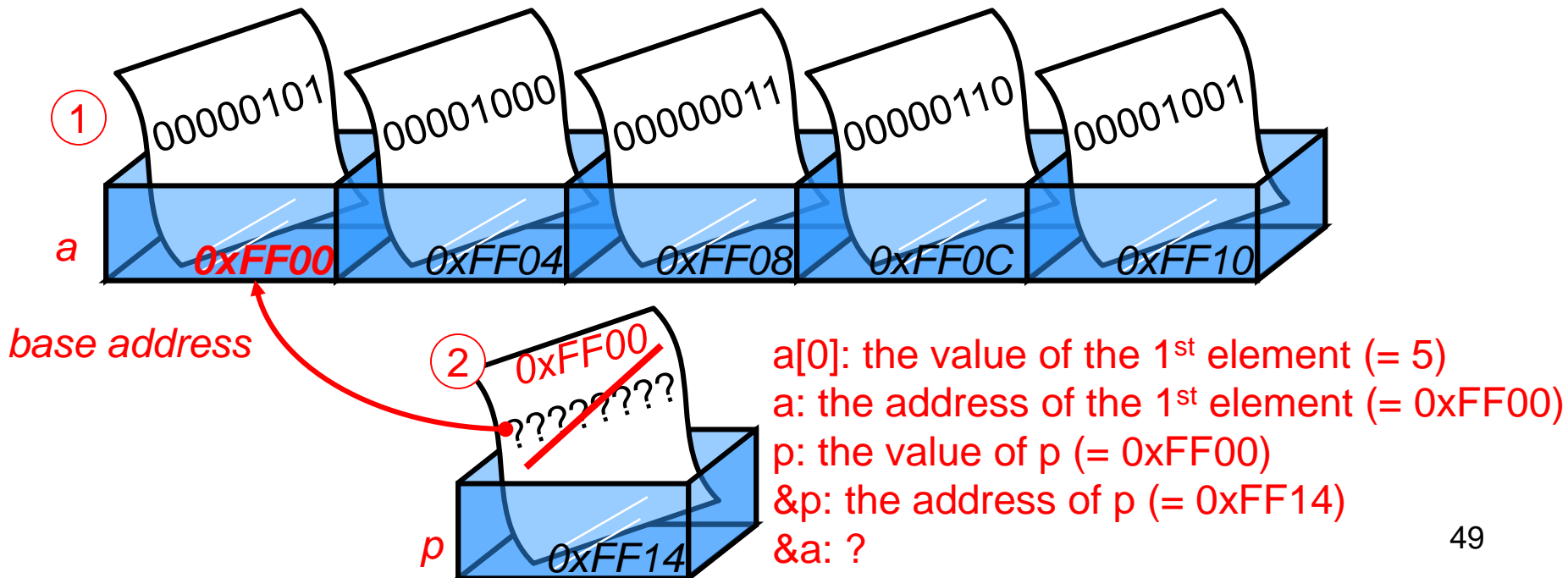


# Base Address of Arrays

Initialization, set size implicitly

```
① int a[] = {5, 8, 3, 6, 9};  
   int *p;  
② p = a; //why not p = &a; ??
```

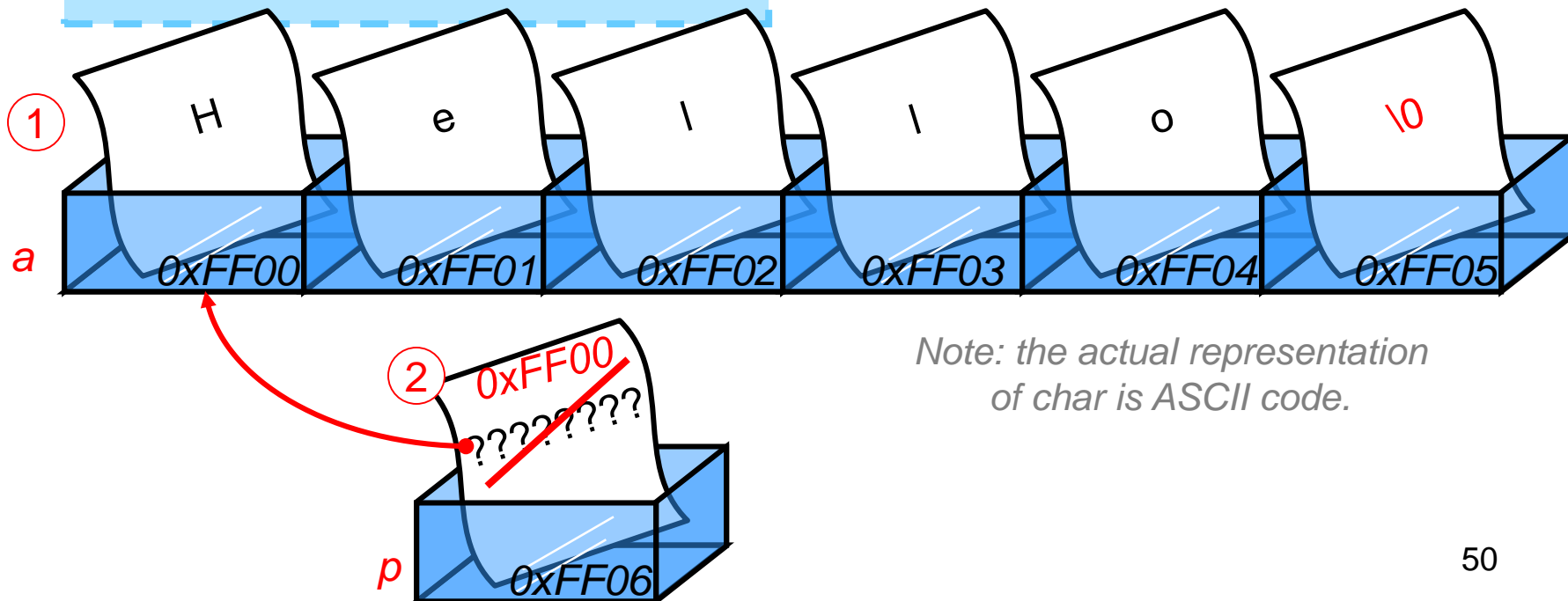
The array variable 'a' is interpreted as a pointer pointing to the first element (base address) of the array.



# C-String (Character Array)

```
① char a[] = "Hello";  
   char *p;  
② p = a;  
   printf("%d\n", sizeof(a));
```

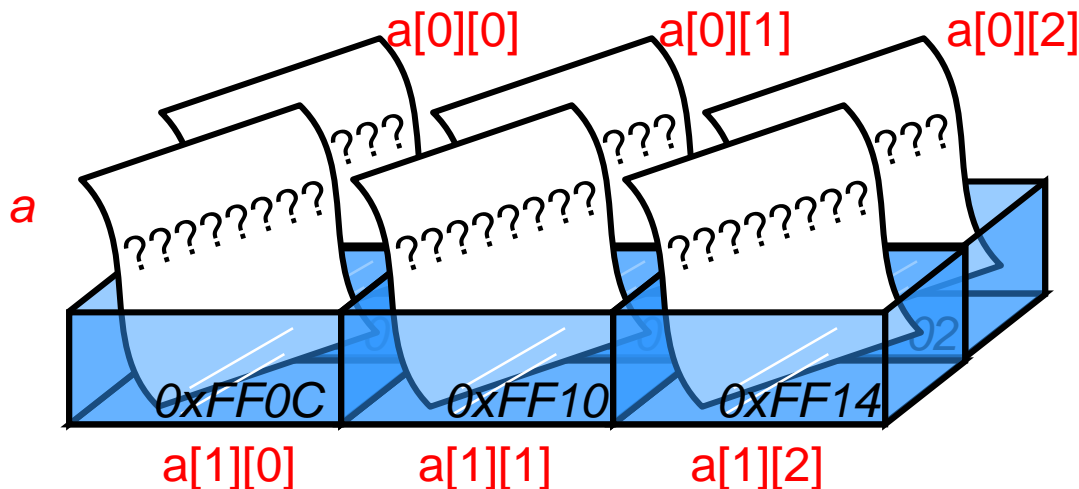
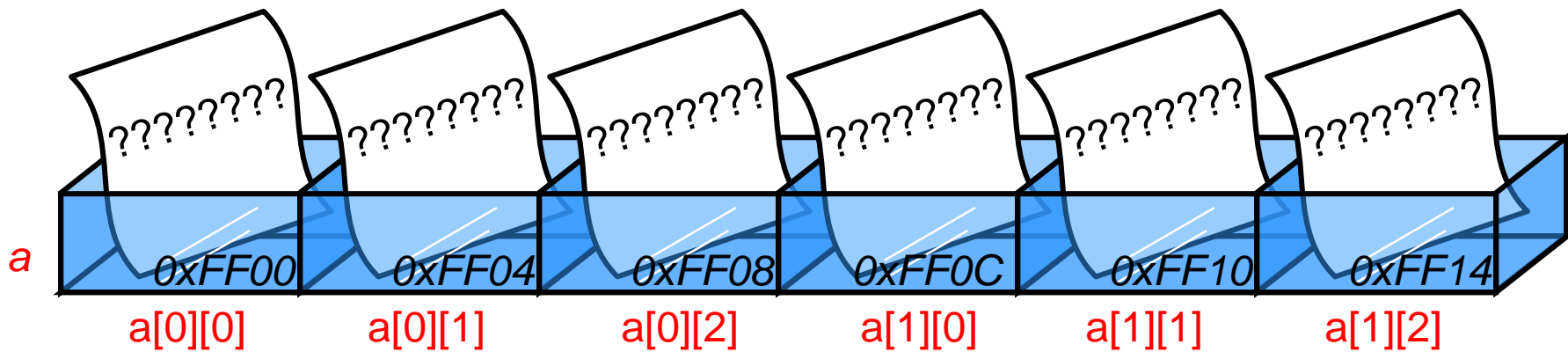
Null character ('\0') filled at the end  
of character array (string)



# 2D Arrays

```
int a[2][3]; //2 rows, 3 columns
```

*Multi-dimensional arrays are mapped to the linear address space of the computer system.*



*In C/C++, elements of a multi-dimensional array are arranged in **row-major order**.*

# Size of Array

- The size of array is **fixed** and **predetermined**
- Cannot declare an array with variable size

```
#define n 10    //n is a macro  
int i, a[n];    //ok, n is substituted by 10 during compilation  
for (i = 0; i < n; i++)  
    a[i] = i;
```

```
int n=100;      // n is a variable  
int i, a[n];    // compilation error  
for (i = 0; i < n; i++)  
    a[i] = i;
```

# Boundaries of Array

- C/C++ will not check the boundaries of array

```
int a[10];  
a[11] = 0;    //allow to run (dangerous!)  
              //but result is unpredictable!
```

- It is the responsibility of programmers to ensure not going out the boundaries

```
int a[10];  
int i = 11;  
if (i >= 0 && i < 10) a[i] = ...; //boundaries checking
```

# Array Mapping Functions

- Today's PCs are byte-addressable
- Let
  - $i$  = row index
  - $j$  = column index
  - cols = number of columns in a row
  - esize = size of an element (no. of bytes)      e.g. 4 for integer
- address of  $b[i] = \text{base}(b) + i * \text{esize}$ 
  - $\text{base}(b)$  = address of  $b[0]$
- address of  $a[i][j] = \text{base}(a) + (i * \text{cols} + j) * \text{esize}$ 
  - $\text{base}(a)$  = address of  $a[0][0]$

# Treat Array as Pointer

- The 2 implementations are equivalent:

```
int sum1(int a[], int n) {  
    int t = 0;  
    for (int i = 0; i < n; i++)  
        t += a[i];  
    return t;  
}
```

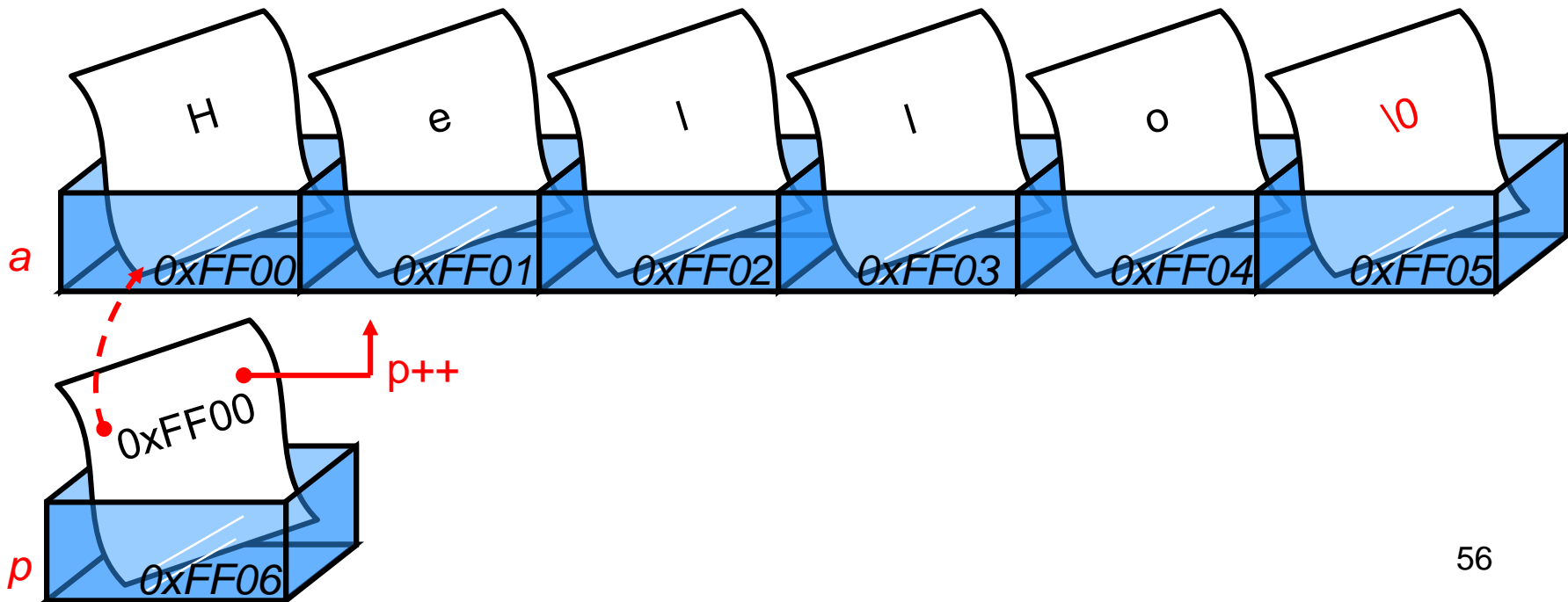
```
int sum2(int *a, int n) {  
    int t = 0;  
    for (int i = 0; i < n; i++)  
        t += *(a+i);  
    return t;  
}
```

Because **esize** is implied by the data type, the arithmetic operation of pointer implicitly takes **esize** into account.

So,  $(a+i)$  = physical address of  $a+i*\text{esize}$

# Arrays and Pointers

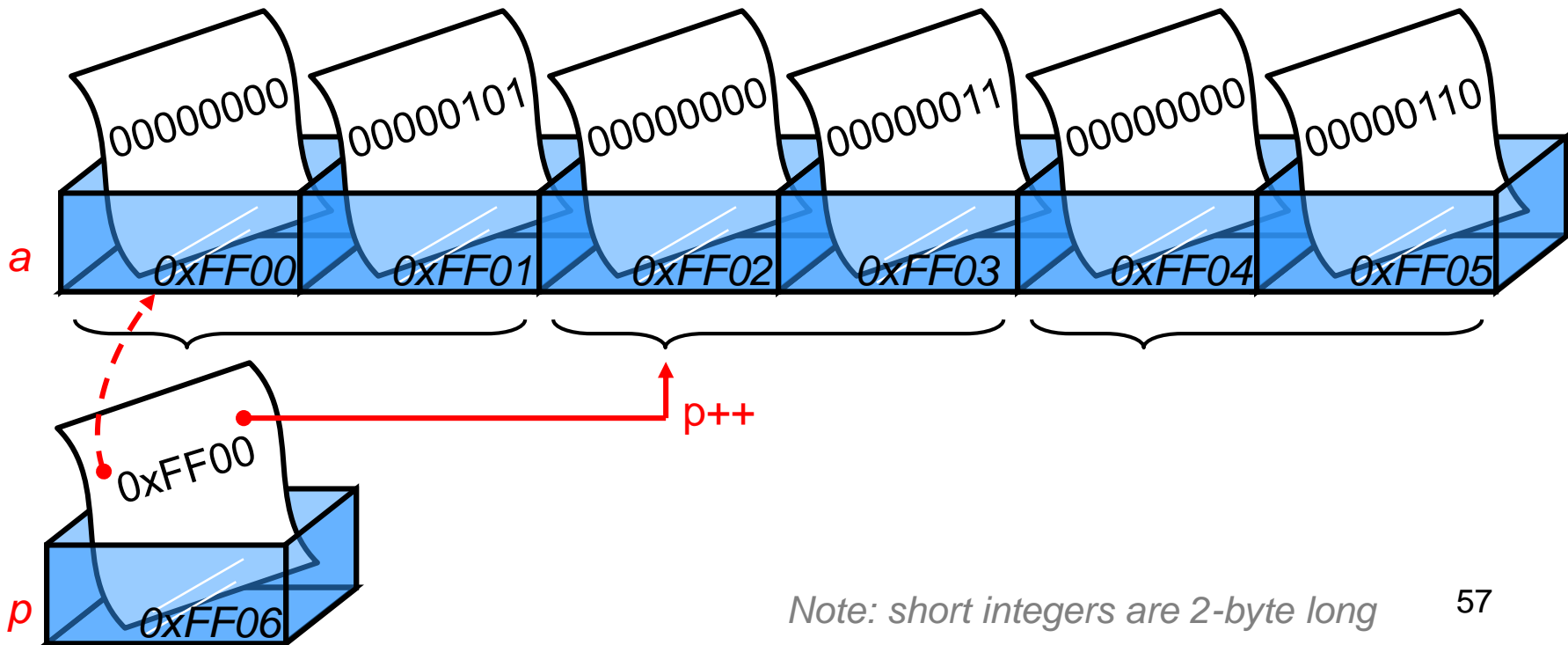
```
char a[] = "Hello";  
char *p;  
p = a;
```





# Arrays and Pointers

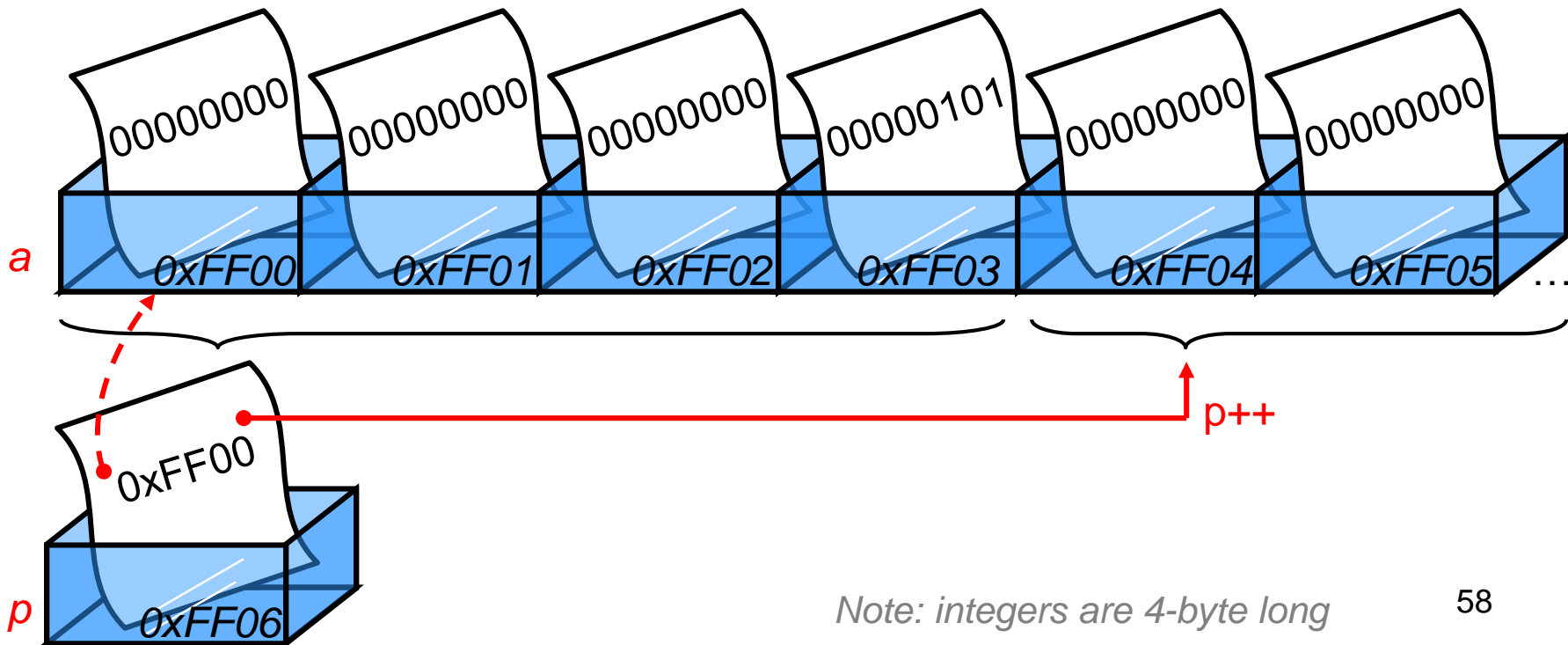
```
short a[] = {5, 3, 6};  
short *p;  
p = a;
```



*Note: short integers are 2-byte long*

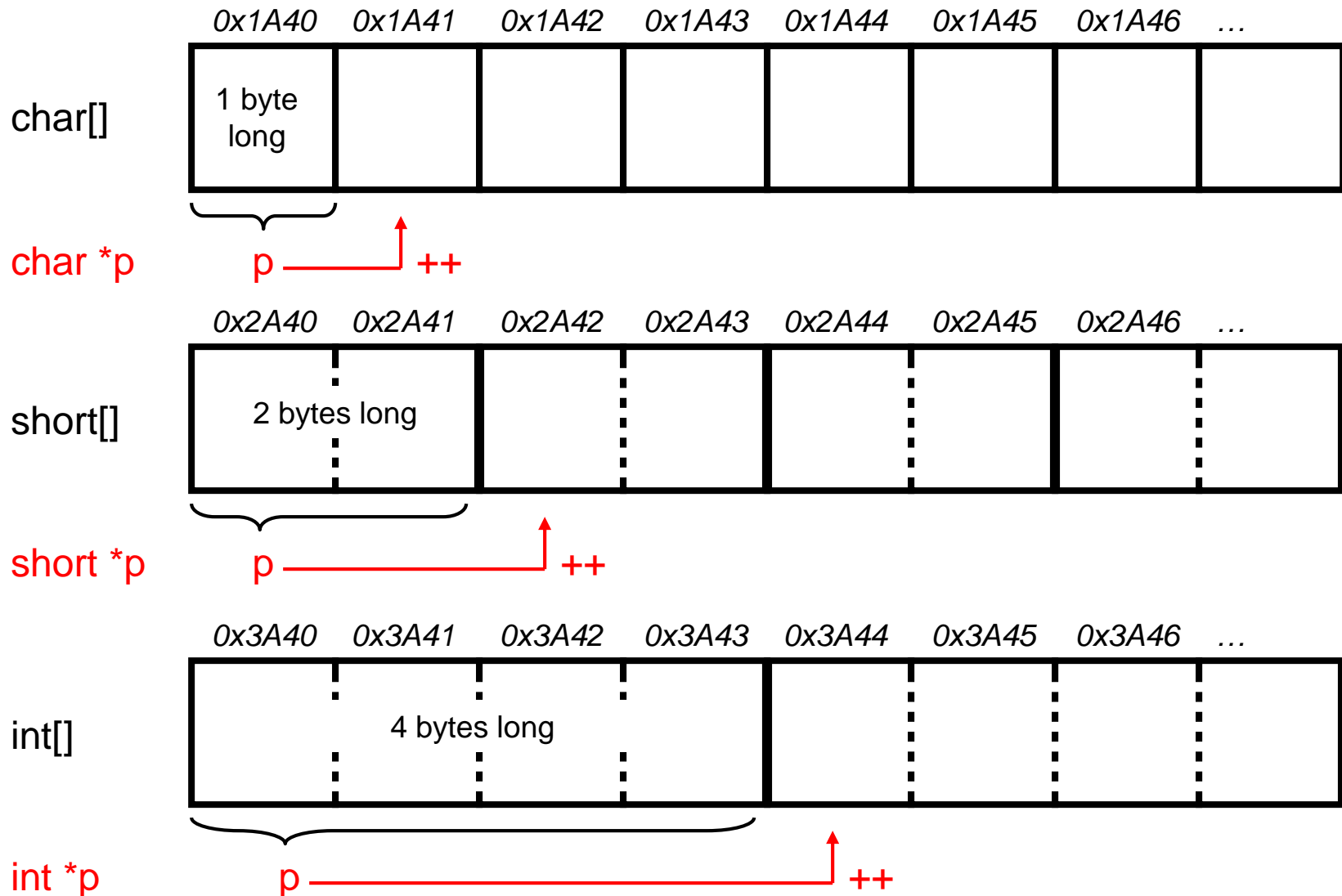
# Arrays and Pointers

```
int a[] = {5, 3, 6};  
int *p;  
p = a;
```



*Note: integers are 4-byte long*

# Arrays and Pointers



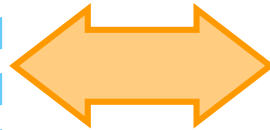
# Composite Structures

# Typedef

- To rename a type to a new name

```
int func(int x) {  
    return x*x;  
}  
int main(...) {  
    int a, b;  
    a = 1;  
    b = func(a);  
    ...  
}
```

equivalent



```
typedef int NUM;  
NUM func(NUM x) {  
    return x*x;  
}  
int main(...) {  
    NUM a, b;  
    a = 1;  
    b = func(a);  
    ...  
}
```

# Structures

- To define a composite structure

```
struct name{  
    data_type1 member1;  
    data_type2 member2;  
    ...  
};
```

- To refer to this structure, use

```
struct name           // C  
name                 // C++
```

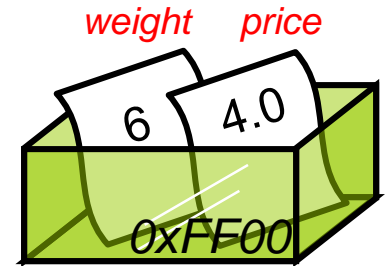
# Structure

```
struct Product{  
    int weight;  
    float price;  
};
```

```
int main(...) {  
    ① Product orange = {6, 4.0};  
    Product apple;  
    apple.weight = 5;  
    ② apple.price = 3.5;  
    printf("%d\n", apple.weight);  
    ...  
}
```

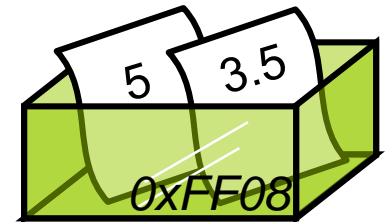
①

*orange*



②

*apple*



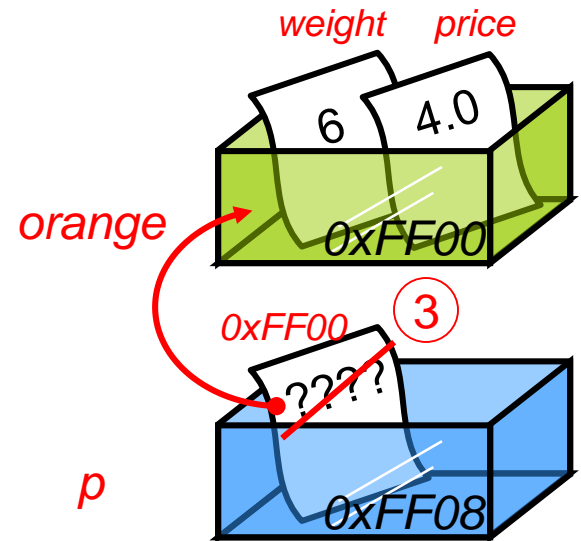
A structure can be initialized by using {}

Or use the . (dot) operator to access the member of a structure

# Pointer to Structure

```
struct Product{  
    int weight;  
    float price;  
};
```

```
int main(...) {  
    ① Product orange = {6, 4.0};  
    ② Product *p;  
    ③ p = &orange;  
    printf("%d\n", p->weight);  
    printf("%d\n", (*p).weight);  
    ...  
}
```



Use the arrow `->` operator to access the member of pointer-to-structure



# Parameter Passing in Functions

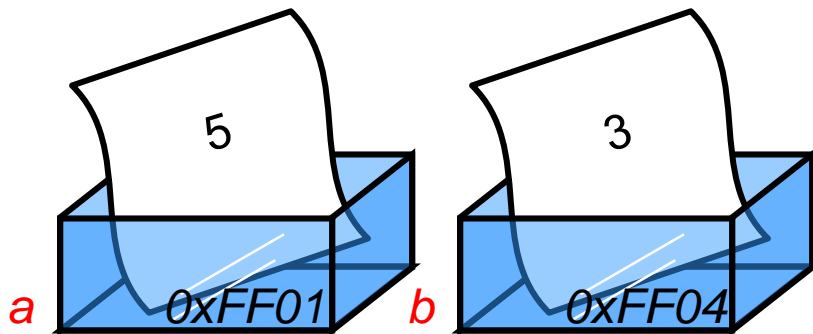
# Parameter Passing in Functions

- Pass by value
  - Involve copying the value of parameters
- Pass by pointer
  - Just pass the **address** of the parameters, without copying the value of them
  - Usually used in passing large-size data structures, e.g. arrays, structures, objects, lists, etc
- Pass by reference
  - C++ reference is a **syntactic sugar** to C pointer
  - Similar to pass-by-pointer but without the hassles of pointers' (&)reference/ (\*)dereference syntax
  - You can specify a formal parameter in the function signature as a **reference parameter**

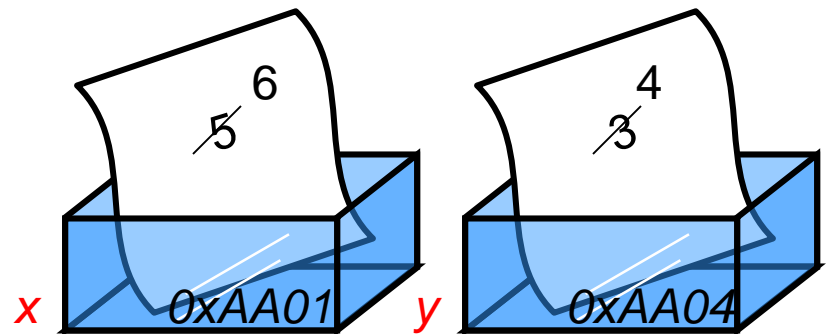
# Pass by Value

```
void plus_one(int x, int y) {  
    x++; y++;  
}
```

```
int a = 5, b = 3;  
plus_one(a, b);
```



The values of *a*, *b* have not been modified



A new set of variables is **duplicate**d in function *plus\_one*

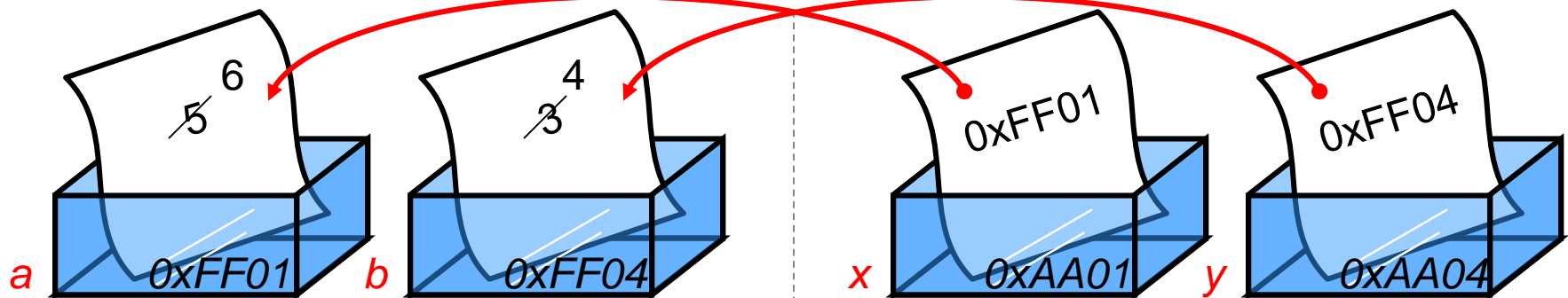
# Pass by Pointer

```
void plus_one(int *x, int *y) {  
    (*x)++; (*y)++;  
}
```

pointers

```
int a = 5, b = 3;  
plus_one(&a, &b);
```

addresses



The values of `a`, `b` have been modified!

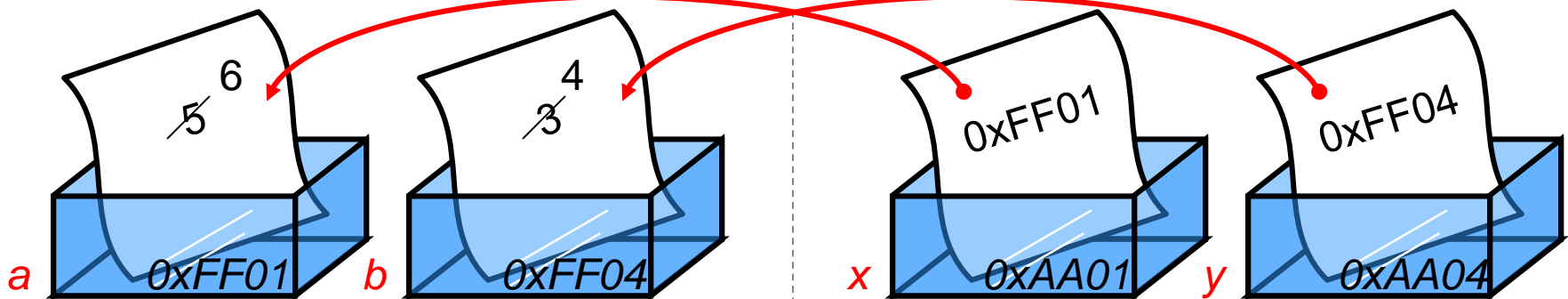
The new set of variables is actually <sup>68</sup>pointing to `a`, `b`

# Pass by Reference

```
void plus_one(int &x, int &y) {  
    x++; y++;  
}
```

reference parameters

```
int a = 5, b = 3;  
plus_one(a, b);
```



The values of `a`, `b` have been modified!

The new set of variables is actually referencing to `a`, `b`

# C++ Reference Example

```
int i = 2;  
//an initial value must be provided in the declaration of r  
int &r = i;      //r is a reference to an integer  
int *p = &i;     //p is a pointer to an integer  
  
printf("%d %d %d %d\n", i, r, p, *p);  
// output: 2 2 001AF9C0 2  
  
r = 4;  
printf("%d %d\n", i, r);  
// output: 4 4
```

# Reference vs. Pointer

1. References cannot be uninitialized. Because it is impossible to reinitialize a reference, they **must be initialized as soon as they are created**.
2. A pointer can be re-assigned any number of times while a reference **cannot be re-seated** after binding.
3. Pointers can point nowhere (NULL), whereas reference always refers to an object (because of #1).
4. You cannot take the address of a reference like what you can do with pointers. Any occurrence of its name refers directly to the object it references.
5. There is no **reference arithmetic** but you can take the address of an object pointed by a reference and do **pointer arithmetic** on it (because of #4).

# **Standard Input / Output**



# cin & cout

- Default input/output stream objects
- A stream is a sequence of bytes (characters) that can be read from or written to
  - cin is a stream on the keyboard input
  - cout is a stream on the screen output
- The extractor (>>) / insertor (<<) is used to read/write from/to the input/output stream

# Standard Output

```
#include <cstdio>
#include <iostream>
using namespace std;

...

int x = 1;
float y = 2.5;
char z = 'a';
char w[80] = "xxxxxx";

printf("%d %f %c %s\n", x, y, z, w);
std::cout << x;
cout << endl;
cout << y << " " << z << " " << w;
```

How to output the values to standard output (screen)?

Use `printf()` in `<cstdio>`:

- integer: `%d`
- float: `%f`
- character: `%c`
- string: `%s`

Use `cout` in `<iostream>`:

- `cout` is defined in the `std` namespace
- Use insertion operator to insert values to output stream.
- Multiple insertions can be chained.
- Use `endl` to set a new line.

# Standard Input

```
#include <cstdio>
#include <iostream>
using namespace std;

...

int x;
float y;
char z;
char w[80];

cin >> x;
scanf("%f", &y);
cin >> z;
scanf("%s", w);
```

How to read the values from standard input (console)?

Use `scanf()` in `<cstdio>`:

- integer: `%d`  
float: `%f`  
character: `%c`  
string: `%s`

Use `cin` in `<iostream>`:

- `cin` is defined in the `std` namespace
- Use extraction operator to extract values from input stream.

# scanf()

- *scanf* can only read a “word”, but not a sentence. It stops reading if meets **whitespace** characters.
- What are whitespace characters?
  - Blank space: ‘ ’
  - Newline: ‘\r’ ‘\n’
  - Tab: ‘\t’
- Visual Studio compiler will tell you the function *scanf* is not safe.
  - Add this code to the beginning of your program to suppress this MS secure warning


```
#ifdef _MSC_VER  
#define _CRT_SECURE_NO_WARNINGS  
#endif
```

# scanf() Examples

scanf() will stop reading when it meets enter, space or tab (whitespace)

```
scanf("%s", w);  
printf("##%s##\n", w);
```

```
abc<enter>  
##abc##
```




The newline character has been ignored by scanf()

```
scanf("%s", w);  
printf("##%s##\n", w);
```

Space

```
abc def<enter>  
##abc##
```



The space and following characters have been ignored by scanf()

# More on Input

- When looking for the input value in the stream, the >> operator **skips any leading whitespace characters** and stops reading at the first character that is inappropriate for the data type (whitespace or otherwise).
- If an inappropriate character is read, the cin stream enters a fail state and the rest of the statements in our program that read from cin are ignored.
- You can use the **get()** function to input the very next character in the input stream **without skipping any whitespace characters**:

```
char someChar;  
cin.get(someChar);
```

- The **ignore()** function is used to skip characters in the input stream:

```
cin.ignore(200, '\n');
```

- The first parameter is an int expression; the second, a char value. This **skips the next 200 characters or until a newline character is read**, whichever comes first

# Output Manipulators

- Manipulators change the output format of your data. To use them, you will need to include this header in your C++ source code.

```
#include <iomanip>
```

- **setw()** sets the width of the field to be printed to the screen

```
■ cout << 5 << setw(4) << 6 << 7;      // output:5    67
```

- **setprecision()** sets the *decimal precision* to be used to format floating-point values:

```
■ cout << setprecision(5) << 3.14159;      // 3.1416
```

```
■ cout << setprecision(1) << 3.14159;      // 3
```

- To specify the number of digits after the decimal point:

```
■ cout << setiosflags(ios::fixed);
```

```
■ cout << setprecision(2) << 12.1234;      // 12.12
```

- To put it back to its original specification:

```
■ cout << setiosflags(ios::scientific);
```

# File Input/Output

- In a similar way C++ provides **streams** which can manipulate **files**
- C++ provides **2 file streams**
  - ifstream** input file stream
  - ofstream** output file stream

Must `#include <fstream>` to use them

- Example:

```
#include <fstream>

int number;
ifstream in("in.dat");
ofstream out("out.dat");
in >> number;
out << number;
```



# Input File Streams (ifstream)

- Allows data to be read from a file
- An input file stream can be defined as follows:

```
ifstream stream_var(filename);
```

Example:

```
ifstream inFile("test.dat");
```

- If stream opened successfully, inFile evaluates to **positive** and the stream becomes attached to the file test.data
- If stream open failed (e.g. file does not exist) inFile evaluates to **zero**
- **Important:** Effects of reading data from file which has failed to open is undefined

# Input File Streams (ifstream)

- When file opened successfully, data can be read using normal **extractor** functions

```
int n;  
char c;  
ifstream inFile("test.dat");  
inFile >> n;  
inFile.get(c);  
inFile.ignore(100, 'A');  
inFile.close();
```

- **Note:** When a file stream goes out of scope it will automatically close the file it is attached to

# File Input Failure/End

- To check if the file has been **opened or not**, you can use:

```
if (inFile) // testing if the file opened successfully
{ ... }
```

- To test for **end of file**, you can use:

```
while (!inFile.eof())
{ ... }
```

For instance:

```
int number;
inFile >> number; // reading number from a file
while (!inFile.eof())
{
    cout << number; // print number on screen
    inFile >> number;
}
```

# Output File Stream (ofstream)

- Allows data to be written to a file

An output file stream can be defined as follows:

```
ofstream stream_var(filename);
```

Example:

```
ofstream outFile("temp.data");
```

- If stream opened successfully, outFile evaluates to **positive** and the stream becomes attached to the file temp.data
- If stream open failed (e.g. no disk space) outFile evaluates to **zero**

## ■ Note:

- If the file already exists its contents will be deleted
- If the file does not exist, a file with the same name is created
- Data can be **appended** to a file by using constructor with two arguments

```
ofstream outFile("temp.data", ios::app);
```

# Example on How to Write to a File

```
#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

int main ()
{
    float first, second, sum;           // Declaring variables
    ofstream outFile("out.dat");        // Opening file for output

    cout << "Enter two numbers" << endl;
    cin >> first >> second;             // Reading in the two numbers
    sum = first + second;
    outFile << setiosflags(ios::fixed); // Formatting the output
    outFile << setprecision(2);
    outFile << sum << endl;             // Writing into the file

    return 0;
}
```

# Pseudo Code

- We need a language to express program development
  - English is **too verbose** and imprecise.
  - The target language, e.g. C/C++, requires too much details.
- Pseudo code resembles the target language in that
  - it is a sequence of steps (each step is precise and unambiguous)
  - it has similar control structure of C/C++
- Pseudo code is a kind of **structured English** for describing algorithms. It allows the designer to focus on the logic of the algorithm **without being distracted by details of language syntax**.

```
x = max{a, b, c}
```

Pseudo code

```
x = a;  
if (b > x) x = b;  
if (c > x) x = c;
```

C++ code

# Pseudo Code Example

- An  $m \times n$  matrix is said to have a saddle point if some entry  $A[i][j]$  is the smallest value on row  $i$  and the largest value in column  $j$ .

An  $6 \times 8$  matrix with a saddle point

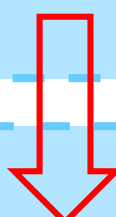
|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 11 | 33 | 55 | 16 | 77 | 99 | 10 | 40 |
| 29 | 87 | 65 | 20 | 45 | 60 | 90 | 76 |
| 50 | 53 | 78 | 44 | 60 | 88 | 77 | 81 |
| 46 | 72 | 71 | 23 | 88 | 26 | 15 | 21 |
| 65 | 83 | 23 | 36 | 49 | 57 | 32 | 14 |
| 70 | 22 | 34 | 19 | 54 | 37 | 26 | 93 |

- Problem:
- Given an  $m \times n$  matrix, determine if there exists one or more saddle points.

# Pseudo Code Solutions

```
// high-level pseudo code solution
for each row {
    j = index of the smallest element on row i;
    if (A[i][j]) is the largest element in column j)
        A[i][j] is a saddle point;
}
```

```
// refined pseudo code
for (i = 0; i < m; i++) {
    j = index of the smallest element on row i;
    for (k = 0; k < m; k++)
        if there does not exist A[k][j] > A[i][j]
            A[i][j] is a saddle point;
}
```





# Suggestions for Good Style

- Use informative and meaningful variable names
- Insert useful comments (i.e. assertions) in the source program
- Format the source file with **proper indentation** of statements and align the braces so that the control structures can be read easily
- Do not use **goto** statement, especially backward jump
- Use **single-entry single-exit** control blocks, or at most one break statement inside a loop
- Avoid ambiguous statements                      e.g. `x[i] = i++;`
- **Minimize direct accesses to global variables**, especially you should avoid modifying the values of global variables in a function
- Always make a **planning** of the program organization and data structures before start writing program codes
- Should avoid using the **trial-and-error** approach without proper understanding of the problem to be solved
- Avoid **side effects** (see example in next page)

# Side Effect

- In computer science, a function or expression is said to have a side effect if, in addition to returning a value, it also modifies some state or has an observable **interaction with calling functions or the outside world**.

```
int x = 0; //global variable
```

```
int f(int n) {  
    x += 1; //side effect: modify the value of x which is not a formal parameter of function f()  
    return n + x;  
}
```

```
int g(int n) {  
    x *= 2; //side effect  
    return n * x;  
}
```

```
void main() {  
    int t;  
    t = f(1) + g(2); // logically the same as t = g(2) + f(1) but the results will be totally different.  
}
```