Pointer

Inst. Nguyễn Minh Huy

Contents

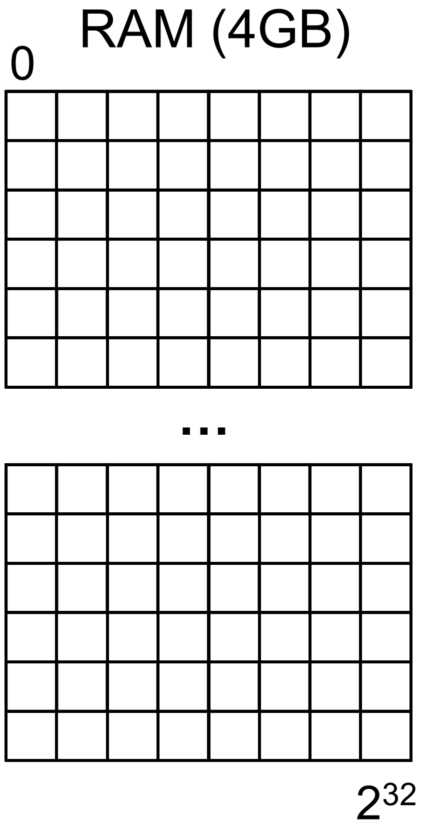
- Pointer concepts.
- Pointer usage.
- Pointer vs. Array.

Contents

- **Pointer concepts.**
- Pointer usage.
- Pointer vs. Array.

Pointer concepts

- Computer memory:
  - RAM (**R**andom **A**ccess **M**emory).
    - Primary vs. Secondary memory.
  - Used to store:
    - Operating system.
    - Programs: variables + functions.
  - Contains 1-byte cells.
    - RAM 4GB ~ 4 billion cells.
  - Each cell has an address number.
    - RAM 4GB address 0 → $2^{32} - 1$.



Pointer concepts

- Variable address:
  - How it works, when declaring a variable?
    - ➢ Allocate a series of memory cells.
    - ➢ Assign variable name to the first cell.
    - ➢ Number of cells? → variable type.

```
int  x;
```

	65	66	67	68
x	?	?	?	?

➔ Variable address = address of first cell.

  - How value is stored in variable?
    - ➢ Divide value into bytes.
    - ➢ Store each byte in cell.
    - ➢ Start from the first cell.

```
x = 1057;
```

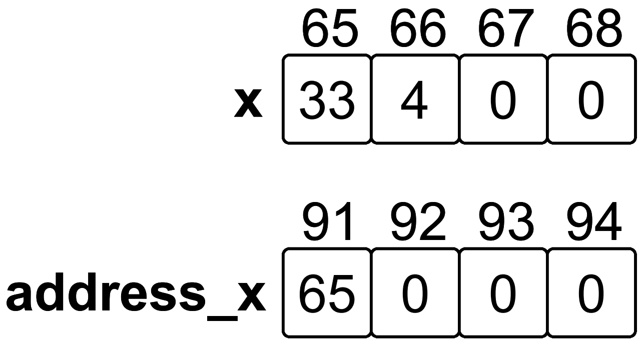
	65	66	67	68
x	33	4	0	0

Pointer concepts

- Address type in C:
  - Store integer, real number? → int, float type.
  - Store variable address? → address type.
  - Syntax: **<type> ***.
    - Address of int: int *.
  - Operator **&**:
    - Usage: get variable address.
    - Syntax: **&**<variable name>;

```
int    x = 1057;
float  y = 1.25;
int    *address_x = &x;
float  *address_y = &y;
```

x = 1057;



	65	66	67	68
x	33	4	0	0

	91	92	93	94
address_x	65	0	0	0

Pointer concepts

- Pointer in C:
  - A variable has address type.
  - Store address of other variable.
  - Its value is an address number.
  - Its size:
    - Fix-sized for all address type.
    - Depend on platform:
      - Intel 8008 (1972), 8-bit, 1 byte (256 B).
      - Intel 8086 (1978), 16-bit, 2 bytes (64 KB).
      - Intel 80386 (1985), 32-bit, 4 bytes (4 GB).
      - Intel Core (2000), 64-bit, 8 bytes (16 TB).

Contents

- Pointer concepts.
- **Pointer usage.**
- Pointer vs. Array.

Pointer usage

- Pointer declaration:
  - Declare variable has address type.
  - Method 1:

```
<type> *<pointer name>;

int    *p1;          // Pointer storing address of int.
float  *p2;          // Pointer storing address of float.
```

  - Method 2:

```
typedef <type> * <alias>;

<alias> <pointer name>;

typedef  int    * int_pointer;
typedef  float * float_pointer;
int_pointer      p1;
float_pointer  p2
```

Pointer usage

- Pointer referencing:
  - Pointer has random address at first → initialization.
  - **Operator &**: get variable address.
    - Syntax: <pointer name> = **&**<variable>;

      ```
      int  x = 5;
      int  *p = &x;
      ```

  - Pointer only accepts address of the same type!!

    ```
    float  y;
    int    *q = &y;        // Wrong!!
    ```

  - NULL address:
    - Empty address → default initialization.

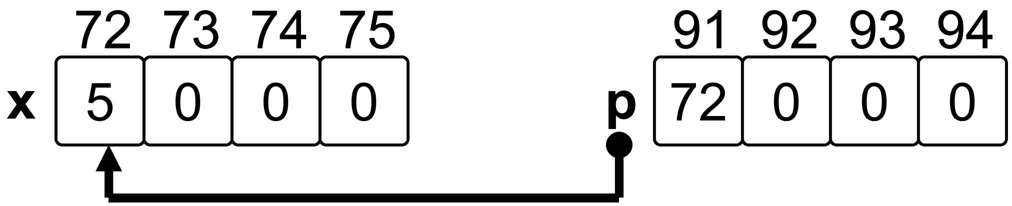
      ```
      int  *r = NULL;      // empty address.
      ```

Pointer usage

- Pointer de-referencing:
  - **Operator ***:
    - Read variable whose address pointer stores.
    - Syntax: <variable> = *<pointer>;

```
int  x = 5;
int  *p = &x;
int  k = *p;              // get x value.
printf("%d\n", p);    // print x address.
printf("%d\n", *p);  // print x value.
printf("%d\n", &p);// print p address.
```

➔ Pointer **points to** variable whose address it stores!



	72	73	74	75
x	5	0	0	0

	91	92	93	94
p	72	0	0	0

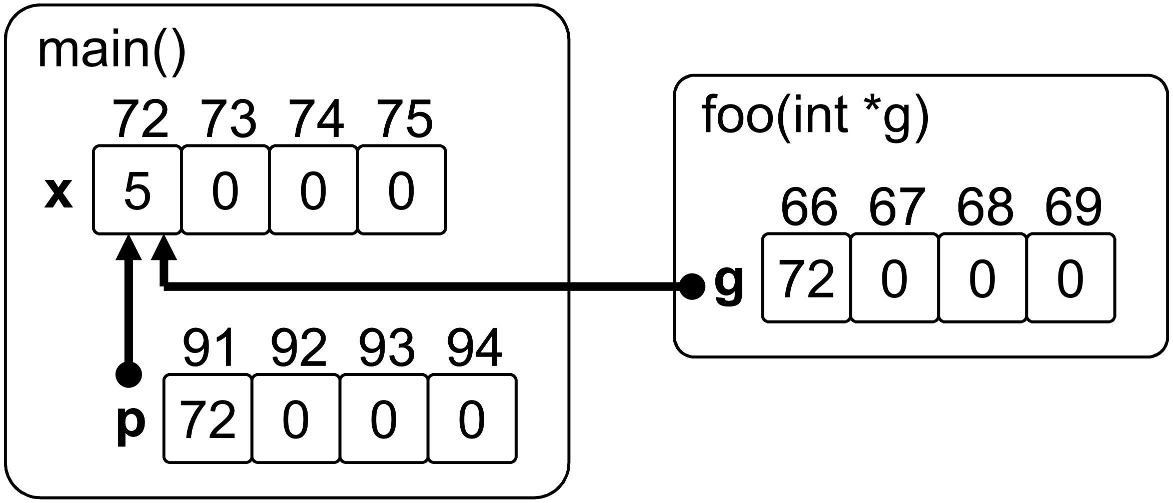
Pointer usage

- Passing pointer to function:
  - Pass-by-value:
    - Pass copy of pointer to function.
    - Address stored in pointer is NOT CHANGED.
    - Variable that pointer points to CAN BE CHANGED.

```
void foo(int *g)
{
        *g = *g + 1;
        g = g + 1
}

int main()
{
        int  x = 5;
        int  *p = &x;

        foo(p);
        // x is changed.
}
```



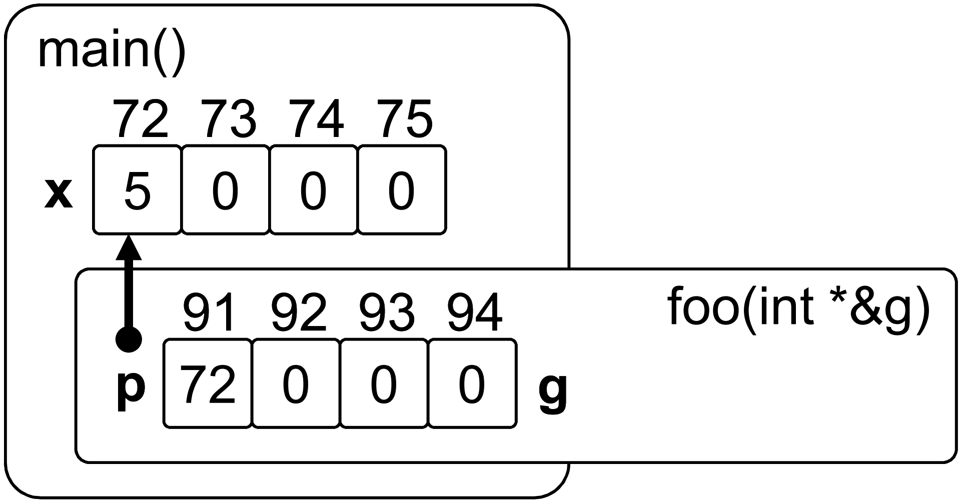
Pointer usage

- Passing pointer to function:
  - Pass-by-reference:
    - Pass real pointer to function.
    - Address stored in pointer CAN BE CHANGED.
    - Variable that pointer points to CAN BE CHANGED.

```
void foo(int *&g)
{
        *g = *g + 1;
        g = g + 1
}

int main()
{
        int  x = 5;
        int  *p = &x;

        foo(p);
        // x is changed.
        // p is changed.
}
```



Pointer usage

- Pointer to struct:
  - Pointer stores address of struct variable.
  - Declaration:
    - Method 1: <struct type> *<pointer name>;
    - Method 2: **typedef** <struct type> * <alias>;
      <alias> <pointer name>;

```
struct Fraction
{
   int numerator, denominator;
};
 typedef Fraction * FractionPointer;

Fraction          *p;
FractionPointer   q;
```

Pointer usage

- Pointer to struct:
  - Access struct member through pointer:
    - Method 1: **(***<pointer name>**).**<struct member>;
    - Method 2: <pointer name>**->**<struct member>;

```
Fraction  f;
Fraction  *p = &f;

(*p).numerator = 1;
p->denominator = 2;
```

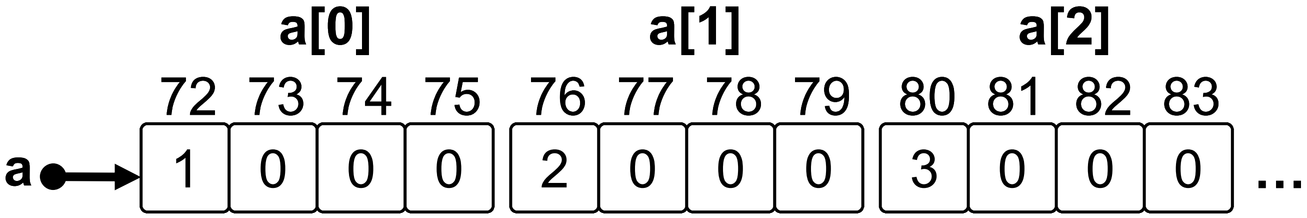
Contents

- Pointer concepts.
- Pointer usage.
- **Pointer vs. Array.**

Pointer vs. Array

- Array in C:
  - Is a pointer.
  - Stores address of first element.

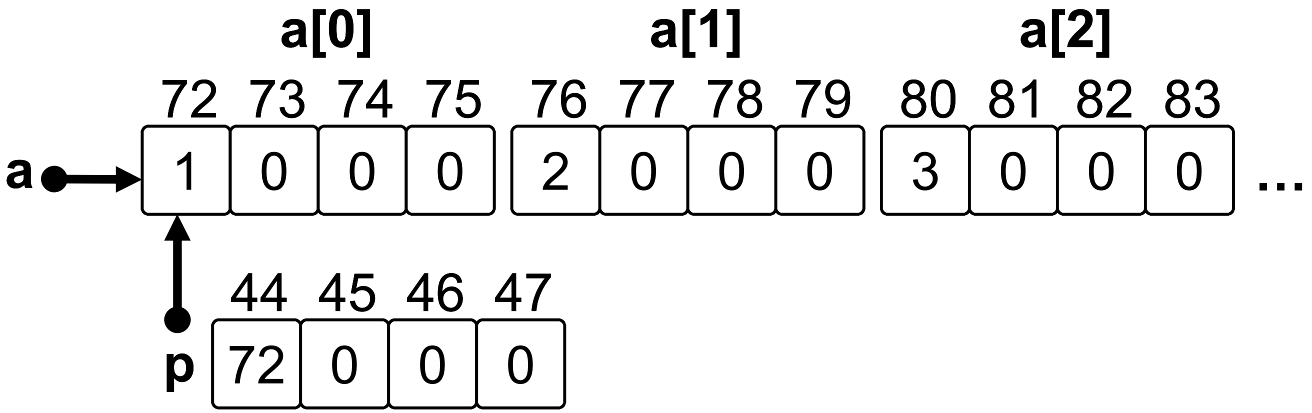
```
int main()
{
        int  a[ 10 ];
        printf(“%d\n”, a);
        printf(“%d\n”, &a[0]);      // a == &a[0].
}
```



Pointer vs. Array

- Pointer to array element:
  - Access array indirectly.
  - Consider the following code:

```
int  a[100] = { 1, 2, 3 };
int  *p = a;              // p = &a[0]
*p = *p + 1;
printf(“%d\n”, *p);
```



a[0] a[1] a[2]

72 73 74 75 76 77 78 79 80 81 82 83

a → 1 0 0 0 2 0 0 0 3 0 0 0 …

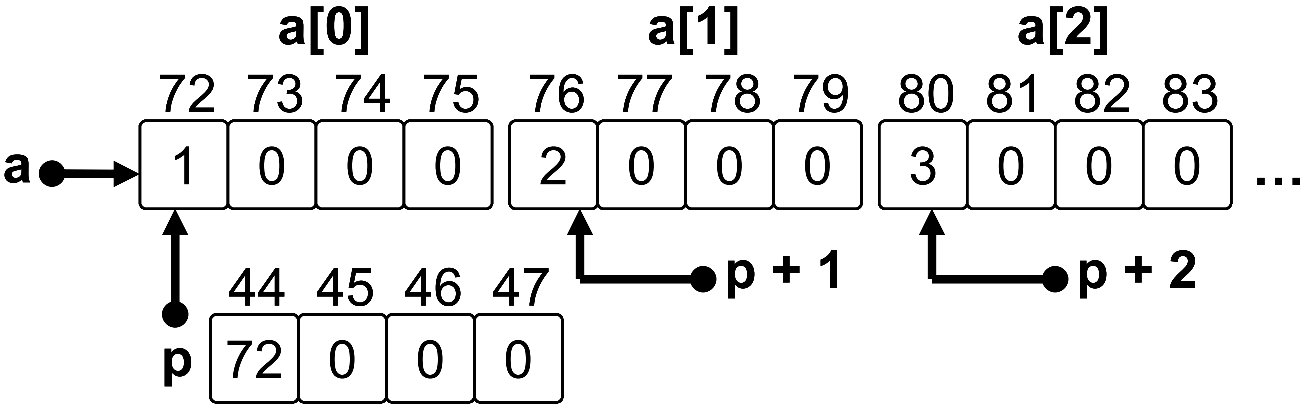
44 45 46 47

p 72 0 0 0

Pointer vs. Array

- Pointer increment/decrement:
  - Pointer value changed based on pointer type.
  - Formula:
    - <Pointer> +/- k = <Address> +/- k * sizeof(<Pointer Type>).

```
int  a[100] = { 1, 2, 3 };
int  *p = a;
printf("%d\n", *(p + 1) );
printf("%d\n", *(p + 2) );
```



Pointer vs. Array

- Operator [ ]:
  - Read memory content pointer points to.
  - Usage:

    <Pointer>[<Index>] ~ * (<Pointer> + <Index>)

```
int  a[100] = { 1, 2, 3 };
int  *p = a;

a[2] = 5;
*(a + 2) = 5;
*(p + 2) = 5;
p[2] = 5;
```

Pointer vs. Array

- Passing array to function:
  - Not passing whole array.
  - Only passing address of first element.

  ➔ Pass pointer points to first element.

```
void printArray(int a[ ], int n) {
        for (int i = 0; i < n; i++)
               printf("%d ", *(a++) );            // Same as a[ i ]…
}
int main() {
        int  a[100];
        printArray(a, 100);
        for (int i = 0; i < n; i++)
               printf("%d ", *(a++) );            // Wrong, why?
}
```

Summary

- Pointer concepts:
  - Variable store address of other variable.
- Pointer usage:
  - Declaration: <Data type> *.
  - Initialization: operator & get variable address.
  - Opertator *: access memory content pointer points to.
- Pointer vs. Array:
  - Array in C is a pointer.
  - Pointer can points to array.
  - Operator [ ]: indirect access array elements.

Practice

- Practice 2.1:

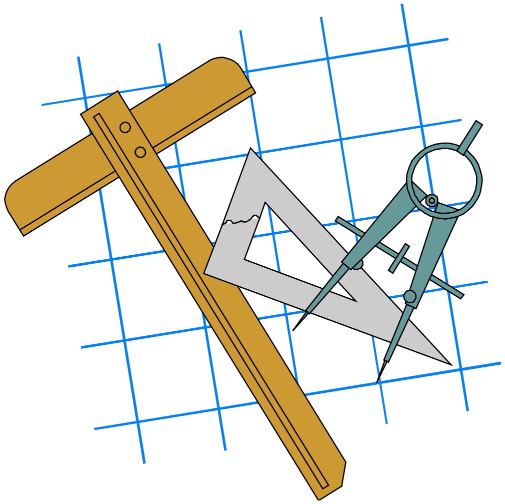
Given the following code:

```
int main()
{
        int     *x, y = 2;
        float  *z = &y;

        *x = *z + y;
        printf(“%d”, y);
}
```

a) Fix error of the code.

b) After fixing, what is displayed on screen?



Practice

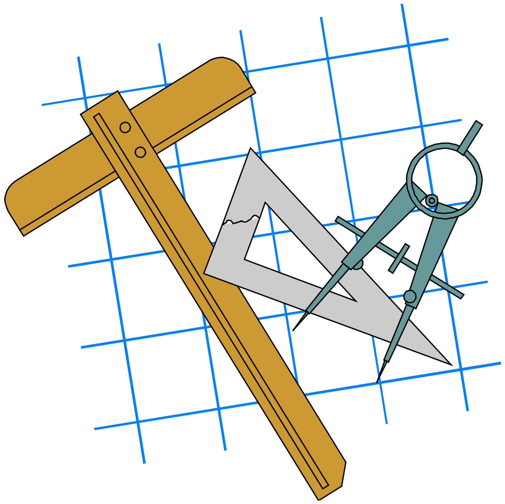
- ## Practice 2.2:

Explain the difference amongst the following 3 functions:

```
void swap1(int x, int y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

```
void swap2(int &x, int &y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

```
void swap3(int *x, int *y)
{
        int temp = *x;
        *x = *y;
        *y = temp;
}
```



Practice

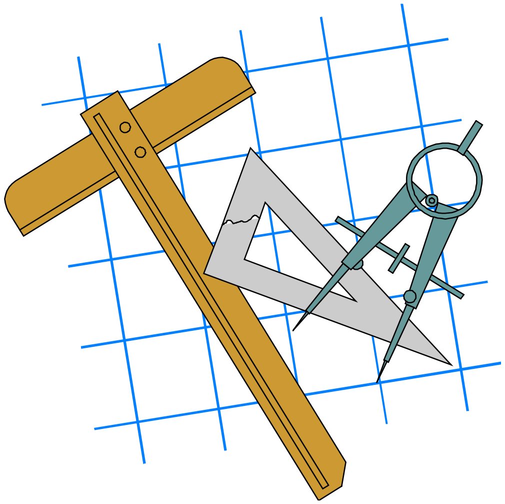
- Practice 2.3:

  Given the following program:

```
int main()
{
        double   m[100];
        double   *p1, *p2;

        p1 = m;
        p2 = &m[6];
}
```

  How many bytes from p1 to p2?



Practice

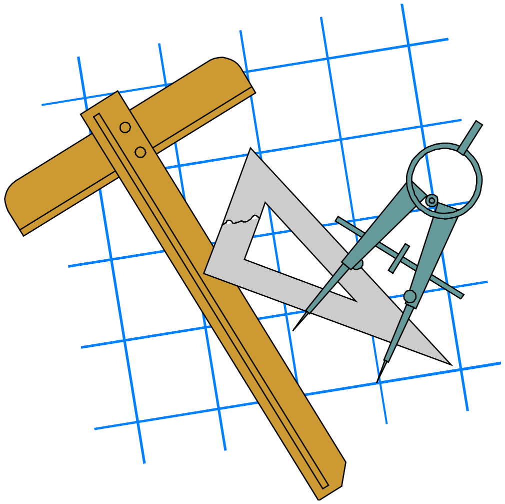
- ## Practice 2.4:

What is displayed on screen of the following code:

```
#include <stdio.h>

int main()
{
        int     x = 1023;
        char  *p = (char *)&x;

        printf(“%d %d %d %d\n”, p[0], p[1], p[2], p[3]);
}
```



Practice

- ## Practice 2.5:

  Using pointer to write a program that can do the followings:

  a) Read from keyboard an array of N fractions.

  b) Extract negative fractions to another array.

  c) Write the result to screen.