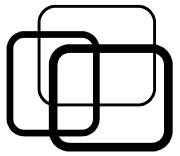
Pointer

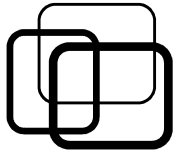
Inst. Nguyễn Minh Huy

Contents



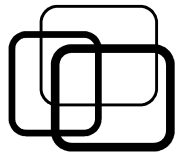
- Pointer concept.
- Pointer usage.
- Pointer vs. array.
- Memory management.

Contents

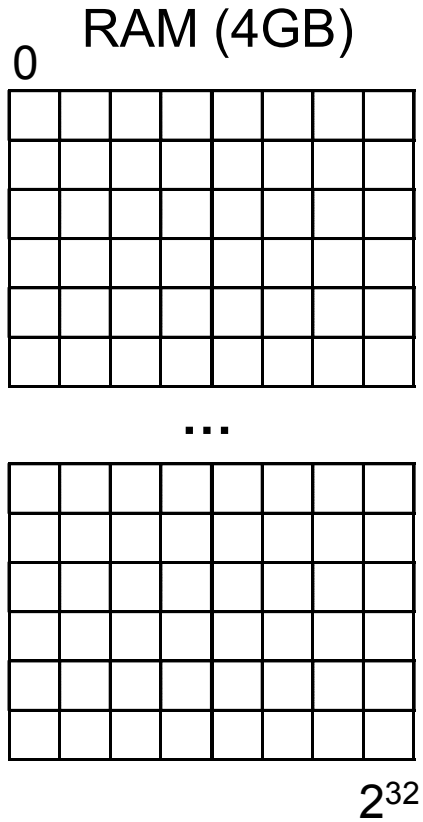


- **Pointer concept.**
- Pointer usage.
- Pointer vs. array.
- Memory management.

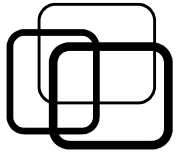
Pointer concept



- 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 concept



- 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.

  → 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.

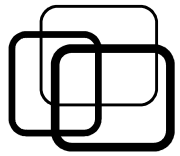
int **x**;

	65	66	67	68
x	?	?	?	?

x = 1057;

	65	66	67	68
x	33	4	0	0

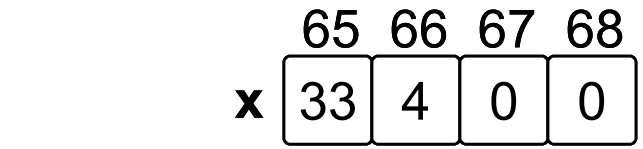
Pointer concept

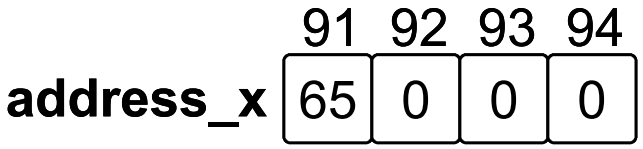


- Address type in C:
  - Store integer, real number? → int, float type.
  - Store variable address? → address type.
  - Syntax: **<type> ***.
    - Address of int: int *.
  - Operator **&**:
    - Usage: get variable address.
    - Syntax: **&**<variable name>;

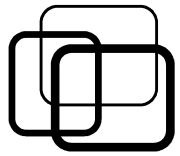
```
int     x = 1057;
float  y = 1.25;
int     *address_x = &x;
float  *address_y = &y;
```

x = 1057;



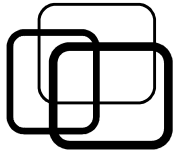


Pointer concept



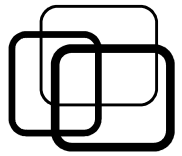
- Pointer in C:
  - A variable having address type.
  - Store address of other variable.
  - Its value is an address number.
  - Its size:
    - Fix-sized for all address type.
    - Depend on platform:
      - Intel 8008 (1972), 8-bit, 1 byte (256 B).
      - Intel 8086 (1978), 16-bit, 2 bytes (64 KB).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

Contents



- Pointer concept.
- **Pointer usage.**
- Pointer vs. array.
- Memory management.

Pointer usage



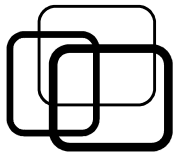
- Pointer declaration:
  - Declare variable having address type.
  - Method 1:

```
<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;
```

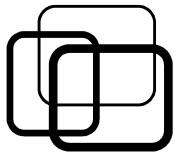
  - Pointer only accepts address of the same type!!

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

  - NULL address:
    - Empty address → default initialization.

```
int  *r = NULL;          // C, use <stdio.h>
int  *s = nullptr;       // C++, keyword
```

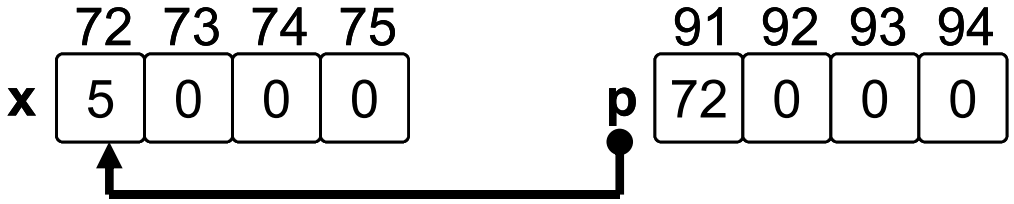
Pointer usage



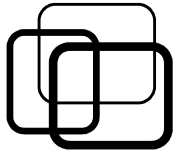
- Pointer de-referencing:
  - **Operator *:**
    - Read variable whose address pointer stores.
    - Syntax: <variable> = *<pointer>;

```
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.
```

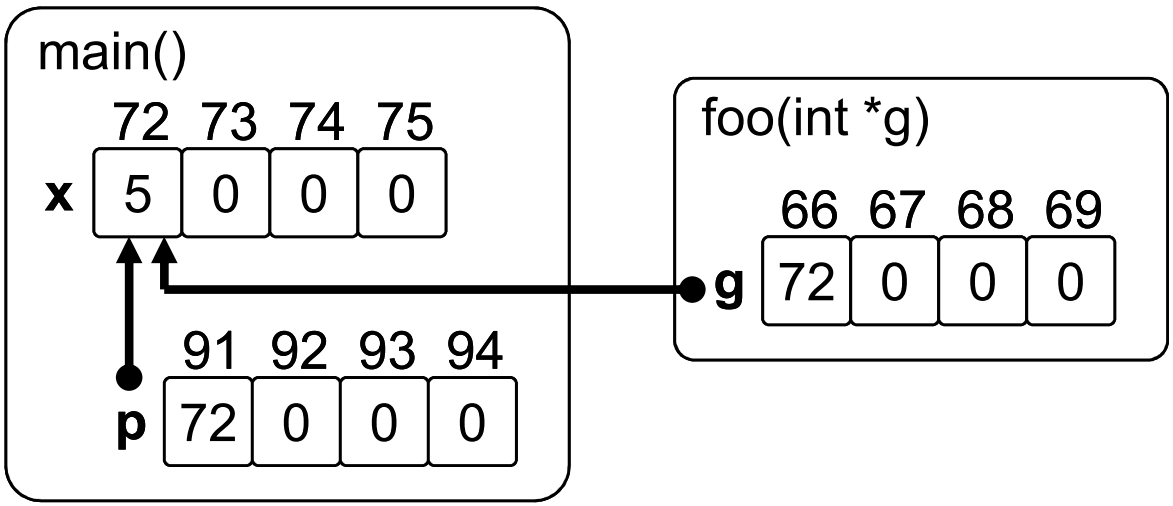
  ➔ Pointer **points to** variable whose address it stores!



Pointer usage

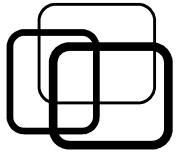


- Passing pointer to function:
  - Pass-by-value:
    - Pass copy of pointer to function.
    - Address stored in pointer is NOT CHANGED.
    - Variable that pointer points to CAN BE CHANGED.

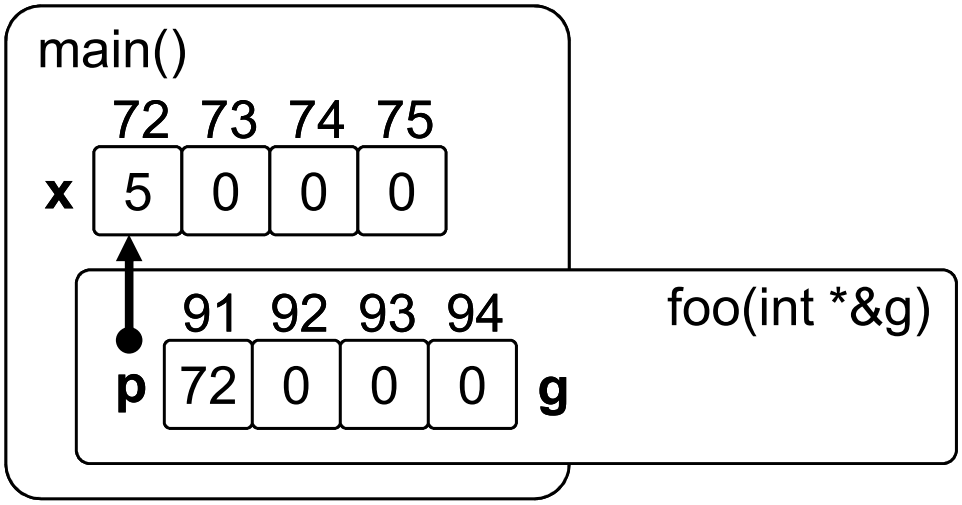



```
void foo(int *g)
{
        *g = *g + 1;
        g = g + 1
}

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

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

Pointer usage

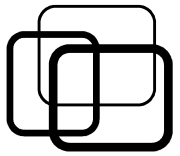


- Passing pointer to function:
  - Pass-by-reference (C++):
    - Pass real pointer to function.
    - Address stored in pointer CAN BE CHANGED.
    - Variable that pointer points to CAN BE CHANGED.




```
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.
}
```

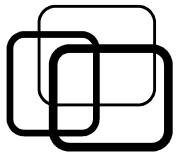
Pointer usage



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

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

struct Fraction       *p;
FractionPointer       q;
```

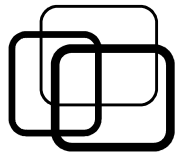
Pointer usage



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

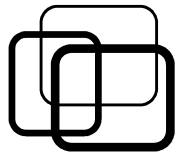
```
struct Fraction     f;
struct Fraction     *p = &f;

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

Contents



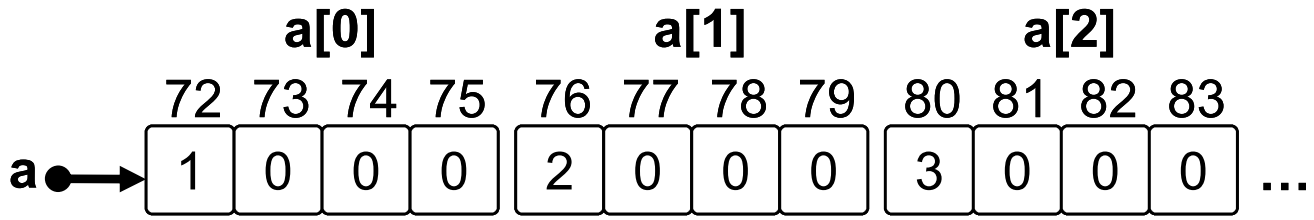
- Pointer concept.
- Pointer usage.
- **Pointer vs. array.**
- Memory management.

Pointer vs. array

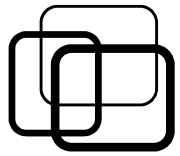


- Pointer as array:
  - Array is a pointer.
  - Stores address of first element.

```
int main()
{
        int  a[ 10 ] = { 1, 2, 3 };
        printf("%d\n", a);
        printf("%d\n", &a[0]);      // a = &a[0].
}
```

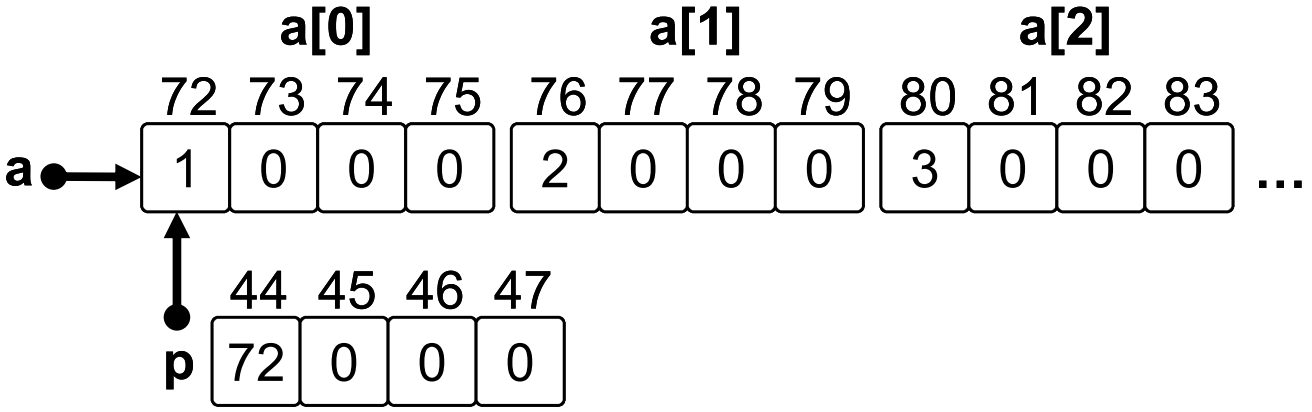


Pointer vs. array

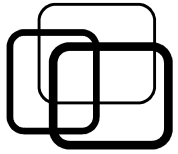


- Pointer to array element:
  - To access array indirectly.
  - Consider the following code:

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

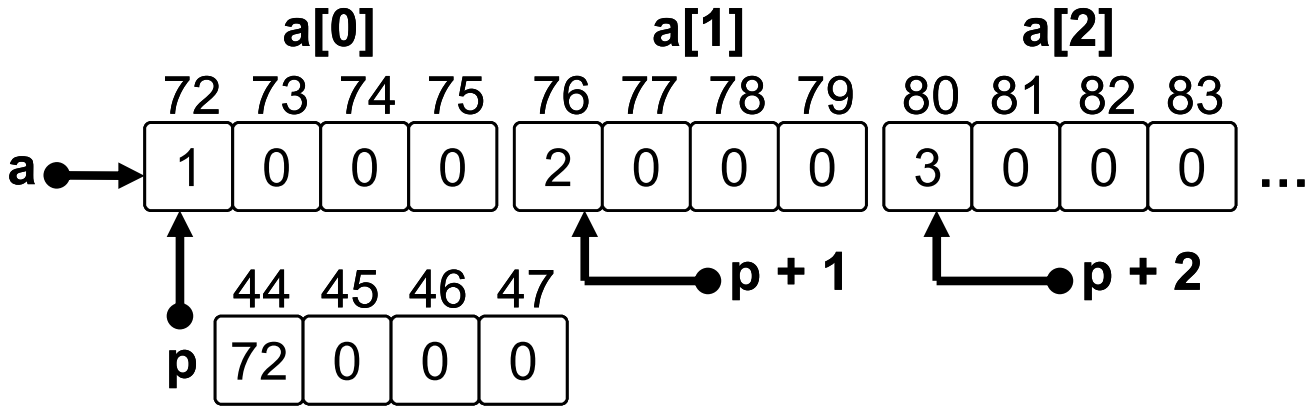


Pointer vs. array

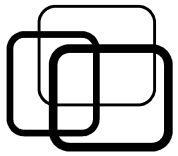


- Pointer increment/decrement:
  - Pointer value changed based on pointer type.
  - Jumping formula:
    - <Pointer> **+/- k** = <Address> **+/- k * sizeof(<Pointer Type>)**.

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



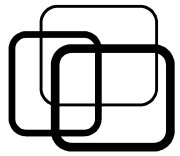
Pointer vs. array



- Operator [ ]:
  - Read memory content pointer jumps to.
  - Syntax: **<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;
```

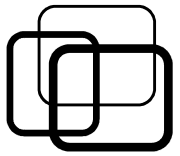
Pointer vs. array



- Passing array to function:
  - Not passing whole array.
  - Only passing address of first element.
  - ➔ Pass pointer points to first element.

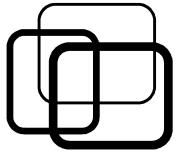
```
void printArray(int a[ ], int n) {
        for (int i = 0; i < n; i++)
                printf("%d ", *(a++) );          // Same as a[ i ]…
}
int main() {
        int  a[100];
        printArray(a, 100);
        for (int i = 0; i < n; i++)
                printf("%d ", *(a++) );          // Wrong,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

Contents



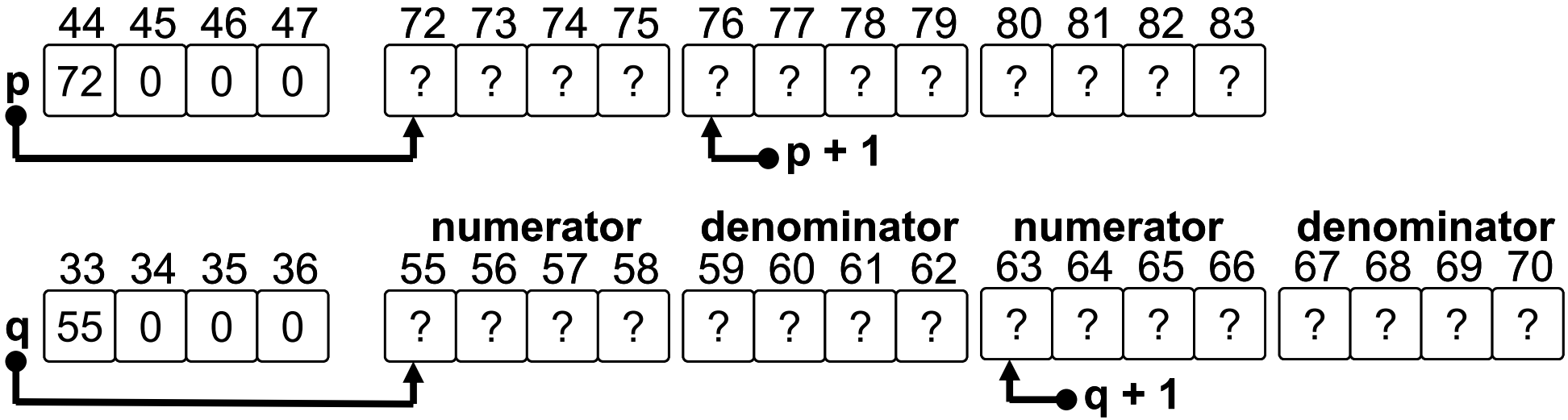
- Pointer concept.
- Pointer usage.
- Pointer vs. array.
- **Memory management.**

Memory management

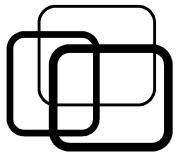


- Memory allocation (library <stdlib.h>):
  - Request memory from RAM.
  - **malloc(** <number of bytes> **)**.
    - Return: allocated memory address or NULL (failed).

```
int         *p = ( int * ) malloc( 3 * sizeof( int ) );
Fraction *q = ( Fraction * ) malloc( 2 * sizeof( Fraction ) );
```



Memory management



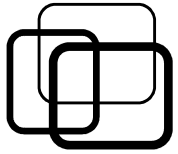
- Memory allocation:
  - **calloc(** <block count>, <block size> **).**
    - Return: allocated memory address or NULL (failed).

      ```
      int    *p = (int *) calloc( 3, sizeof(int) );
      Fraction *q = (Fraction *) calloc( 2, sizeof(Fraction) );
      ```

    - malloc vs. calloc?
  - **realloc(** <allocated address>, <bytes> **).**
    - Resize allocated memory.
    - Return: resized memory address or NULL (failed).

      ```
      int    *p = (int *) malloc( 2 * sizeof(int) );
      p[ 0 ] = 5;
      int   *q = (int *) realloc( p, 4 * sizeof(int) );
      ```

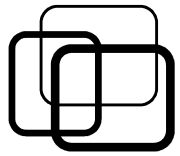
Memory management



- Memory de-allocation (library <stdlib.h>):
  - Return memory to RAM after use.
  - Memory leak problem:
    - Declared variables are auto return.
    - Allocated memory ARE NOT auto return.
    - Forget to return → memory leak.
  - **free(** <pointer> **)**.

```
float  *p = ( float * ) malloc( 20 * sizeof( float ) );
free( p );
p = NULL;   // Safe practice.
```

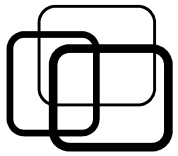
Memory management



- C++ memory management:
  - Is compatible with C (malloc, calloc, realloc, free).
  - Has new way to manage memory.
  - Operator **new**: allocate memory.
    - Syntax: **new** <type>**[**<number of elements>**]**;
    - Return: address of allocated memory.
  - Operator **delete**: de-allocate memory.
    - Syntax: **delete** <pointer>;

```
int         *p = new int [ 10 ];
Fraction *q = new Fraction [ 30 ];
delete [ ]p;
delete [ ]q;
```

Memory management



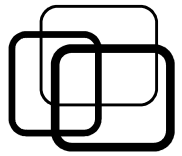
- Dynamic 1-D array:
  - Array has flexible size:
    - Use pointer.
    - Allocate memory as needed.
    - De-allocate when finish.

  ➔ Use memory more efficient.

```
void inputArray( int *&a, int &n ) {
        printf("Enter number of elements: ");
        scanf("%d", &n);
        a = new int [ n ];
        for (int i = 0; i < n; i++) {
              printf("Enter element %d:", i);
              scanf("%d", &a[ i ]);
        }
}
```

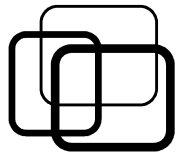
```
int main()
{
        int  *a;
        int  n;

        inputArray(a, n);
        delete [ ]a;
}
```

Summary



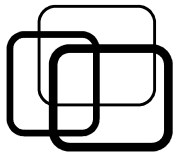
- Pointer concept:
  - Each variable has a memory address.
  - Pointer is variable storing other variable address.
- Pointer usage:
  - Declaration: <Data type> *.
  - Referencing (operator &): points to memory address.
  - De-referencing (operator *): read memory content.

Summary



- Pointer vs. Array:
  - Array is a pointer pointing to first element.
  - Operator [ ]: jump and de-reference.
  - Array argument is pointer passed to function.
- Memory management:
  - Allocation: malloc, calloc, realloc.
  - De-allocation: free.
  - Memory leak: allocated memory not returned.
  - C++:
    - Allocation: operator new.
    - De-allocation: operator delete.

Practice



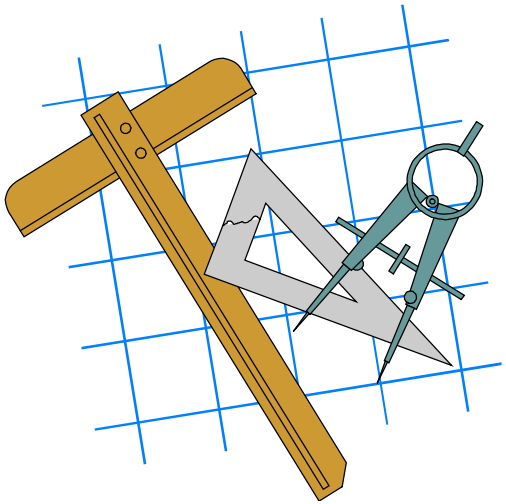
- ## Practice 7.1:

Given the following code:

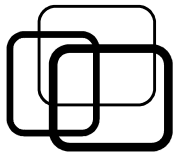
```
int main()
{
        int     *x, y = 2;
        float  *z = &y;

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

a) Fix error of the code.

b) After fixing, what is displayed on screen?



Practice

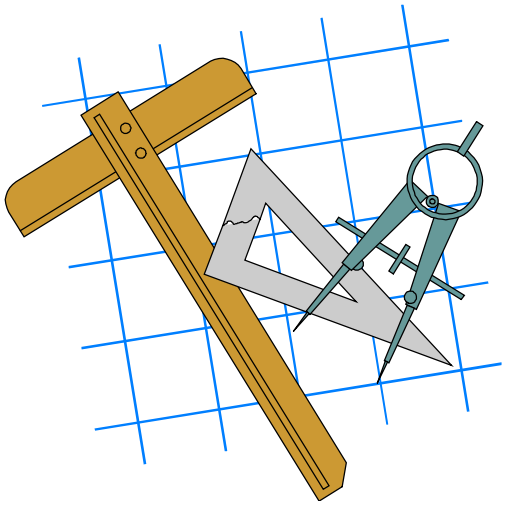


- Practice 7.2:

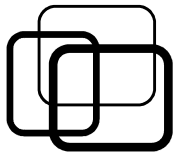
  Explain the difference amongst the following 3 functions:

```
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;
}
```



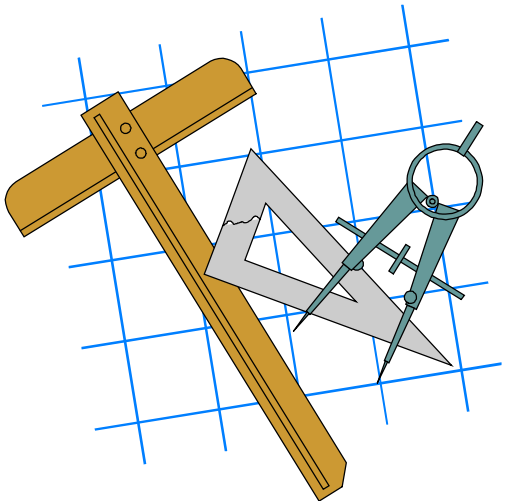
Practice



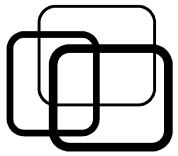
- **Practice 7.3:**

  Explain what the following program prints:

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

        p1 = m;
        p2 = &m[6];
        printf("%lld", p2 – p1);
}
```



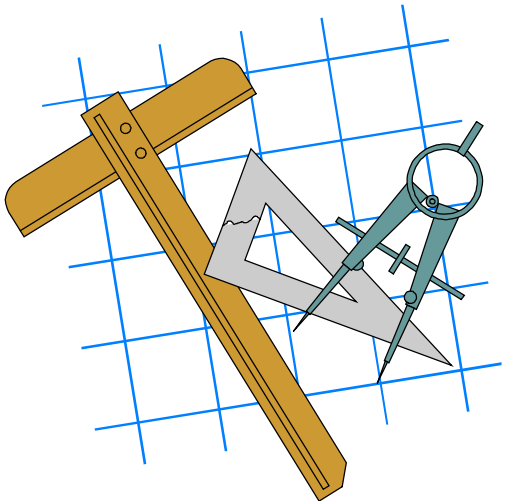
Practice



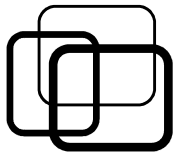
- Practice 7.4:

  Explain what the following program prints:

```
#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 7.5:

Write C/C++ program to use pointer as dynamic array:

- Enter an array of N fractions.
- Delete fractions having duplicate values (keep the first one).
- Print the result array.

Input format:

```
Number of fractions = 5
Fraction 0 = 1/2
Fraction 1 = 2/5
Fraction 2 = 4/8
Fraction 3 = 9/7
Fraction 4 = 18/14
```

Output format:

```
1/2 2/5 9/7
```