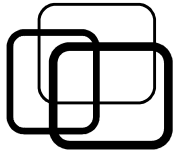
Advanced Pointer

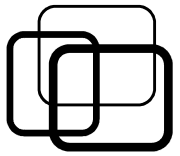
Inst. Nguyễn Minh Huy

Contents



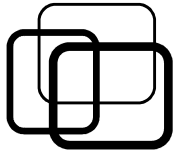
- Memory management.
- Pointer of pointer.
- Other types of pointers.

Contents



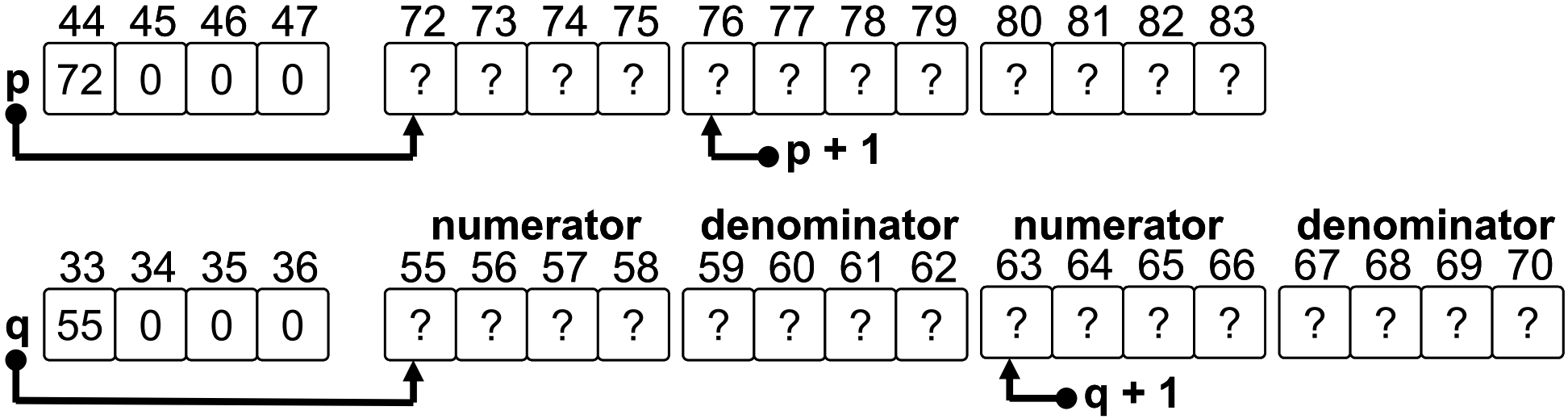
- **Memory management.**
- Pointer of pointer.
- Other types of pointers.

Memory management

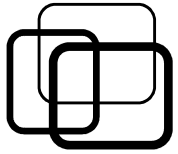


- Memory allocation in C:
  - Request memory from RAM.
  - **malloc**: #include <malloc.h>
    - Syntax: **malloc(** <number of bytes> **)**;
    - Return: address of allocated memory.

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



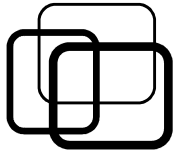
Memory management



- Memory de-allocation in C:
  - Return memory to RAM.
  - Memory management rule in C:
    - Declared variables are auto de-allocated.
    - Allocated memory are not auto de-allocated.
    - Forget to de-allocate memory → memory leak.
  - **free**: #include <malloc.h>
    - Syntax: **free(** <pointer> **)**;

      float *p = (float *) **malloc(** 20 * sizeof(float) **)**;
      free(p **)**;

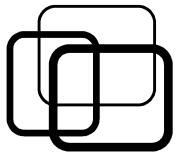
Memory management



- Memory management in C++:
  - C++ is compatible with C (support malloc).
  - C++ has new method for memory management.
  - **new** operator: allocate memory.
    - Syntax: **new** <type>**[**<number of elements>**]**;
    - Return: address of allocated memory.
  - **delete** operator: de-allocate memory.
    - Syntax: **delete** <pointer>;

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

Memory Management



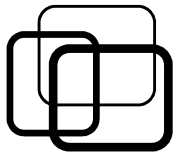
- Dynamic 1-D array:
  - Array has flexible size:
    - Use pointer.
    - Allocate memory as needed.
    - De-allocate when finish.

  ➔ Use memory effectively.

```
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 ]);
        }
}
```

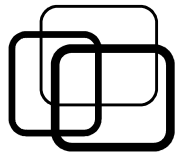
```
void main()
{
        int  *a;
        int  n;

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

Contents



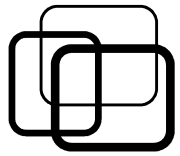
- Memory management.
- **Pointer of pointer.**
- Other types of pointers.

Pointer of pointer



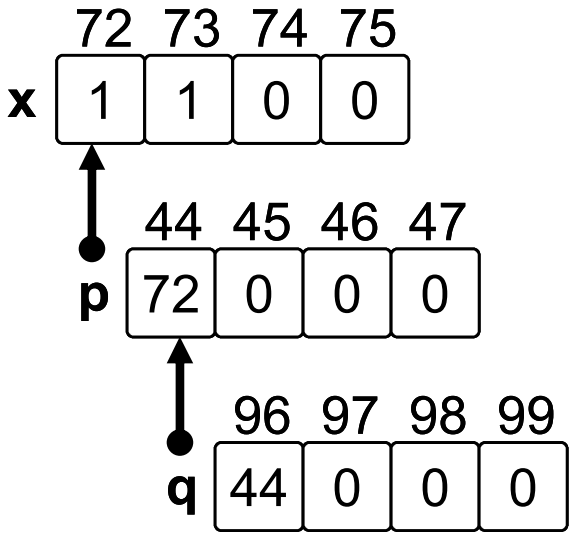
- Address of pointer:
  - Variable has an address.
    - int has address int *.
  - Pointer also has an address.
    - int * has address type?
  - Pointer of pointer:
    - A variable stores address of another pointer.
    - Declaration: **<pointer type> *** <pointer name>;

Pointer of pointer

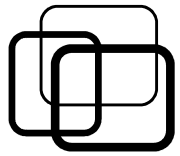


- Pointer of pointer in C:
  - Declaration:
    - Method 1: use *****.
    - Method 2: use **typedef**.
  - Initialization:
    - Use **NULL**.
    - Use **&** operator.

```
int  x = 257;
int  *p = NULL;
int  **q = NULL;

p = &x;
q = &p;
```



Pointer of pointer

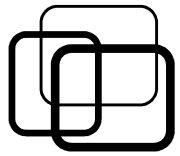


- Pointer of pointer in C:
  - Access memory content:
    - 1-level access: operator *.
    - 2-level access: operator **.
  - Passing argument:
    - Pass-by-value.
    - Pass-by-reference.

➔ Which values are changed in foo() ?

```
void foo(int **g, int **&h)
{
        (**g)++; (*g)++; g++;
        (**h)++; (*h)++; h++;
}

void main()
{
        int  a[10];
        int  *p = a;
        int  **q = &p;
        int  **r = &p;

        foo(q, r);
}
```

Pointer of pointer

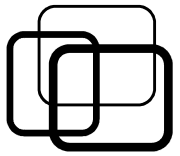


- Dynamic matrix:
  - Array of pointers:
    - Level-1 pointer is 1-dimensional dynamic array.
    - Level-2 pointer is 2-dimensional dynamic array.

```
void inputMatrix(int **&m, int &rows, int &cols) {
        cout << "Enter rows and cols = ";
        cin >> rows >> cols;

        m = new int * [ rows ];
        for (int i = 0; i < rows; i++) {
              m[ i ] = new int [ cols ];
              for (int j = 0; j < cols; j++)
                    cin >> m[ i ][ j ];
        }
}
```

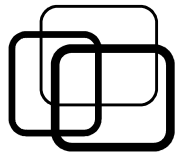
```
void main()
{
        int  **m;
        int  rows, cols;

        inputMatrix(m, rows, cols);
        delete [ ]m;
}
```

Contents



- Memory management.
- Pointer of pointer.
- **Other types of pointers.**

Other types of pointers



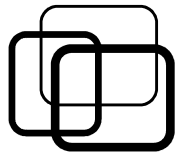
- Constant pointer:
  - Pointer points to only 1 address “for life”.
  - Declaration: <type> * **const** <pointer name>;

```
int  x = 5, y = 6;
int * const p = &x;
p = &y;     // Wrong.
```

  - All static arrays in C are constant pointers.
- Pointer to constant:
  - Memory content pointer points to cannot be changed.
  - Declaration: **const** <type> * <pointer name>;

```
int  x = 5;
const int *p = &x;
*p = 6;      // Wrong.
```

Other types of pointers

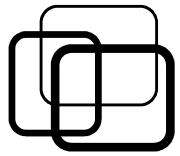


- void pointer:
  - Pointer can store address of any types.
  - Declaration: **void *** <pointer name>.
  - Cast to specific type when accessing content.

```
void printBytes(void *p, int size)
{
        char *q = ( unsigned char * ) p;
        for ( int i = 0; i < size; i++ )
                printf( "%d ", q[ i ] );
}
```

```
void main()
{
        int     x = 1057;
        double y = 1.25;

        printBytes(&x, 4);
        printBytes(&y, 8);
}
```

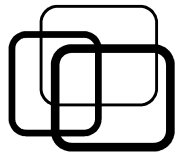
Other types of pointers



- Function pointer:
  - Function address:
    - Functions are also stored in memory.
    - Each function has an address.
  - Function pointer stores address of function.
  - Declaration:

    <return type> (* <pointer name>**) (<arguments>)**;
    typedef <return type> (* <alias>**) (<arguments>)**;
    <alias> <pointer name>;
  - Functions have same address type if:
    - Same return type.
    - Same arguments.

Other types of pointers

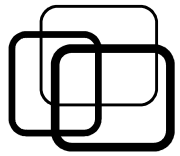


- Function pointer:

```
typedef  int  (*Operator)(int a, int b);

int add(int u, int v)
{
        return u + v;
}
int mul(int u, int v)
{
        return u * v;
}
int calculate(int u, int v, Operator p)
{       // u^3 operator v^2.
        return p(u*u*u, v*v);
}
```

```
void main()
{
        int  x = 5;
        int  y = 6;

        Operator  p = add;
        int r1 = p(x, y);

        p = mul;
        int r2 = p(x, y);

        int r3 = calculate(x, y, add);
}
```

Other types of pointers



- Pointer to fix-sized memory:
  - Address of static array:
    - What address type of static array?

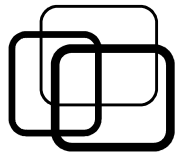
```
int   a[ 10 ];
int   *p = a          // p and a store address of a[ 0 ].
???  q = &a;
```

  - Pointer to fix-sized memory:
    - Pointer stores address of static array.
    - Declaration:

<array type> **(***<pointer name>**)[**<array size>**]**;

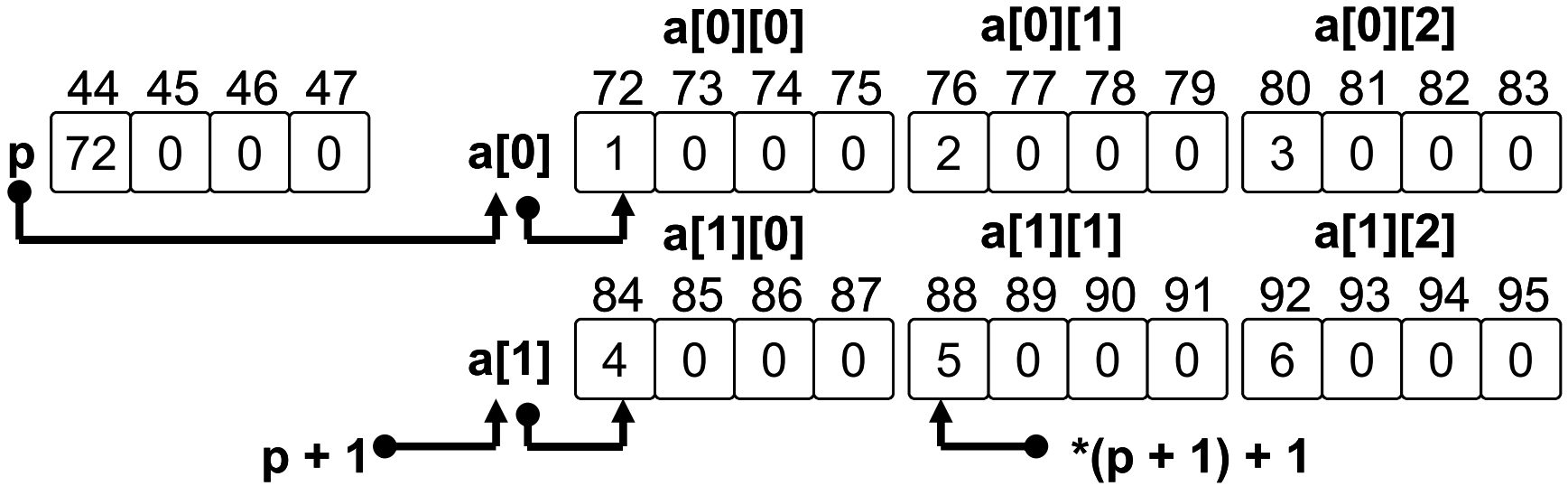
```
int  a[ 10 ];
int  ( *p )[ 10 ] = &a;   // p points to 10-element array.
```

Other types of pointers

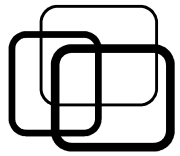


- Pointer to fix-sized memory:
  - Static 2-D array in C:
    - Is pointer to fix-sized 1-D array.
    - Stores address of the first row.

```
int  a[ 2 ] [ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
int ( *p )[ 3 ] = a;    // a = &a[ 0 ].
printf("%d\n", *( *(p + 1) + 1 ) );
```



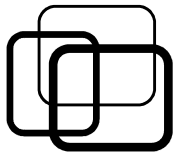
Other types of pointers



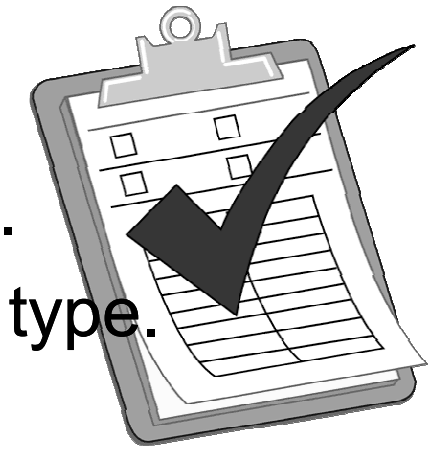
- Pointer to fix-sized memory:
  - Passing static 2-D array to function:
    - Not passing whole array.
    - Only passing address of first row.

```
void printMatrix(int a[ ][20], int rows, int cols)  {  // pass &a[ 0 ].
        for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++)
                        printf("%d ", a[ i ][ j ] );
                printf("\n");
        }
}
void main() {
        int  a[10][20];
        printMatrix(a, 10, 20);
}
```

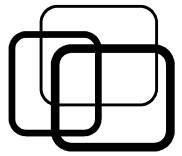
Summary



- Memory management:
  - Allocate:
    - Get memory from RAM.
    - malloc, new operator (C++).
  - De-allocate:
    - Return memory to RAM.
    - free, delete operator (C++).
  - Level-1 pointer is dynamic 1-D array.
- Types of pointers:
  - Different types → different address types.
  - Each address type stored by one pointer type.

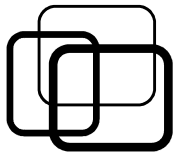


Summary



- Types of pointers:
  - Pointer of pointer → stores address of pointer.
  - Constant pointer → stores constant address.
  - Pointer to constant → stores address of constant.
  - void pointer → stores address of any types.
  - Function pointer → stores address of function.
  - Pointer to fix-sized memory → stores address of static array.

Practice



- ## Practice 3.1:

Given static 2-D array as follow:

```
int  m[4][6];
```

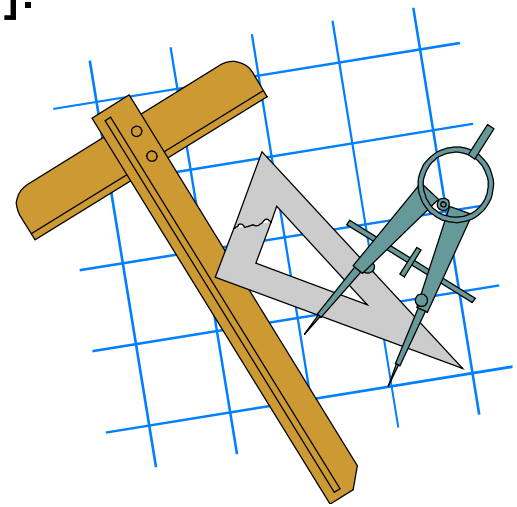
What types of addresses of the following variables?

a) m[1][3].

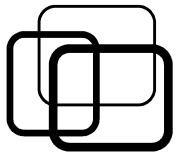
b) m[0].

c) m.

Write code to access m[2][4] without using operator [ ].



Practice



Practice 3.2:

Given the following C code:

```
void initialize(double **p)
{
        for (int i = 0; i < 5; i++)
                *(p + i) = new double[ i + 1 ];
}
```

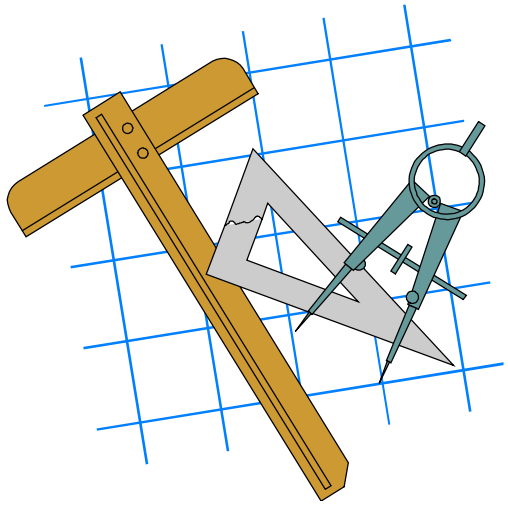
```
void main()
{
        double *p[10];
        initialize(p + 3);
        release(p);
}
```

Answer the following questions:

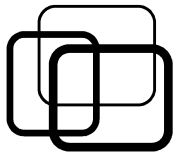
a) How many bytes are allocated at each line of main( ) ?

b) How memory are allocated by initialize( ) function in main( ) ?

c) Write release( ) function to avoid memory leak.



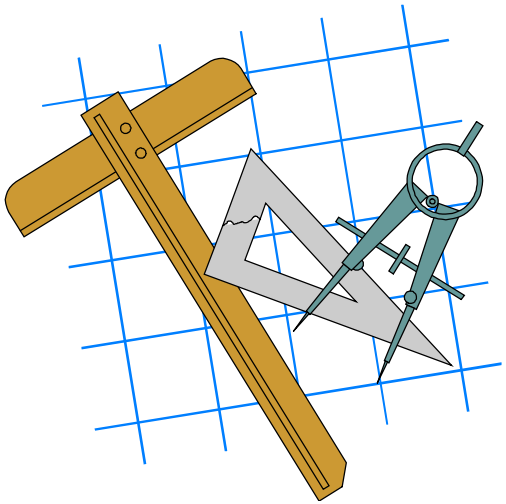
Practice



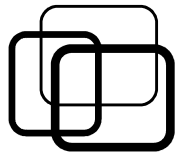
- Practice 3.3:

  Declare address type for the following functions (use typedef):

  ```
  void process();
  int power(int x, int n);
  int * inputArray(int &n);
  void printArray(int a[ ], int n);
  Fraction add(Fraction f1, Fraction f2);
  ```



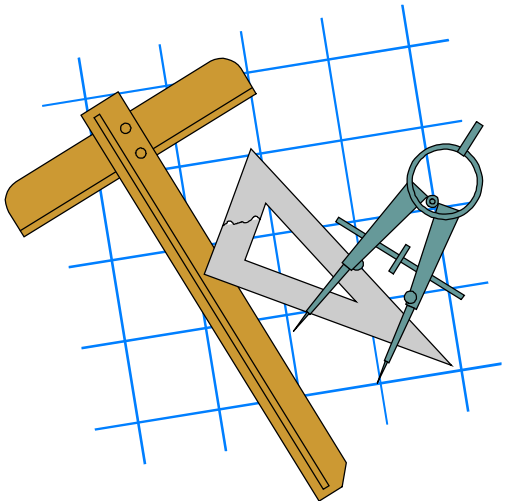
Practice



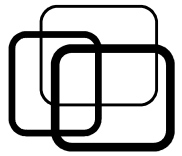
- Practice 3.4:

  Write C program (use dynamic matrix) to do the followings:

  - Enter from keyboard matrix of M x N integers.
  - Get a list of primes from the input matrix.
  - Print the prime list to screen.



Practice



- ## Practice 3.5 (*):

  Write C program to sort an input array of N integers, the sort order is defined by user.

  Notes: use function pointer to pass user-defined sort order function to sort function.