Pointer Advances

Objectives

- Level 2 pointer
- Pointer & multi dimensional array
- Array of pointers
- Function pointer

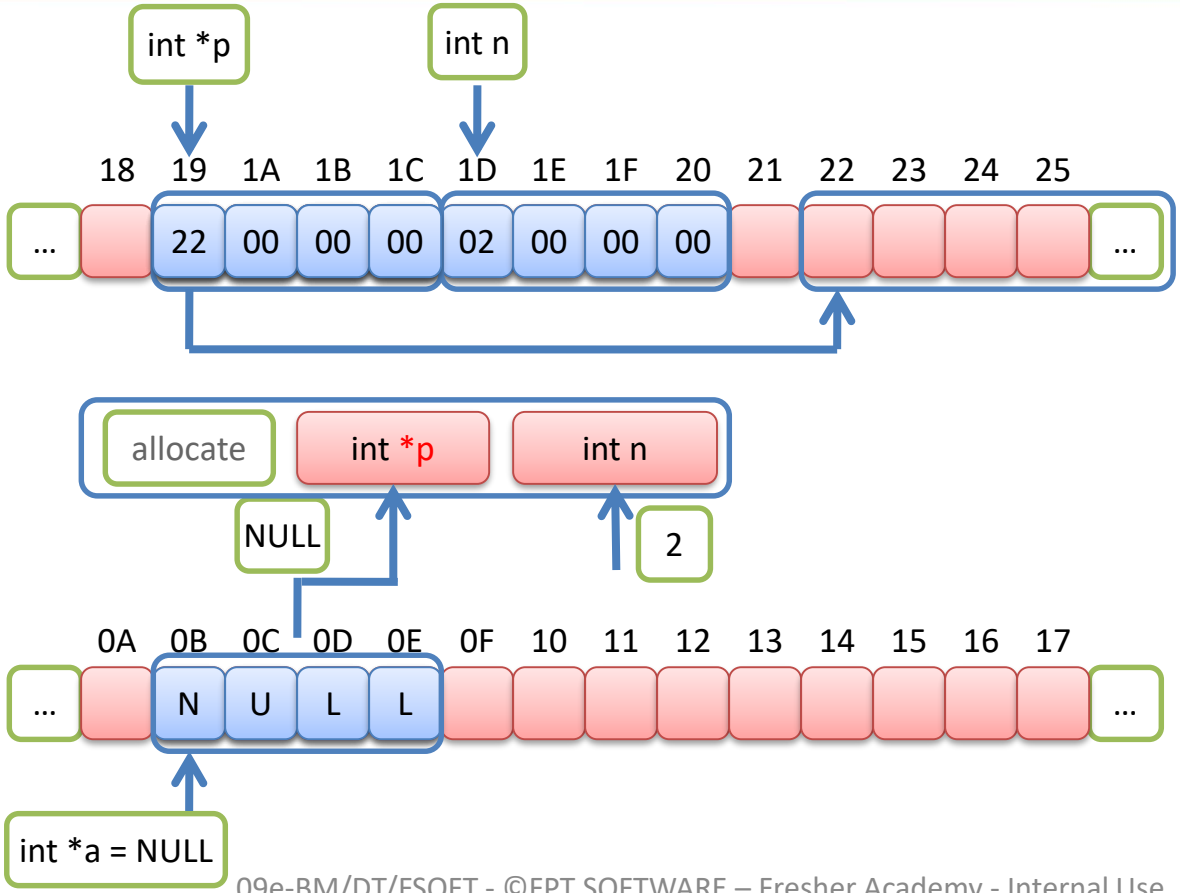
Level 2 pointer (pointer point to pointer)

- Problem

  How to change value of pointer (not value it point to) after calling function?

```
void Allocate(int *p, int n)
{
    p = (int *)malloc(n * sizeof(int));
}
void main()
{
    int *a = NULL;
    Allocate (a, 2);
    // a still = NULL
}
```

Level 2 pointer - 1



Level 2 pointer - 2

- Solution
  - ✓ Using reference int *&p (in C++)

```
void CapPhat(int *&p, int n)
{
    p = (int *)malloc(n * sizeof(int));
}
```

  - ✓ Not change parameter directly and return

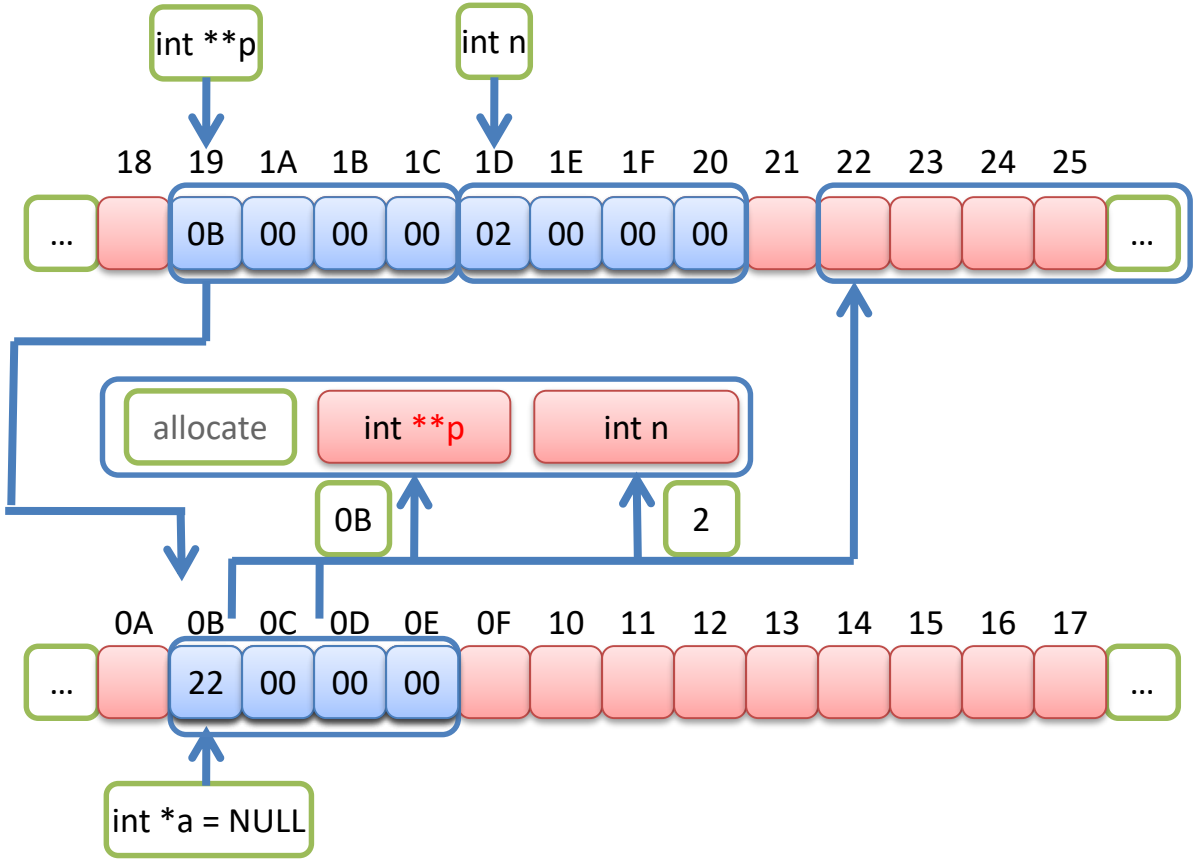
```
int* Allocate(int n)
{
    int *p = (int *)malloc(n * sizeof(int));
    return p;
}
```

Level 2 pointer - 3

- Solution
  - ✓ Using pointer p point to pointer a. The function will change value of pointer a indirect through pointer p.

```
void Allocate(int **p, int n)
{
    *p = (int *)malloc(n * sizeof(int));
}

void main()
{
    int *a = NULL;
    Allocate (&a, 4);
}
```

Level 2 pointer - 4



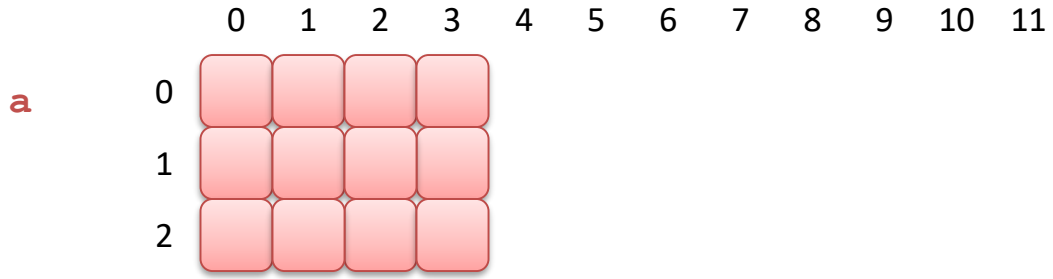
Level 2 pointer - 5

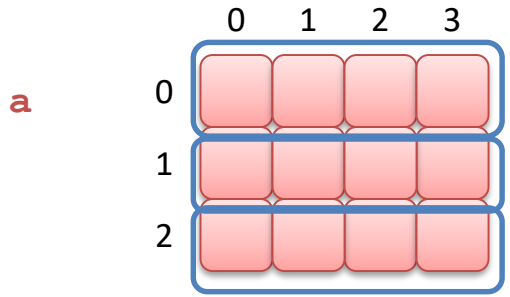
- Note

```
int x = 12;
int *ptr = &x;            // OK
int k = &x; ptr = k;          // Error

int **ptr_to_ptr = &ptr;      // OK
int **ptr_to_ptr = &x;        // Error

**ptr_to_ptr = 12;        // OK
*ptr_to_ptr = 12;             // Error

printf("%d", ptr_to_ptr);     // Address of ptr
printf("%d", *ptr_to_ptr);    // Value of ptr
printf("%d", **ptr_to_ptr);   // Value of x
```

Pointer & 2 dimensional array - 1

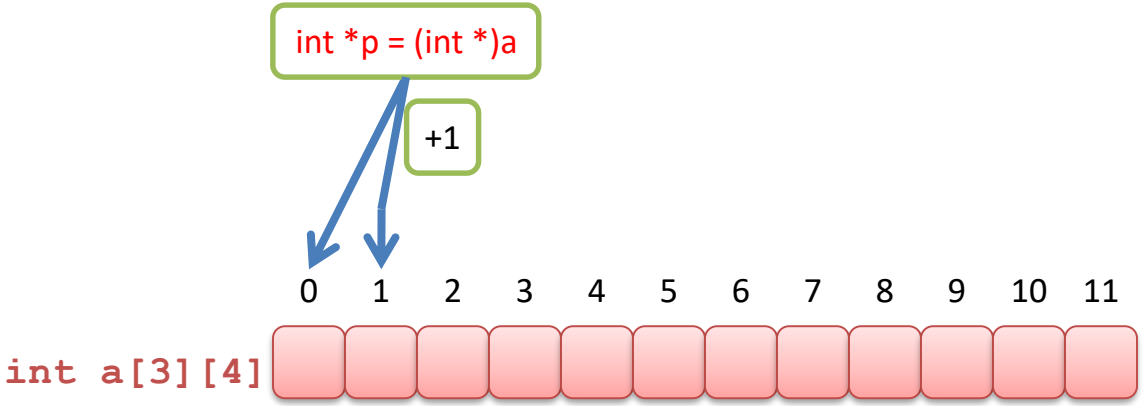
```
int a[3][4];
```





Pointer & 2 dimensional array - 2

- Method 1
  - ✓ Elements create 1 dimensional array
  - ✓ Using pointer int * to access 1 dimensional array



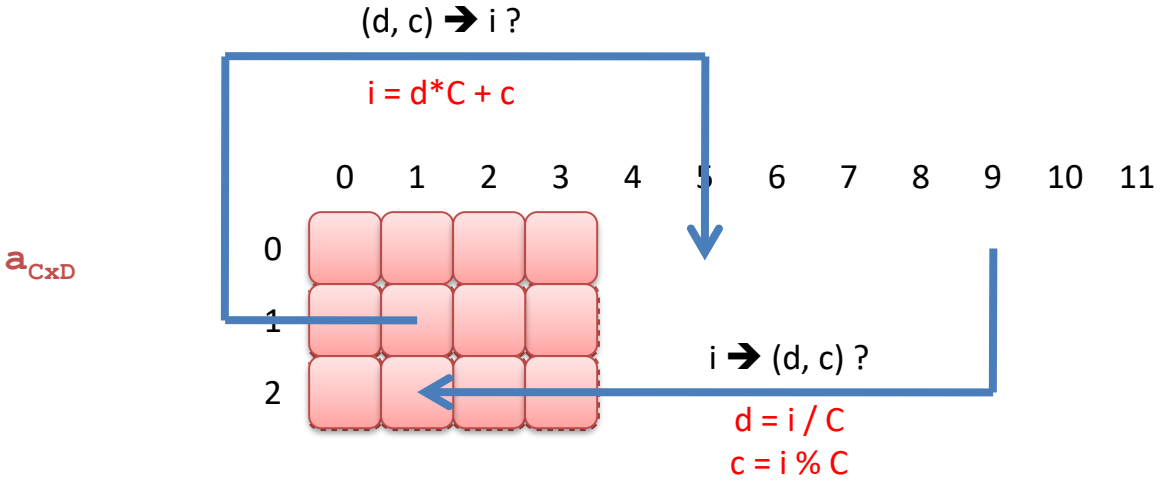
Method 1 - 1

- Input / Output by index of 1 dimensional array

```
#define D 3
#define C 4
void main()
{
    int a[D][C], i;
    int *p = (int *)a;
    for (i = 0; i < D*C; i++)
    {
        printf("Input element %d: ", i);
        scanf("%d", p + i);
    }

    for (i = 0; i < D*C; i++)
        printf("%d ", *(p + i));
}
```

Method 1 - 2

- Relationship between index of 1 & 2 dimensional array



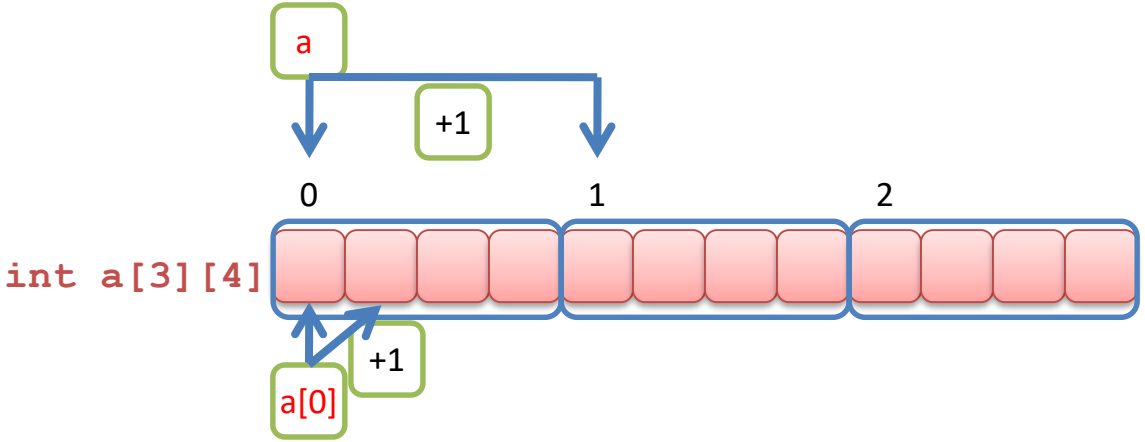
Method 1 - 3

- Input/ Output by index of 2 dimensional array

```
int a[D][C], i, d, c;
int *p = (int *)a;

for (i = 0; i < D*C; i++)
{
    printf("Input a[%d][%d]: ", i / C, i % C);
    scanf("%d", p + i);
}
for (d = 0; d < D; d++)
{
    for (c = 0; c < C; c++)
        printf("%d ", *(p + d * C + c));
      // *p++ printf("\n");
}
```

Pointer and 2 dimensional array

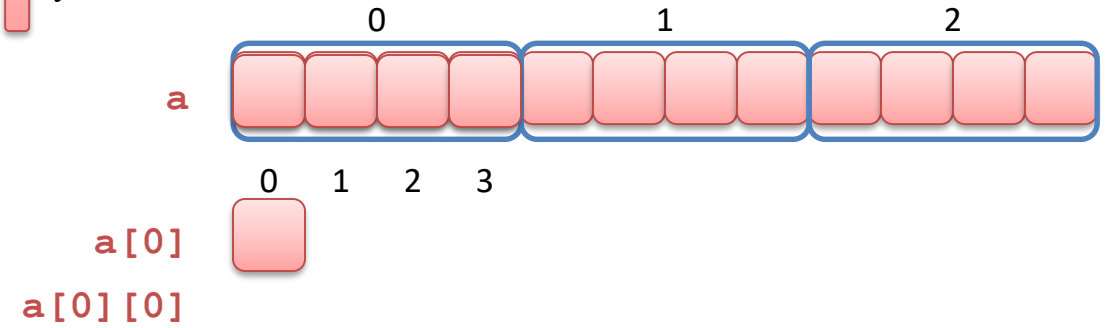
- Method 2
  - ✓ 2 dimensional array, each element is one 1 dimensional array
    - a contains a[0], a[1], … ➔ a = &a[0]
    - a[0] contains a[0][0], a[0][1], … ➔ a[0] = &a[0][0]



Method 2 - 1

- The size of array

```
void main()
{
    int a[3][4];
    printf("Size of a = %d", sizeof(a));
    printf("Size of a[0] = %d", sizeof(a[0]));
    printf("Size of a[0][0] = %d", sizeof(a[0][0]));
}
```



Method 2 - 2

- Comment
  - ✓ a points to a[0], a[0] points to a[0][0] ➔ a is level 2 pointer.
  - ✓ Access a[0][0] by 3 ways:

```
void main()
{
    int a[3][4];
    a[0][0] = 1;
    *a[0] = 1;
    **a = 1;

    a[1][0] = 1; *a[1] = 1; **(a+1) = 1;
    a[1][2] = 1; *(a[1]+2) = 1; *(*(a+1)+2) = 1;
}
```

Method 2 - 3

- Pass array to function
  - ✓ Pass address of the first element to function.
  - ✓ Declare pointer and assign address of array to the pointer so it points to the array.
  - ✓ The pointer must have the same type with array, that's mean the pointer points to memory of n elements.
- Syntax

```
<data type> (*<pointer name>)[<number of elements>];
```

- Example

```
int (*ptr)[4];
```

Method 2 - 4

- Pass array to function

```
void Output_1_Array_C1(int (*ptr)[4])    // ptr[][4]
{
    int *p = (int *)ptr;
    for (int i = 0; i < 4; i++)
        printf("%d ", *p++);
}
void main()
{
    int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    int (*ptr)[4];
    ptr = a;
    for (int i = 0; i < 3; i++)
        Output_1_Array_C1(ptr++);  // or ptr + i
        Output_1_Array_C1(a++);    // wrong => a + i
}
```

Method 2 - 5

- Pass array to function

```
void Output_1_Array_C2(int *ptr, int n) // ptr[]
{
    for (int i = 0; i < n; i++)
        printf("%d ", *ptr++);
}
void main()
{
    int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    int (*ptr)[4];
    ptr = a;
    for (int i = 0; i < 3; i++)
        Output_1_Array_C2((int *)ptr++);
        Output_1_Array_C2((int *)(a + i));// a++
wrong
}
```

Method 2 - 6

- Pass array to function

```
void Output_n_Array_C1(int (*ptr)[4], int n)
{
    int *p = (int *)ptr;
    for (int i = 0; i < n * 4; i++)
        printf("%d ", *p++);
}
void main()
{
    int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    int (*ptr)[4];
    ptr = a;

    Output_n_Array_1(ptr, 3);
    Output_n_Array_1(a, 3);
}
```

Method 2 - 7

- Pass array to function

```
void Output_n_Array_C2(int (*ptr)[4], int n)
{
    int *p;
    for (int i = 0; i < n; i++)
    {
        p = (int *)ptr++;

        for (int i = 0; i < 4; i++)
            printf("%d ", *p++);
        printf("\n");
    }
}
```

Array of pointers - 1

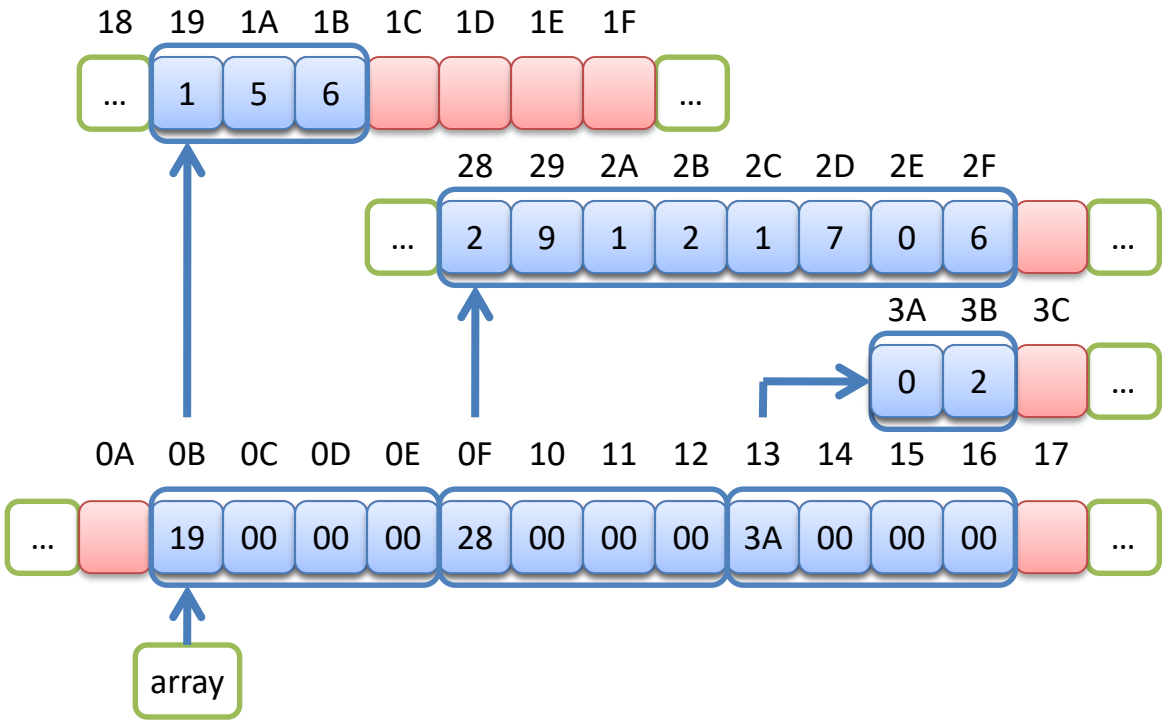
- Problem
  - ✓ Use which data structure to store the data below?

	0	1	2	3	4	5	6	7
0	1	5	6					
1	2	9	1	2	1	7	0	6
2	0	2						

- Solution?
  - ✓ Way 1: 2 dimensional array 3x8 (waste memory)

Array of pointers - 2

✓ Way 2: 1 dimensional array of pointers



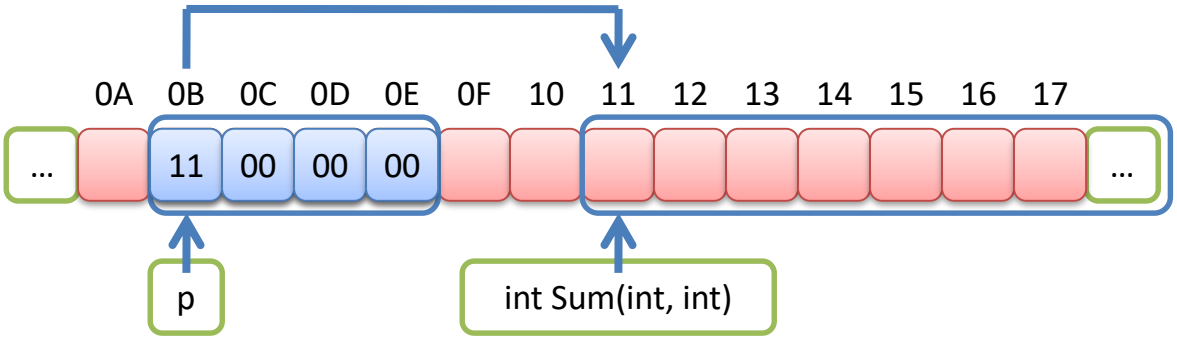
Array of pointers - 3

- Example

```
void print_strings(char *p[], int n)
{
    for (int i = 0; i<n; i++)
        printf("%s ", p[i]);
}

void main()
{
    char *message[4] = {"Fpt", "Software",
"Workforce", "Assurance"};

    print_strings(message, 4);
}
```

Function pointer - 1

- Concept
  - ✓ Functions are stored in memory, they have address.
  - ✓ Function pointer is pointer point to memory of function and call function through the pointer.



Function pointer - 2

- Exclusive declare

```
<return type> (* <pointer name>)(paramenter list);
```

- Example

```
// Pointer to function with int parameters, return int
int (*ptof1)(int x);

// Pointer to function with 2 params double, return nothing
void (*ptof2)(double x, double y);

// Pointer to function with array parameter, return char
char (*ptof3)(char *p[]);

// Pointer to function do not have arguments and return nothing
void (*ptof4)();
```

Function pointer - 3

- Implicit declare (through type)

```
typedef <return type> (* <type name>)(params list);
<type nam> <pointer name>;
```

- Example

```
int (*pt1)(int, int);   // Exclicit

typedef int (*Operator)(int, int);

Operator pt2, pt3;  // Implicit
```

Function pointer - 4

- Assign value to function pointer

```
<func pointer> = <func name>;
<func poiter> = &<func name>;
```

✓ Assigned function must have the same prototype (input, output)

- Example

```
int Sum(int x, int y);            // Function
int Subtraction(int x, int y);    // Function
int (*calculate)(int x, int y); // func pointer

calculate = Sum;     // short style
calculate = &Subtraction;   // using address
calculate = NULL;    // point to nothing
```

Function pointer - 5

- Compare function pointer

```
if  (calculate != NULL)
{
    if (calculate == &Sum
            printf("Pointer to Sum function");
    else
        if (calculate== &Subtraction
            printf("Pointer to Sub function");
    else
        printf("Pointer to other functions");
}
else
    printf("Not declared function pointer");
```

Function pointer - 6

- Call function through function pointer
  - ✓ Using “*” operator (formal) but this case can be ignored

```
int Sum(int x, int y);
int Subtraction(int x, int y);

int (*calculate)(int, int);

calculate = Sum;
int kq1 = (*calculate)(1, 2);    // Formal
int kq2 = calculate(1, 2);  // Short style
```

Function pointer - 7

- Pass parameter as function pointer

```
int Sum(int x, int y);
int Subtraction(int x, int y);
int Calculate(int x, int y, int (*operator)(int,
int))
{
    int kq = (*operator)(x, y); // Call function
    return kq;
}

void main()
{
    int (*operator)(int, int) = &Sum;
    int kq1 = Calculate(1, 2, operator);
    int kq2 = Calculate(1, 2, &Subtraction);
}
```

Function pointer - 8

- Return function pointer

```
int (*GetOperator(char code))(int, int)
{
    if (code == '+')
        return &Sum;
    return &Subtraction;
}

void main()
{
    int (*operator)(int, int) = NULL;
    operator = GetOperator('+');
    int kq2 = operator(1, 2, &Subtraction);
}
```

Function pointer - 9

- Return function pointer

```
typedef (*Operator)(int, int);
Operator GetOperator(char code)
{
    if (code == '+')
        return &Sum;
    return &Subtraction;
}

void main()
{
    Operator operator = NULL;
    operator = GetOperator('+');
    int result2 = operator(1, 2, &Subtraction);
}
```

Function pointer - 10

- Array of function pointers

```
typedef (*Operator)(int, int);
void main()
{
    int (*array1[2])(int, int); // explicit
    Operator array2[2];     // implicit

    array1[0] = array2[1] = &Sum;
    array1[1] = array2[0] = &Subtraction;

    printf("%d\n", (*array1[0])(1, 2));
    printf("%d\n", array1[1](1, 2));
    printf("%d\n", array2[0](1, 2));
    printf("%d\n", array2[1](1, 2));
}
```

Function pointer - 11

- Note
  - ✓ Do not miss (*) when declare function pointer
    - int (*Operator)(int x, int y);
    - int *Operator(int x, int y);
  - ✓ Can skip parameter name when function pointer
    - int (*Operator)(int x, int y);
    - int (*Operator)(int, int);

Callback Function - 1

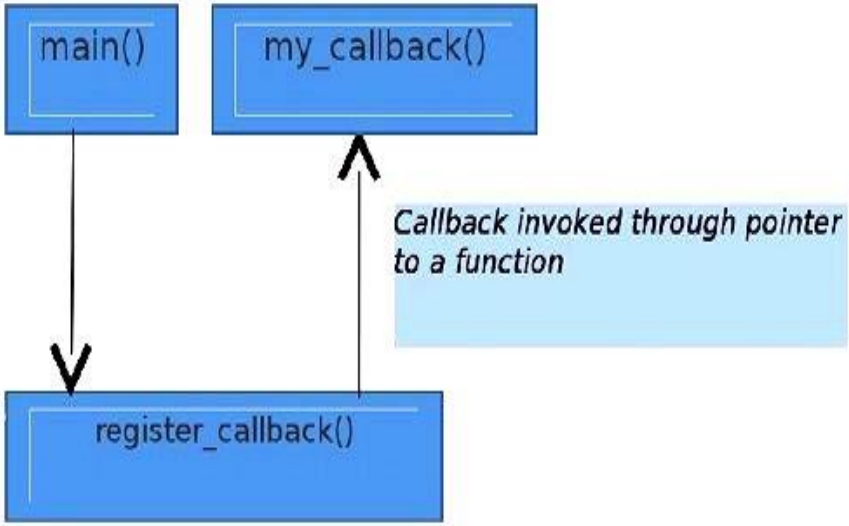
- Concept

  - ✓ Callback function is a function that is called through a function pointer which is passed as an argument from another method.
  - ✓ When that pointer is used to call the function it points to, it is said that a call back is made.

Callback Function - 2

- Example 1

```
typedef void (*callback)(void);

void my_callback(void)
{
  cout << “inside my_callback\n";
}

void register_callback(callback ptr_callback)
{
  (*ptr_callback)();
}

int main(void)
{
  callback ptr_my_callback = my_callback;
  register_callback(ptr_my_callback);
  return 0;
}
```



Callback Function - 3

- Example 2

```
typedef double (*callback)(double a, double b);

double Plus(double a, double b)
{ return a + b;}

double Minus(double a, double b)
{ return a – b;}

void register_callback(callback ptr_callback,double a, double b) {
    (*ptr_callback)(a, b);
}

int main(void) {
    callback ptr_callback = Plus;
    cout << “result = “ << register_callback(ptr_callback, 3, 4) << endl;

    ptr_callback = Minus;
    cout << “result = “ << register_callback(ptr_callback, 3, 4) << endl;
}
```

```
Output:
  result = 7
  result = -1
```

Callback Function - 4

- Implement a Callback to a static C++ Member Function
  - ✓ This is the same as implement callbacks to C++ functions.

```
typedef void (*callback)(void);

class MyClass{
public:
  static void StaticCallBack(void) {cout << “inside my_callback\n";}
};

void register_callback(callback ptr_callback) {
  (*ptr_callback)();
}

int main(void) {
  callback ptr_my_callback = &MyClass::StaticCallBack;
  register_callback(ptr_my_callback);
  return 0;
}
```

Callback Function - 5

- Implement a Callback to a non-static C++ Member Function
- Pointers to non-static members need the this-pointer of a class object to be passed and write a static member function as a wrapper.

```
typedef void (*callback)(void*);

class MyClass{
public:
  void CallBack(void) {cout << “inside my_callback\n";}
  static void Wrapper_To_Call(void* ptObject)
  {
       // explicitly cast to a pointer to MyClass
      MyClass* objA = (MyClass*) ptObject;

      // call member
      objA->CallBack();
  }
};
```

Callback Function - 6

- Implement a Callback to a non-static C++ Member Function (cont.)

```
void register_callback(void* ptobject, callback ptr_callback)
{
  (*ptr_callback)(ptobject);
}

int main(void)
{
  MyClass objA;
  callback ptr_my_callback = &MyClass ::Wrapper_To_Call;
  register_callback((void*)&objA, ptr_my_callback);
  return 0;
}
```

Thank you

Q&A