# Pointers and
# Dynamic Memory Management

# Intended Learning Outcomes

- Distinguish between little endian and big endian
- Describe how to use pointers
- Define a copy constructor structure
- Distinguish between shallow copy and deep copy

# Little Endian and Big Endian

How are numbers stored in memory space?

# Decimal and Hexadecimal Numbers

| Decimal number | |
|---|---|
| 9 | |
| 10 | |
| 15 | |
| 255 | |
| 128 | |
| 258 | |
| | |

Which direction should we read numbers from?

# Decimal and Hexadecimal Numbers

| Decimal number | |
|---|---|
| 9 | |
| 10 | |
| 15 | |
| 255 | |
| 128 | |
| 258 | |
| | |

Which direction should we read numbers from? Read from A1

# Decimal and Hexadecimal Numbers

| Decimal number | Hexadecimal number |
|---|---|
| 9 | 9 |
| 10 | A |
| 15 | F |
| 255 | FF |
| 128 | 80 |
| 258 | 0102 |
| | 12 34 56 78 AB CD EF 01 |

Which direction should we read numbers from? Read from left to right

# Decimal and Hexadecimal Numbers

| Decimal number | Hexadecimal number |
|---|---|
| 9 | 9 |
| 10 | A |
| 15 | F |
| 255 | FF |
| 128 | 80 |
| 258 | 0102 |
| | 12 34 56 78 AB CD EF 01 |

**How are the numbers stored in memory? How do we order the bytes?**

# Decimal and Hexadecimal Numbers

| Decimal number | Hexadecimal number |
|---|---|
| 9 | 9 |
| 10 | A |
| 15 | F |
| 255 | FF |
| 128 | 80 |
| 258 | 0102 |
| | 12 34 56 78 AB CD EF 01 |

**How are the numbers stored in memory? How do we order the bytes?**

# Decimal and Hexadecimal Numbers

| Decimal number | Hexadecimal number |
|---|---|
| 9 | 9 |
| 10 | A |
| 15 | F |
| 255 | FF |
| 128 | 80 |
| 258 | 0102 |
| | 12 34 56 78 AB CD EF 01 |

**Significant bytes**

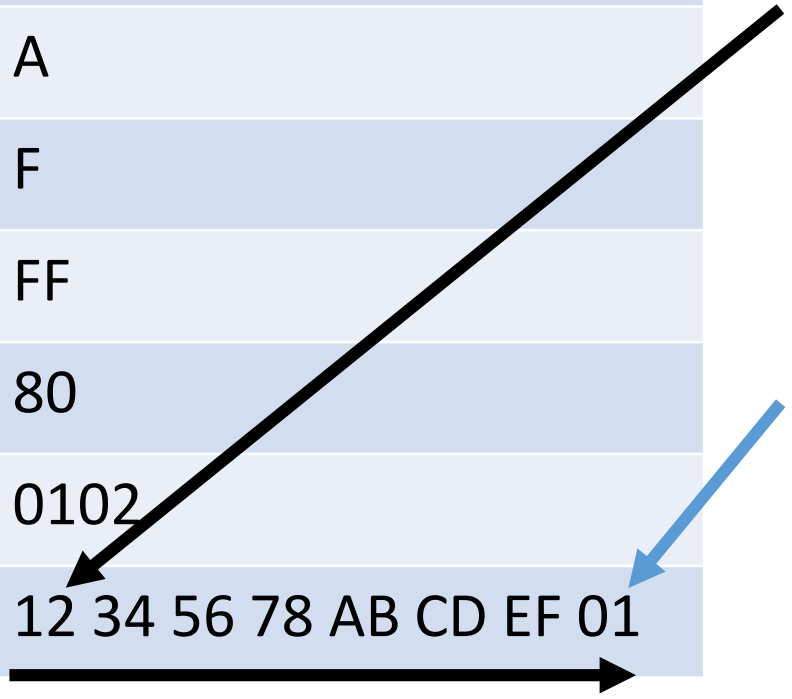
**Most significant byte (MSB)**
The byte in that position of a multi-byte number which has the **greatest** potential value.

**Least significant byte (LSB)**
The byte in that position of a multi-byte number which has the **least** potential value.

**How are the numbers stored in memory? How do we order the bytes?**

# Decimal and Hexadecimal Numbers

| Decimal number | Hexadecimal number |
|---|---|
| 9 | 9 |
| 10 | A |
| 15 | F |
| 255 | FF |
| 128 | 80 |
| 258 | 0102 |
| | 12 34 56 78 AB CD EF 01 |



**Significant bytes**

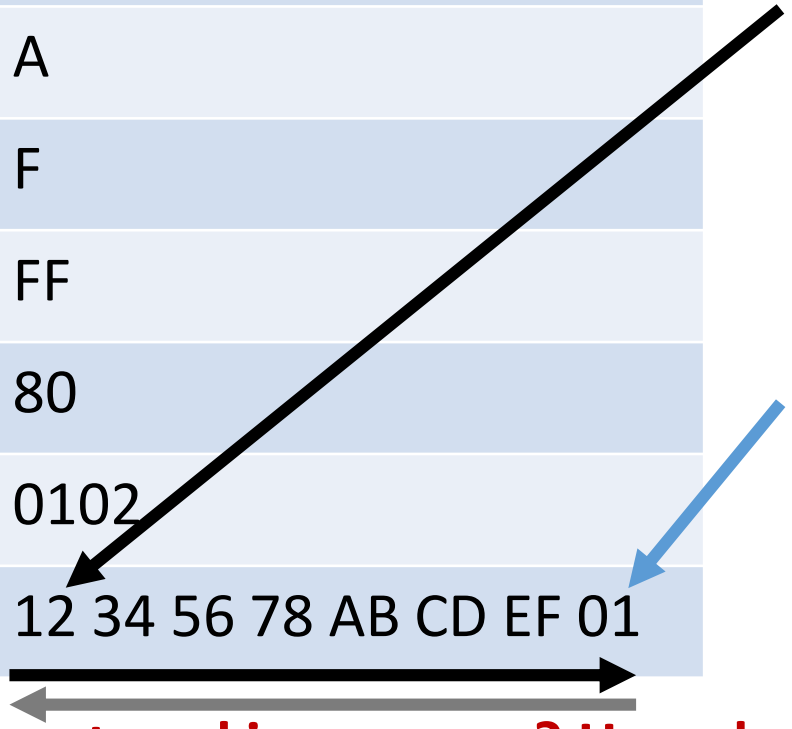
**Most significant byte (MSB)**
The byte in that position of a multi-byte number which has the **greatest** potential value.

**Least significant byte (LSB)**
The byte in that position of a multi-byte number which has the **least** potential value.

**How are the numbers stored in memory? How do we order the bytes?**

# Decimal and Hexadecimal Numbers

| Decimal number | Hexadecimal number |
|---|---|
| 9 | 9 |
| 10 | A |
| 15 | F |
| 255 | FF |
| 128 | 80 |
| 258 | 0102 |
| | 12 34 56 78 AB CD EF 01 |



**Significant bytes**

**Most significant byte (MSB)**
The byte in that position of a multi-byte number which has the **greatest** potential value.

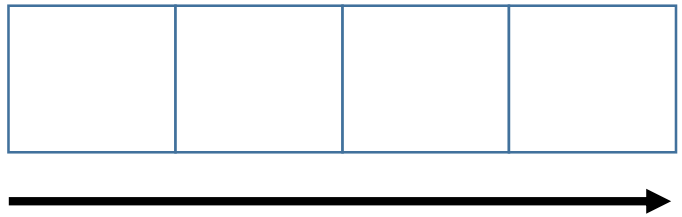
**Least significant byte (LSB)**
The byte in that position of a multi-byte number which has the **least** potential value.

**How are the numbers stored in memory? How do we order the bytes?**

# Little Endian and Big Endian

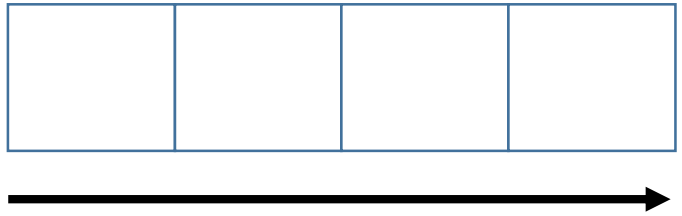
Do we place the bytes in these two ways?

AB CD EF 01



Memory address increasing

AB CD EF 01



Memory address increasing

# Little Endian and Big Endian

Do we place the bytes in these two ways?

AB CD EF 01 “Half-byte”?


Memory address increasing

One byte?

AB CD EF 01


Memory address increasing

**“Half-byte”: a nibble**

# Little Endian and Big Endian

Two possible directions to store a number in member space.

AB CD EF 01


Memory address increasing

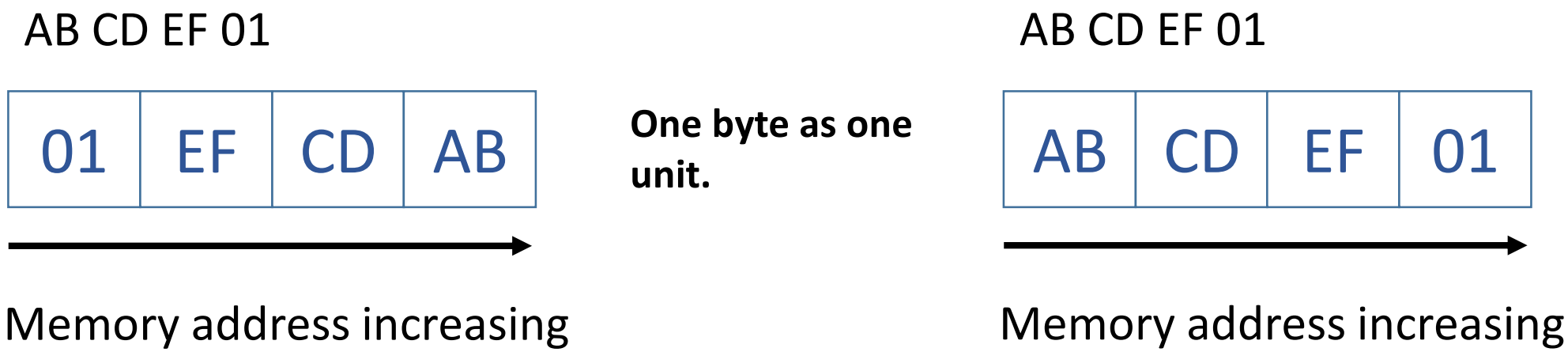
**One byte as one unit.**

AB CD EF 01


Memory address increasing

# Little Endian and Big Endian

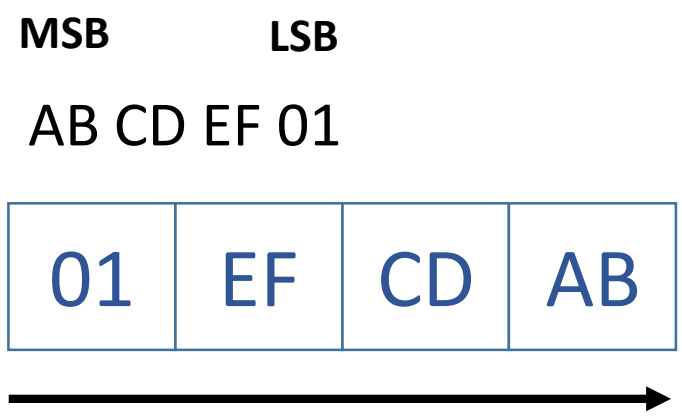
- Endianness: refer to the order of bytes within a binary representation of a number
- Big Endian ordering: place the most significant byte first and the least significant byte last.
- Little Endian ordering: place the least significant byte first and the most significant byte last.



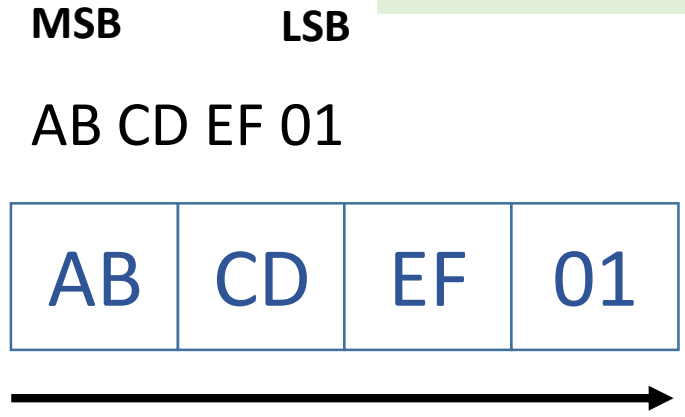
# Little Endian and Big Endian

- Endianness: refer to the order of bytes within a binary representation of a number
- Big Endian ordering: place the most significant byte first and the least significant byte last.
- Little Endian ordering: place the least significant byte first and the most significant byte last.

MOST Significant Byte (**MSB**)
LIST Significant Byte (**LSB**)

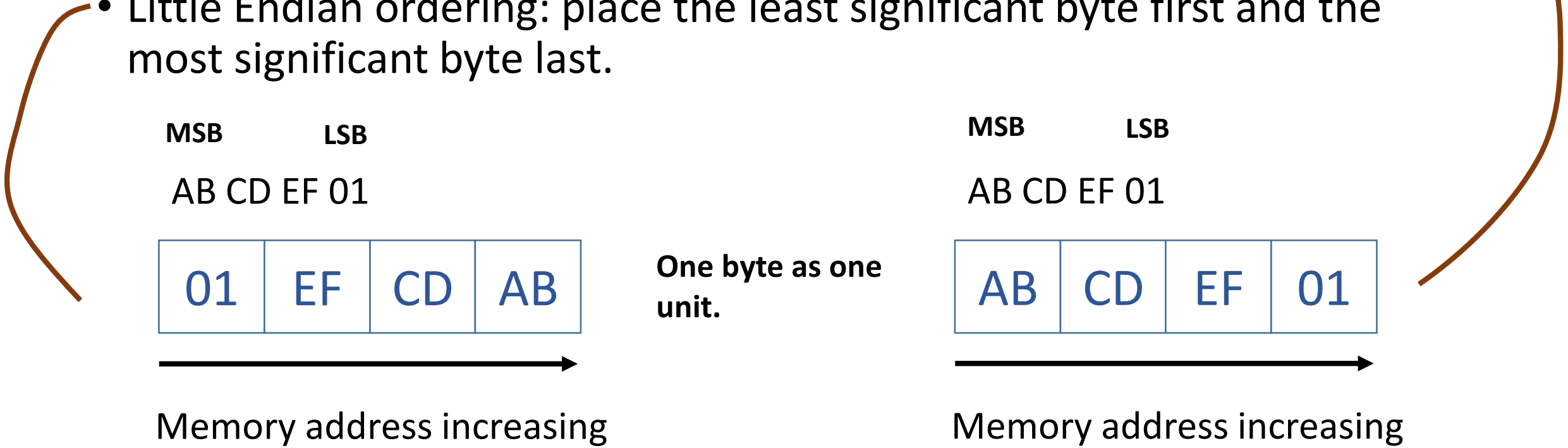



**One byte as one unit.**



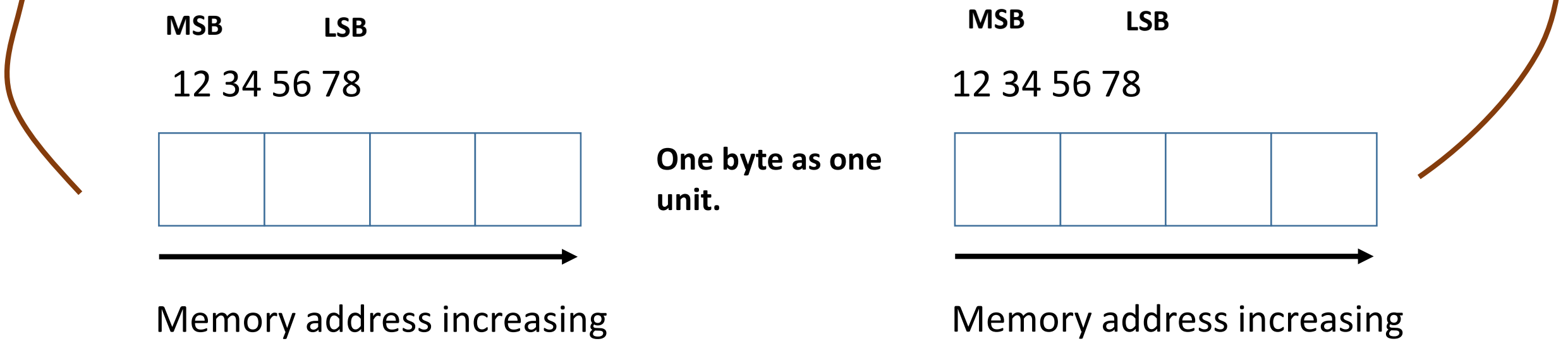
# Little Endian and Big Endian

- Endianness: refer to the order of bytes within a binary representation of a number
- Big Endian ordering: place the most significant byte first and the least significant byte last.
- Little Endian ordering: place the least significant byte first and the most significant byte last.



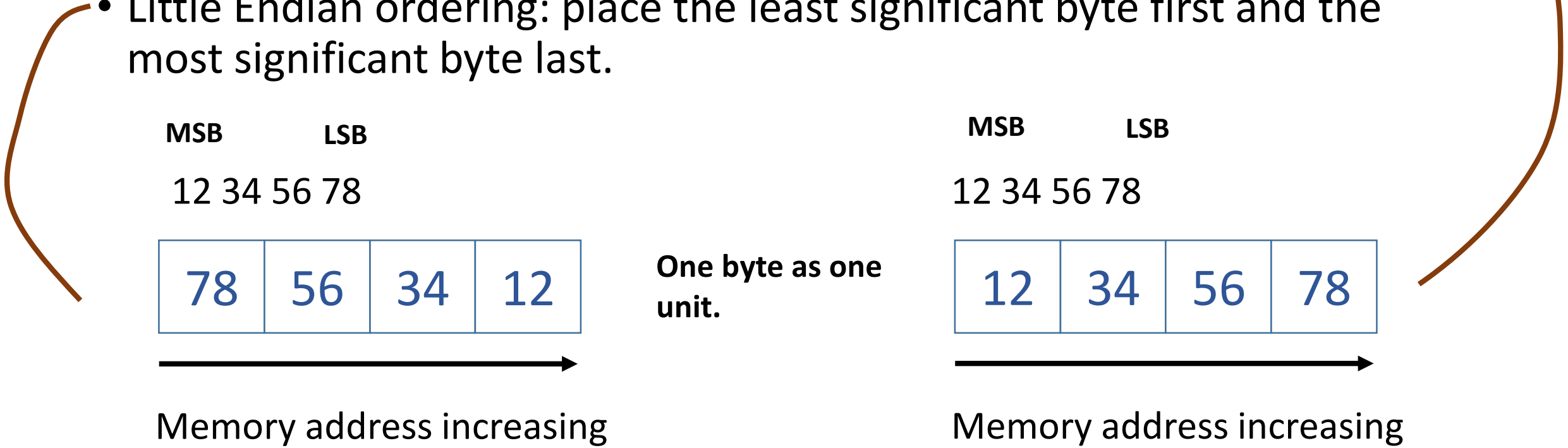
# Little Endian and Big Endian: Example

- Endianness: refer to the order of bytes within a binary representation of a number
- Big Endian ordering: place the most significant byte first and the least significant byte last.
- Little Endian ordering: place the least significant byte first and the most significant byte last.



# Little Endian and Big Endian: Example

- Endianness: refer to the order of bytes within a binary representation of a number
- Big Endian ordering: place the most significant byte first and the least significant byte last.
- Little Endian ordering: place the least significant byte first and the most significant byte last.



# Pointers

How do you do dynamic memory allocation?

# Pointer variables (Pointers)

**Usage: hold memory addresses as their values**.

# Pointer variables (Pointers)

**Usage: hold memory addresses as their values**.

A variable: contain a specific value, e.g., an integer, a floating-point value, and a character.

A pointer: contain the memory address of a variable

```
int count = 0x12345678;
short status = 0x02;
char letter = ‘B';

int*        pCount = &count;
short*      pStatus = &status;
char*       pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

# Pointer variables (Pointers)

**Usage: hold memory addresses as their values**.

A variable: contain a specific value, e.g., an integer, a floating-point value, and a character.

A pointer: contain the memory address of a variable

**Place the bytes starting at address:**
**0015FF80.**

**Adopt little endian ordering**

'B': 66 (ASCII): 0x42

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

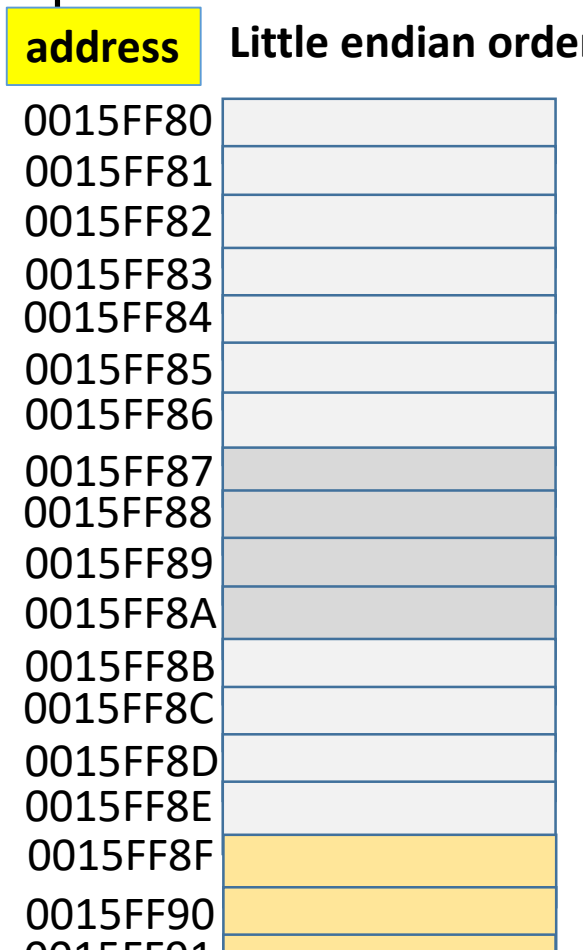
*: dereference operator
*pCount means the value pointed by pCount

# Pointer variables (Pointers)

**Usage: hold memory addresses as their values**.

A variable: contain a specific value, e.g., an integer, a floating-point value, and a character.

A pointer: contain the memory address of a variable




**Place the bytes starting at address: 0015FF80.**

**Adopt little endian ordering**

'B': 66 (ASCII): 0x42

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

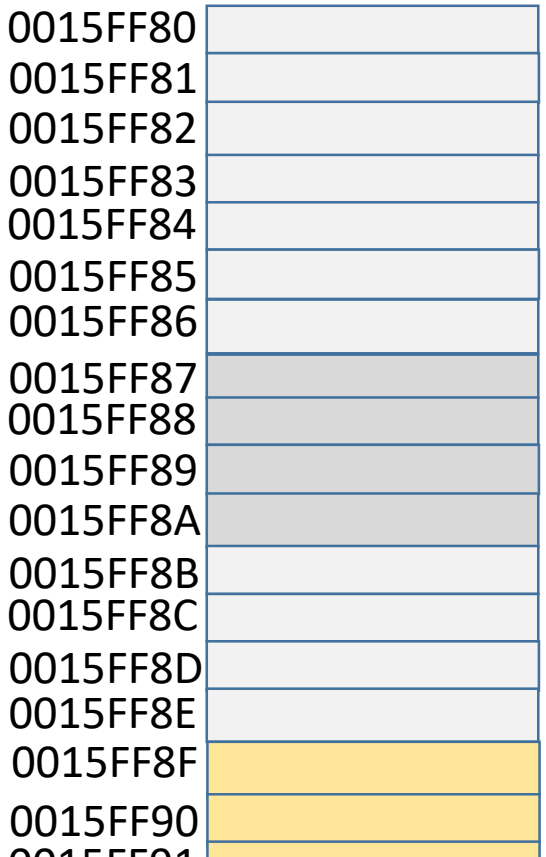
# Pointer variables (Pointers)

**Usage: hold memory addresses as their values**.

A variable: contain a specific value, e.g., an integer, a floating-point value, and a character.

A pointer: contain the memory address of a variable

**Little endian ordering**




```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

'B': 66 (ASCII): 0x42

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

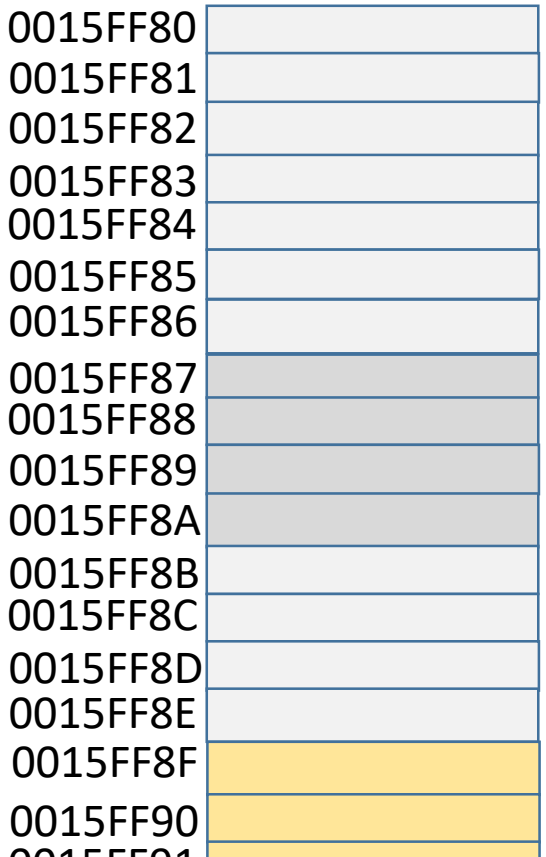
# Pointer variables (Pointers)

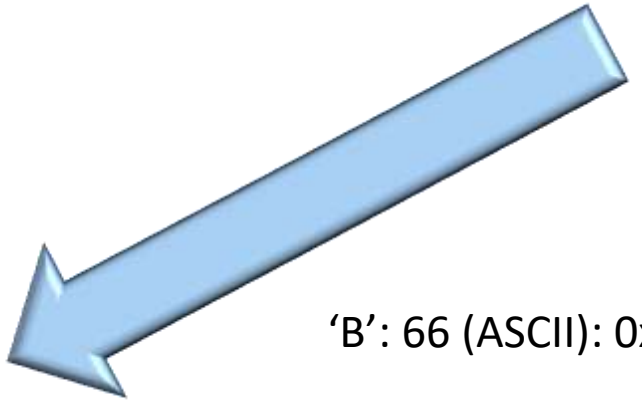
**Usage: hold memory addresses as their values**.

A variable: contain a specific value, e.g., an integer, a floating-point value, and a character.

A pointer: contain the memory address of a variable

**Little endian ordering**



‘B’: 66 (ASCII): 0x42

- How do you store the values of the variables in memory space?
- What is the memory layout for the variables?

```
int count = 0x12345678;
short status = 0x02;
char letter = ‘B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

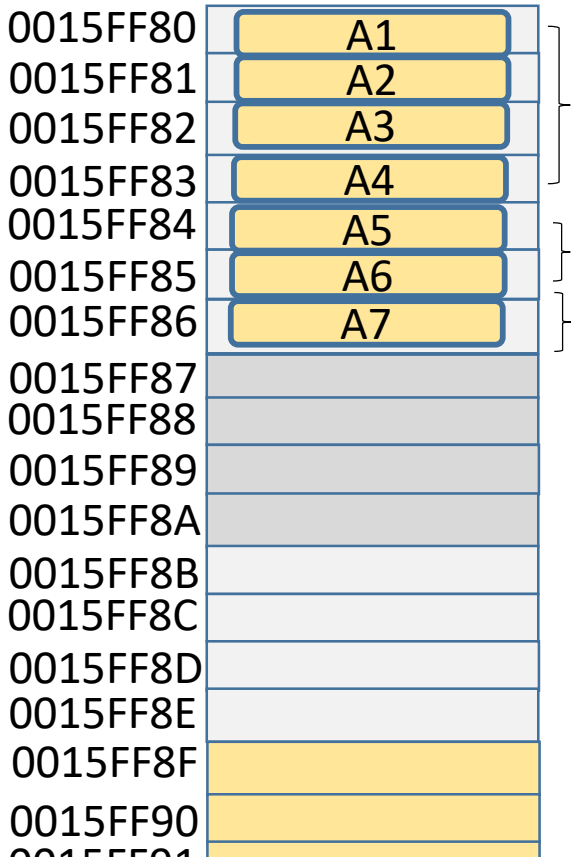
# Pointer variables (Pointers)

**Usage: hold memory addresses as their values**.

A variable: contain a specific value, e.g., an integer, a floating-point value, and a character.

A pointer: contain the memory address of a variable

**Little endian ordering**




count (4 bytes)

status(2 bytes)

letter(1 byte)  'B': 66 (ASCII): 0x42

- How do you store the values of the variables in memory space?
- What is the memory layout for the variables?

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

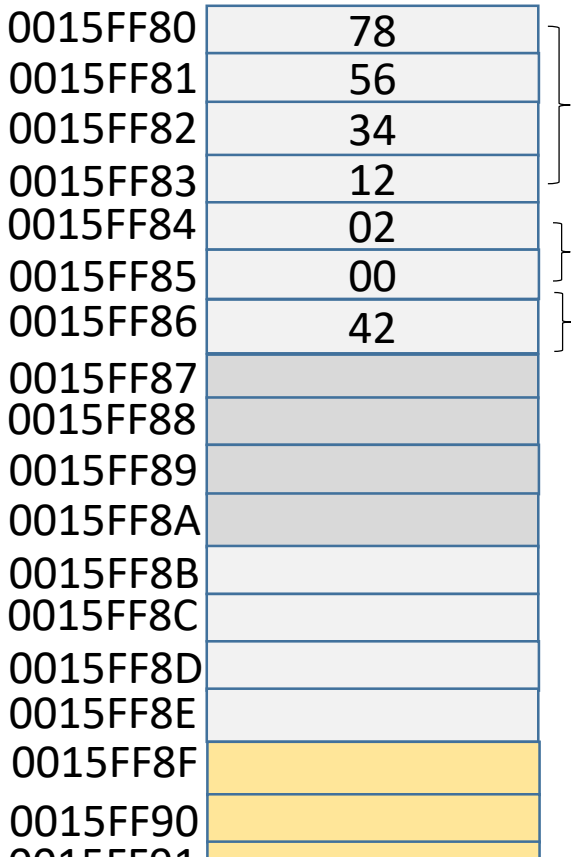
# Pointer variables (Pointers)

**Usage: hold memory addresses as their values**.

A variable: contain a specific value, e.g., an integer, a floating-point value, and a character.

A pointer: contain the memory address of a variable

**Little endian ordering**




count (4 bytes)

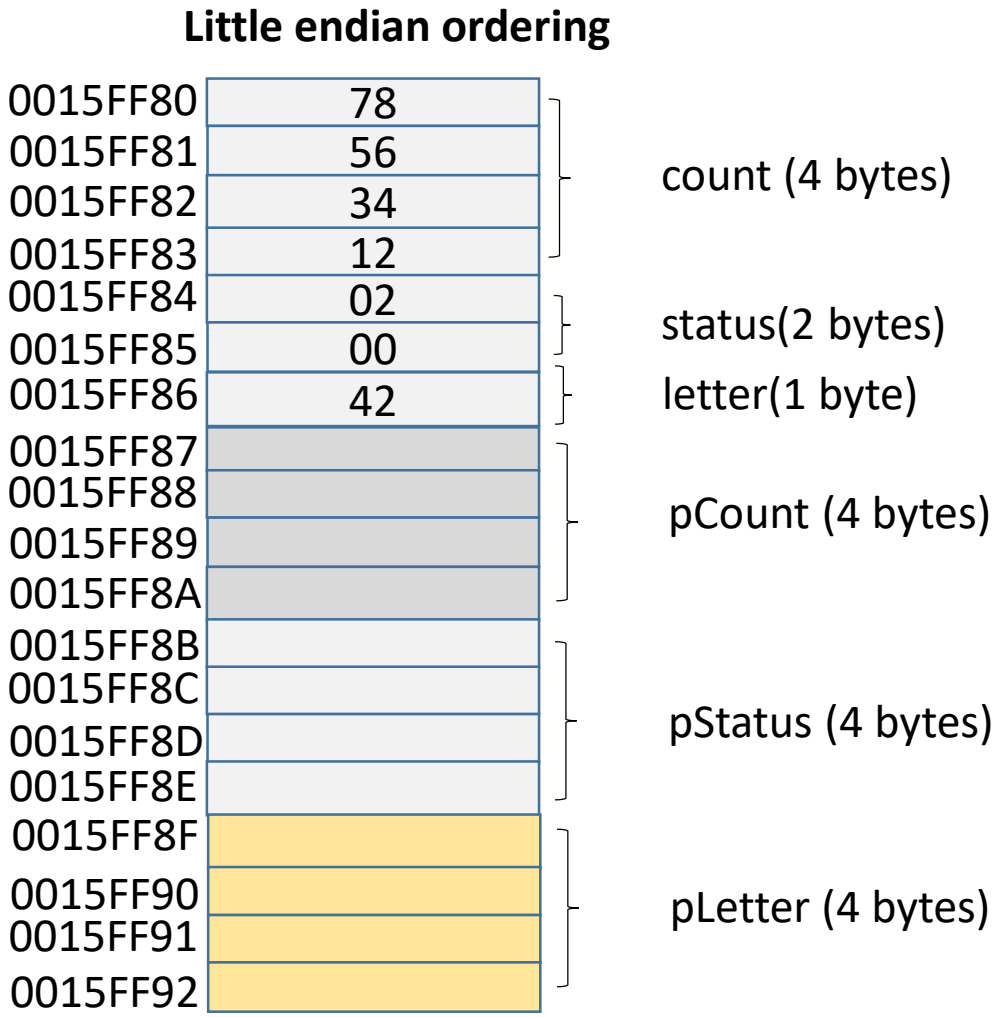
status(2 bytes)

letter(1 byte)

'B': 66 (ASCII): 0x42

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount
```

# Pointer variables (Pointers)

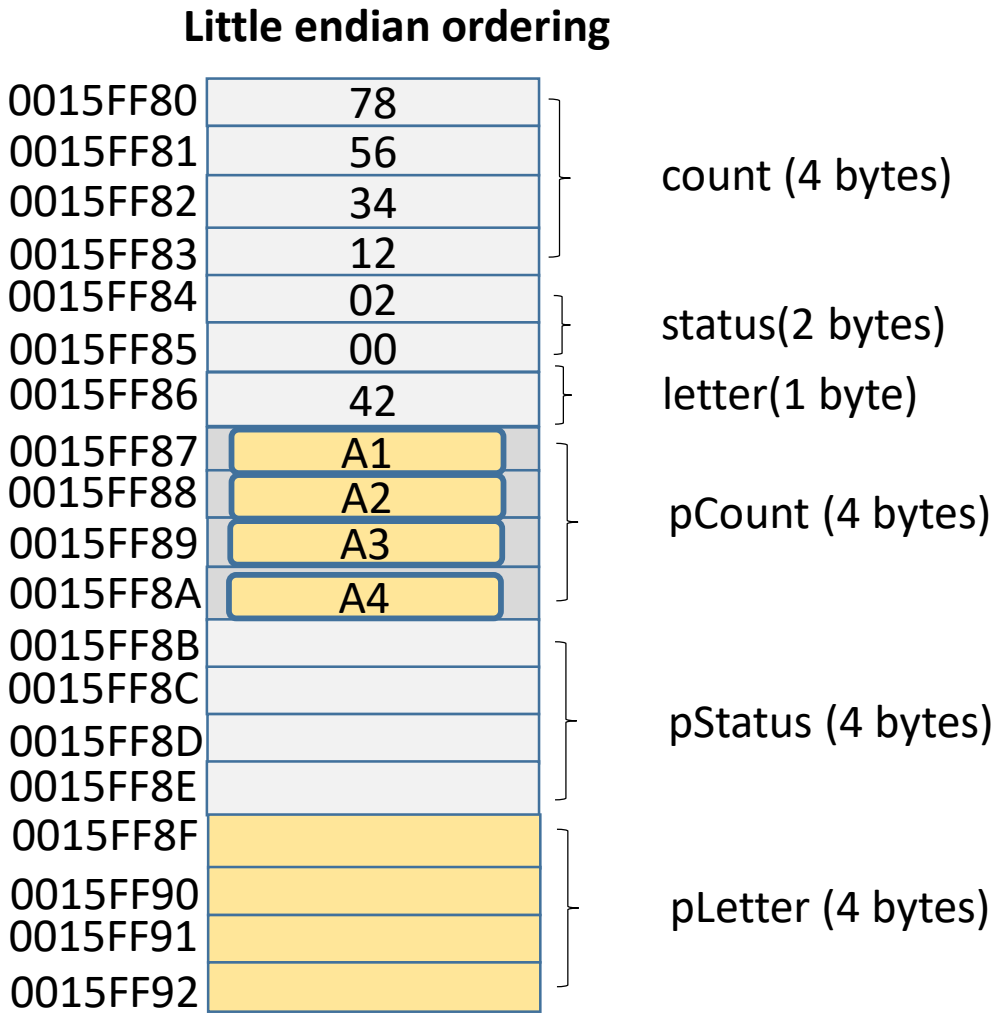



```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

# Pointer variables (Pointers)

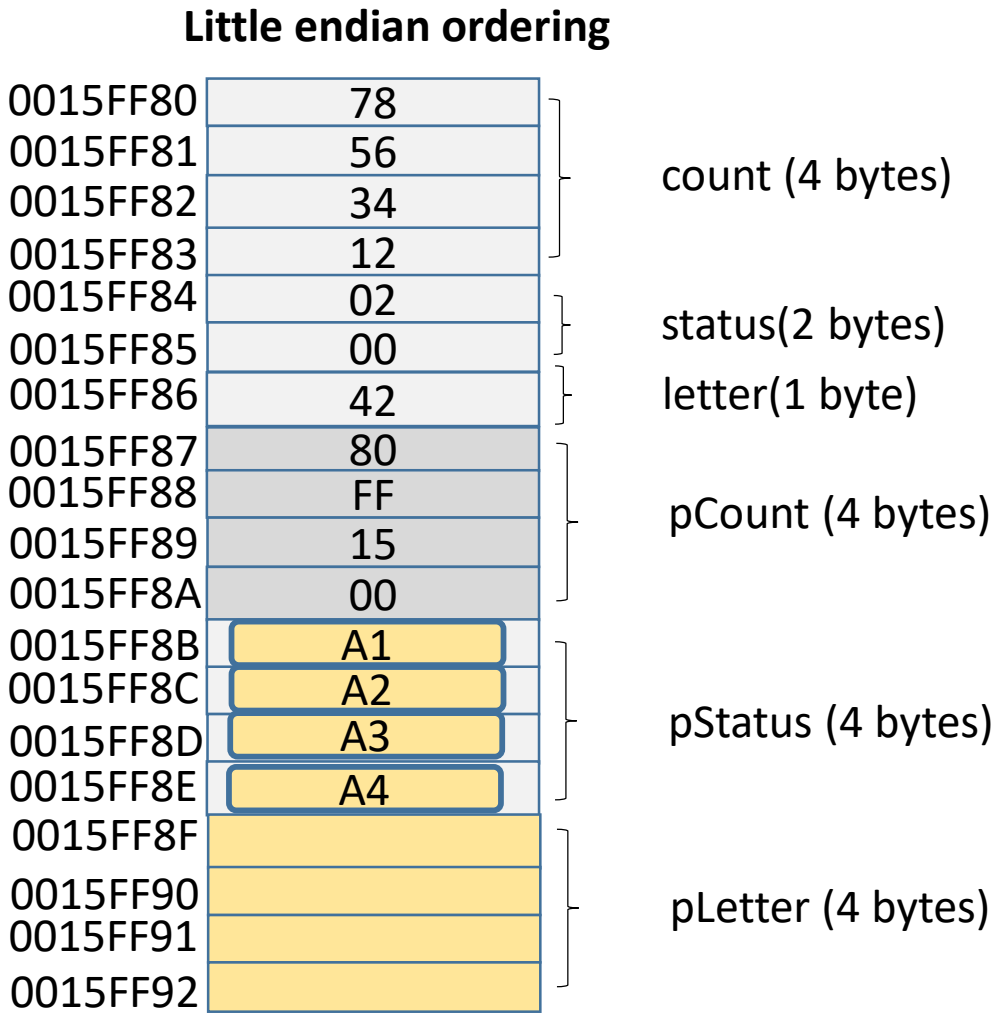



```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

# Pointer variables (Pointers)

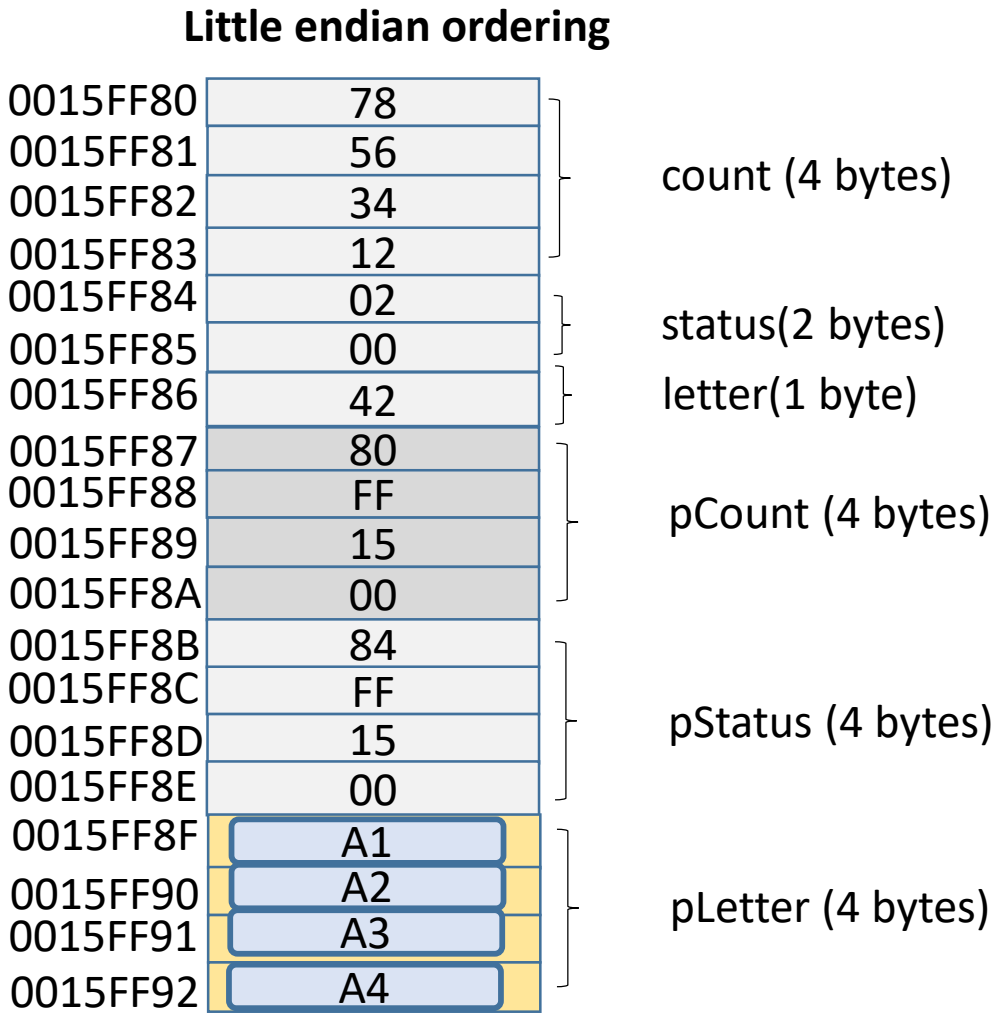



```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

# Pointer variables (Pointers)



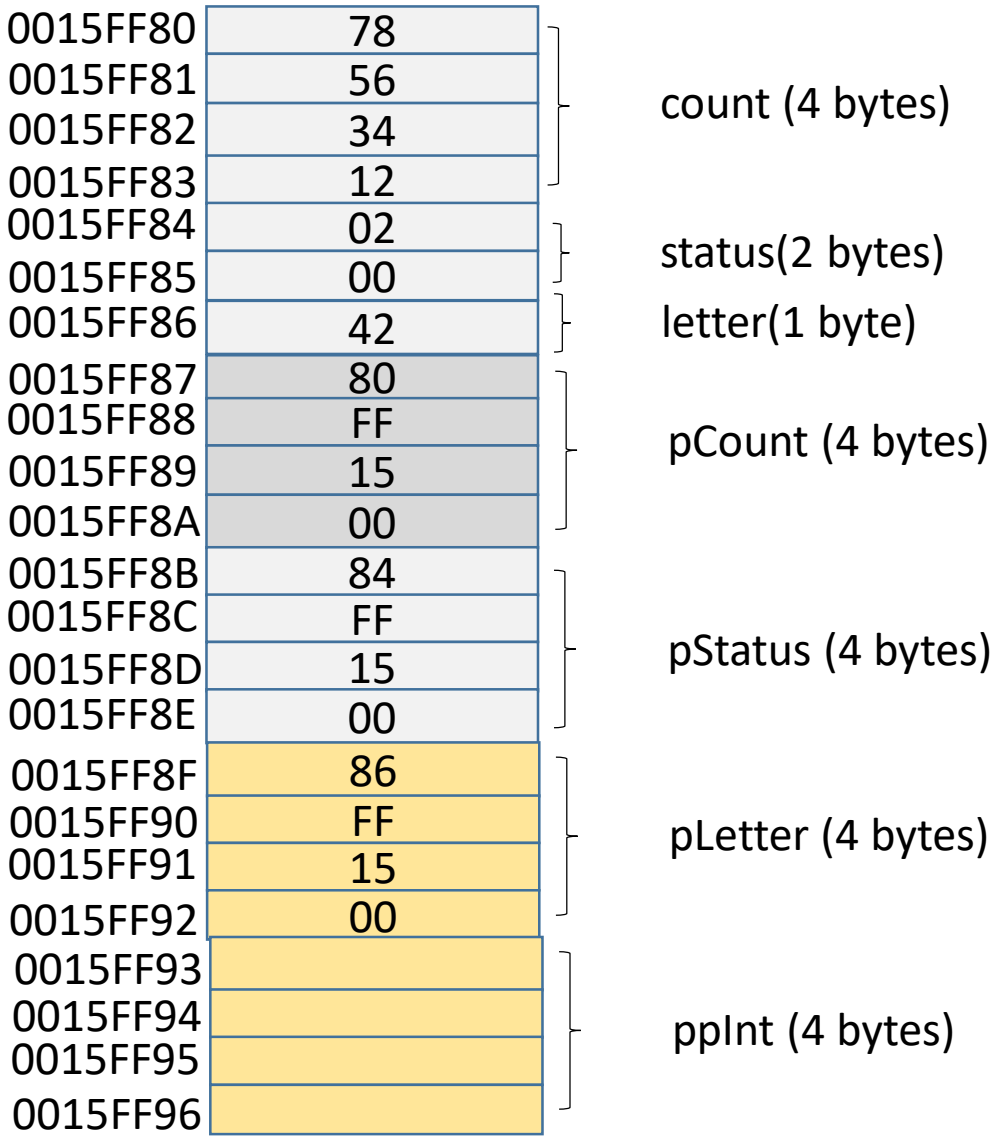

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

pCount = &count;
```

&: address operator
&count means the address of count

*: dereference operator
*pCount means the value pointed by pCount

# Pointer variables (Pointers)

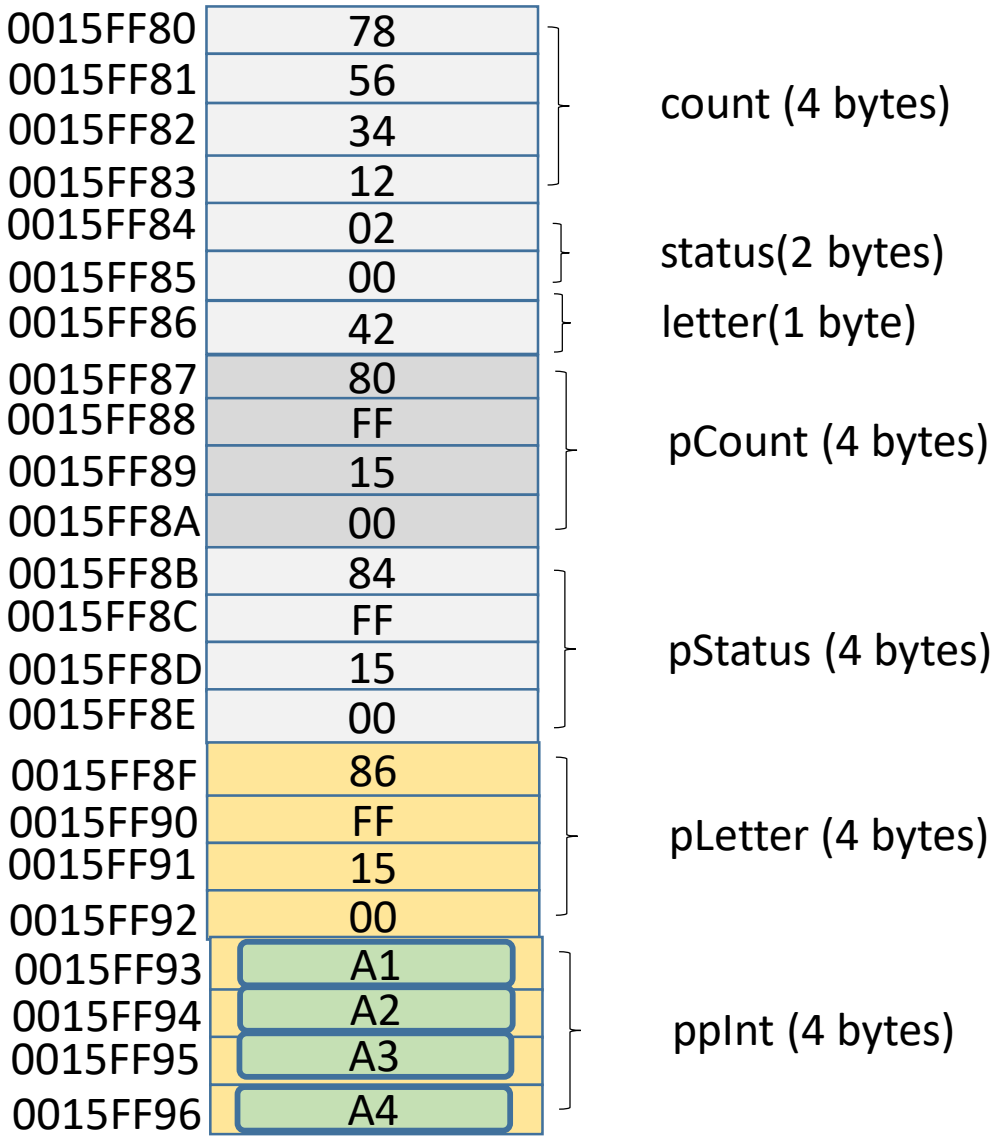
**Little endian ordering**




```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

int **ppInt = &pCount;
```

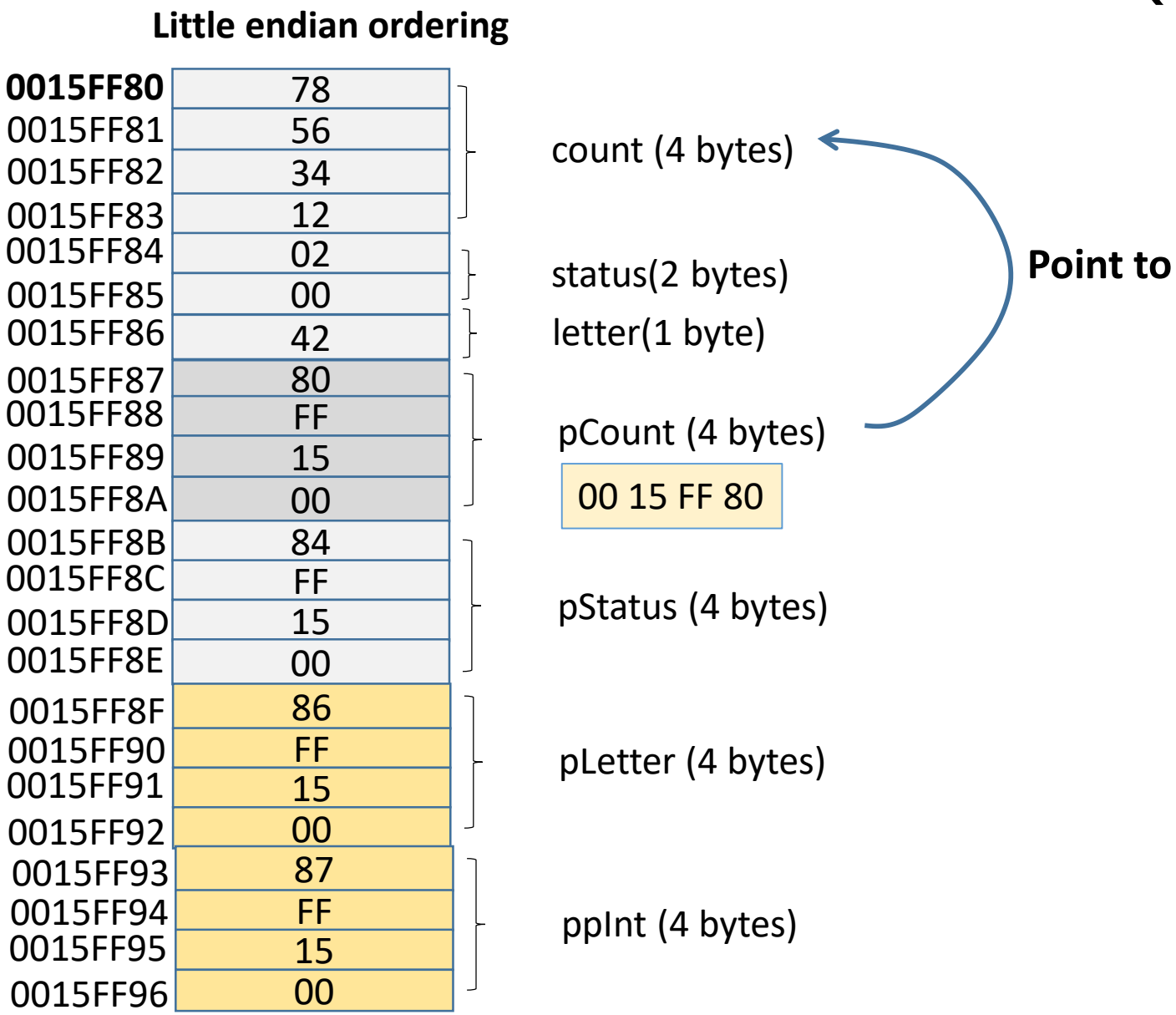
# Pointer variables (Pointers)

**Little endian ordering**



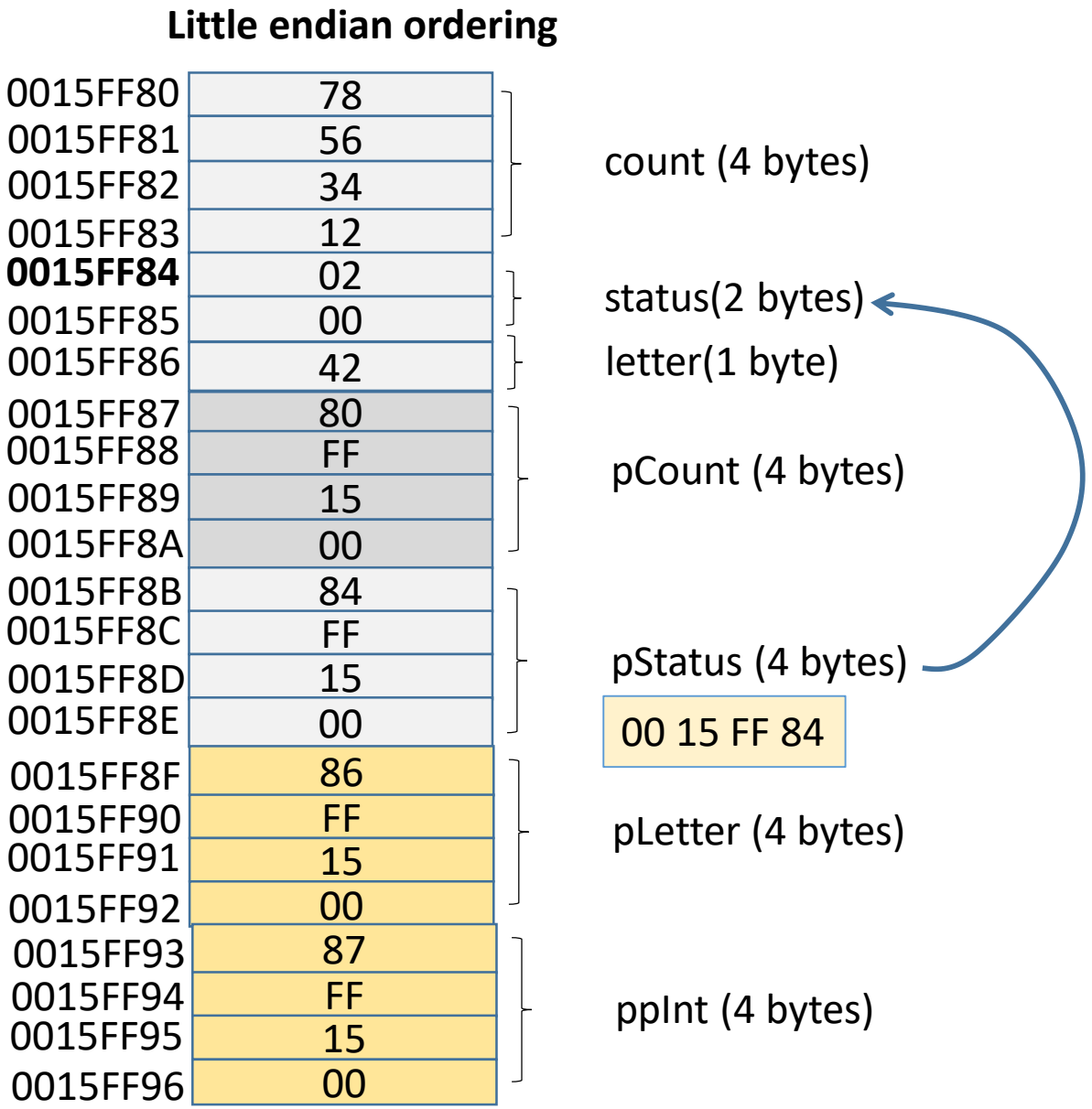

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

int **ppInt = &pCount;
```

# Pointer variables (Pointers)



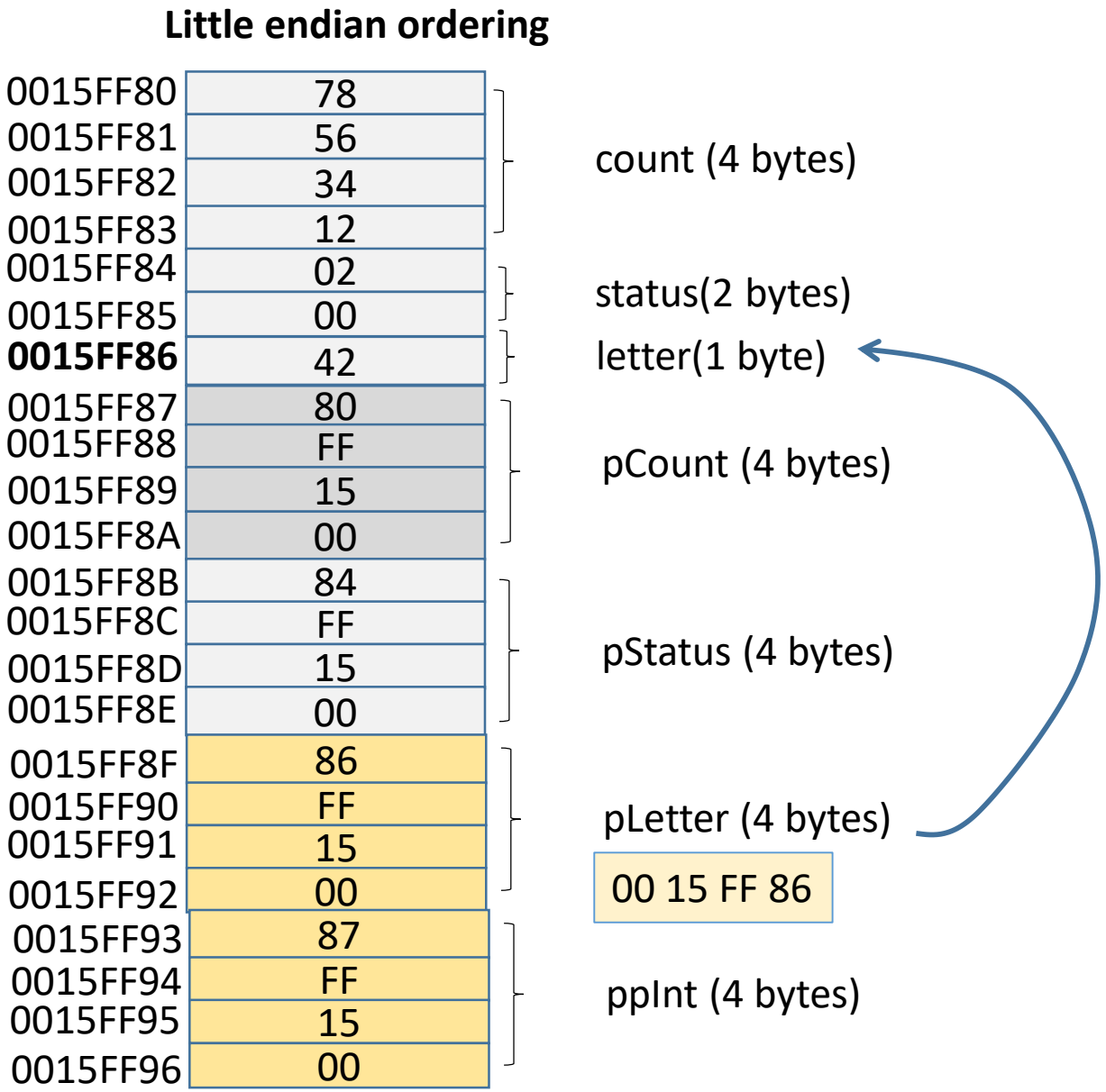

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

int **ppInt = &pCount;
```

# Pointer variables (Pointers)



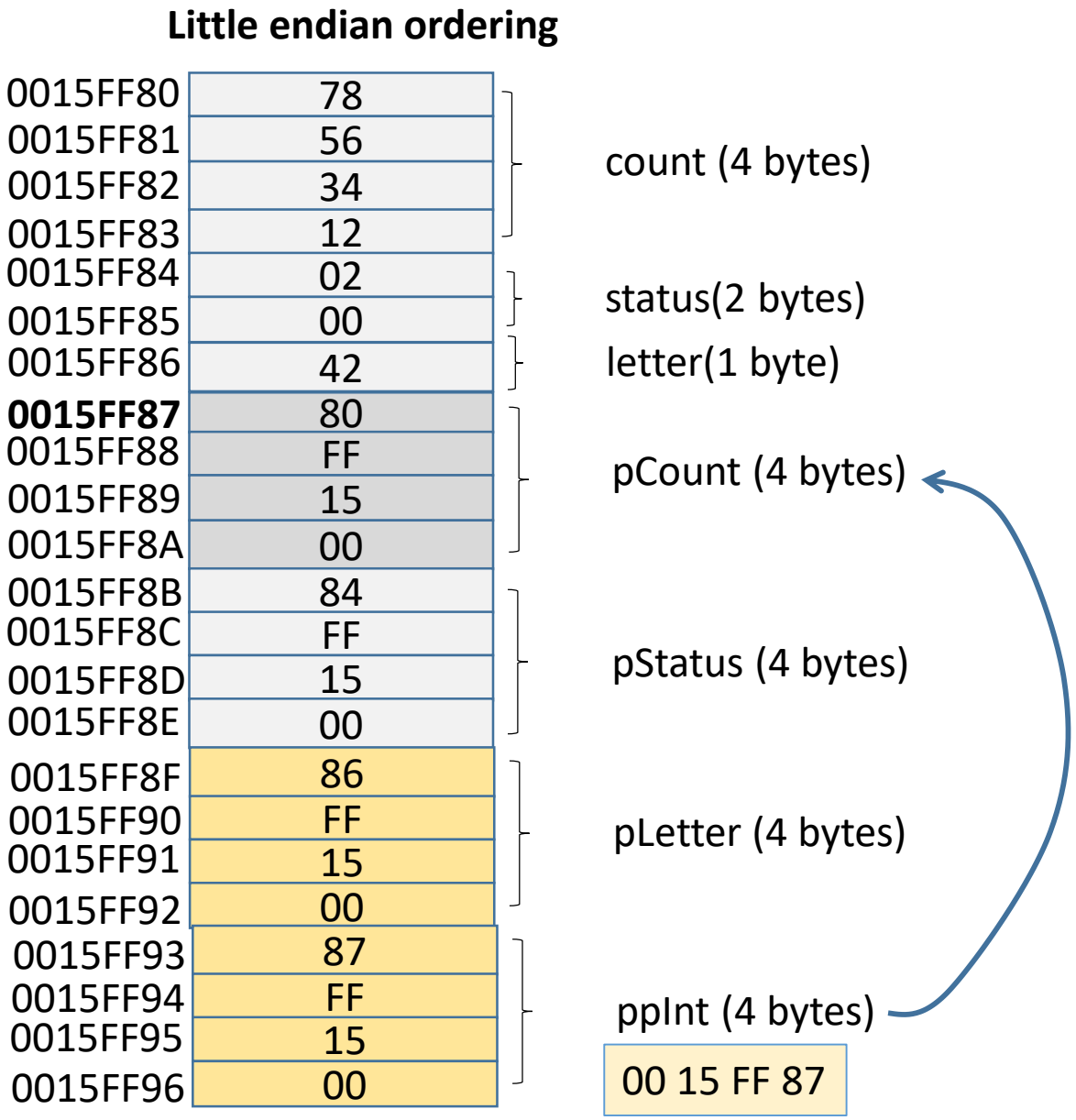

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

int **ppInt = &pCount;
```

# Pointer variables (Pointers)



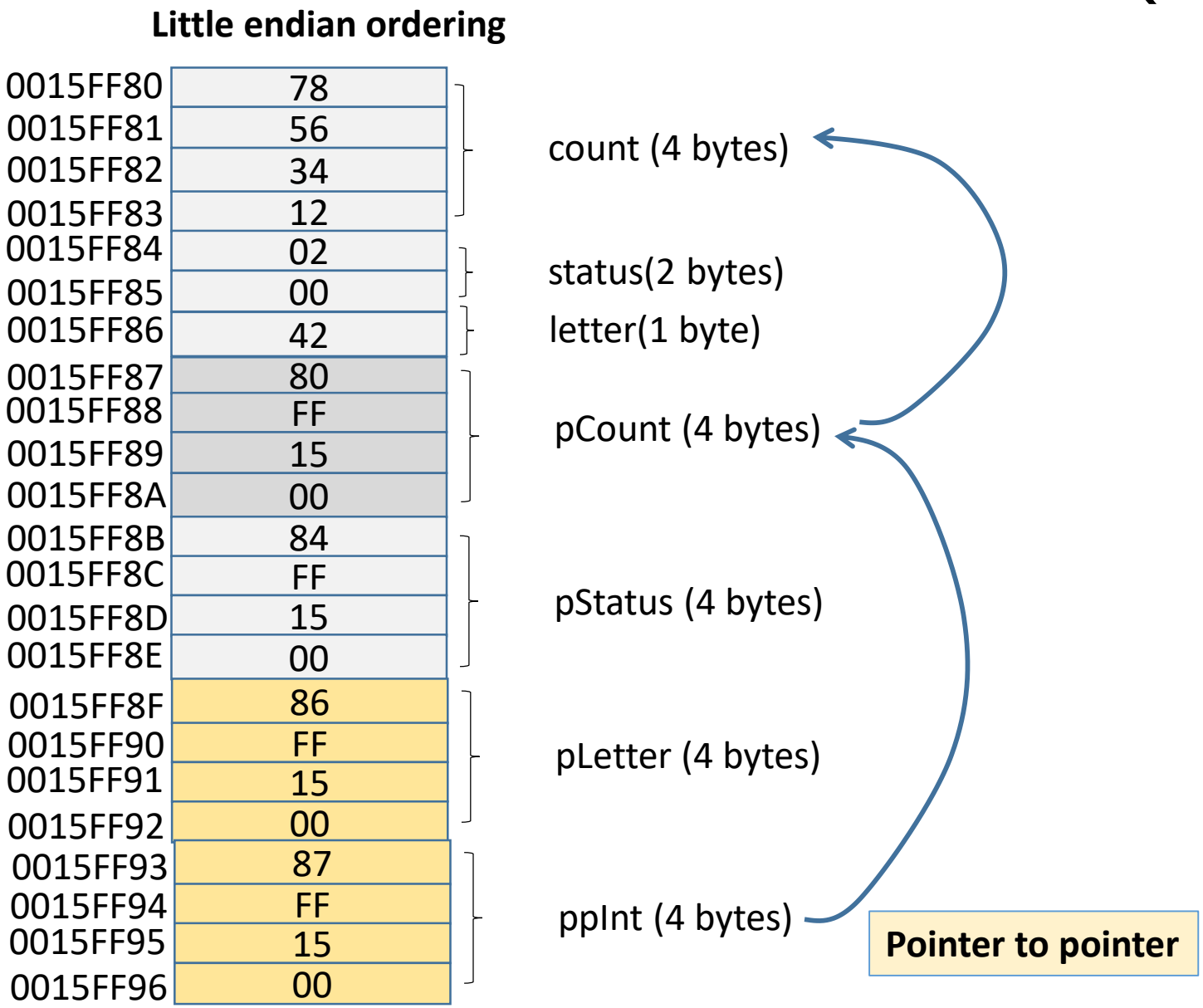

```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

int **ppInt = &pCount;
```

# Pointer variables (Pointers)




```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

int **ppInt = &pCount;
```

# Pointer variables (Pointers)




```
int count = 0x12345678;
short status = 0x02;
char letter = 'B';

int*    pCount = &count;
short*  pStatus = &status;
char*   pLetter = &letter;

//pointer to pointer (double pointer)
int **ppInt = &pCount;
```

*ppInt := pCount

**ppInt := count

**ppInt = 5;  count = 5;

# Pointers
# Declaration, initialization and assignment

```
DataType *ptr;

Example:

int *ptrInteger;
double *ptrDouble;
int *abc;
char *xyz;
```

```
int a, b;                          //declaration
double d;
int *ptrInteger = &a;              //assign the address of a to the pointer
double *ptrDouble = &d; // declaration and initialization
int *abc;                          // A1
abc = &b;                          // A2
```

& ampersand  * an asterisk

# Pointers
# Declaration, initialization and assignment

```
DataType *ptr;

Example:

int *ptrInteger;
double *ptrDouble;
int *abc;
char *xyz;
```

```
int a, b;                       //declaration
double d;
int *ptrInteger = &a;           //assign the address of a to the pointer
double *ptrDouble = &d; // declaration and initialization
int *abc;                       // declaration
abc = &b;                       // assignment
```

& ampersand  * an asterisk

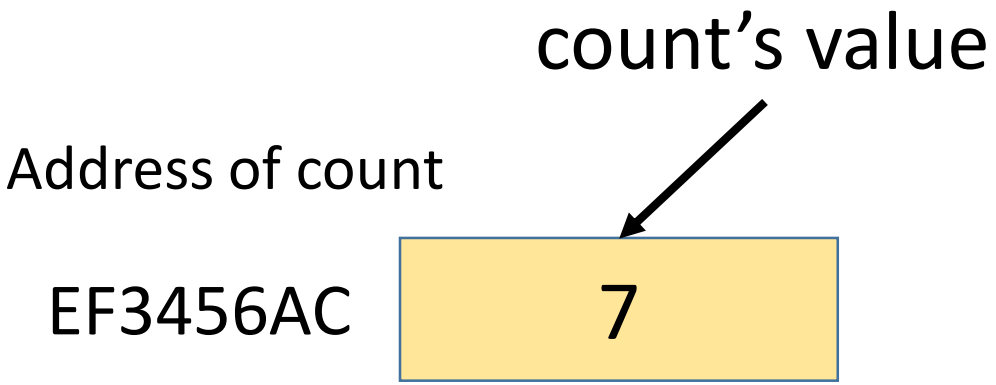
# Pointers

dataType* pVarName;  // declaration

int count = 7;

…

**int*** pCount = &count; // a pointer points to an integer



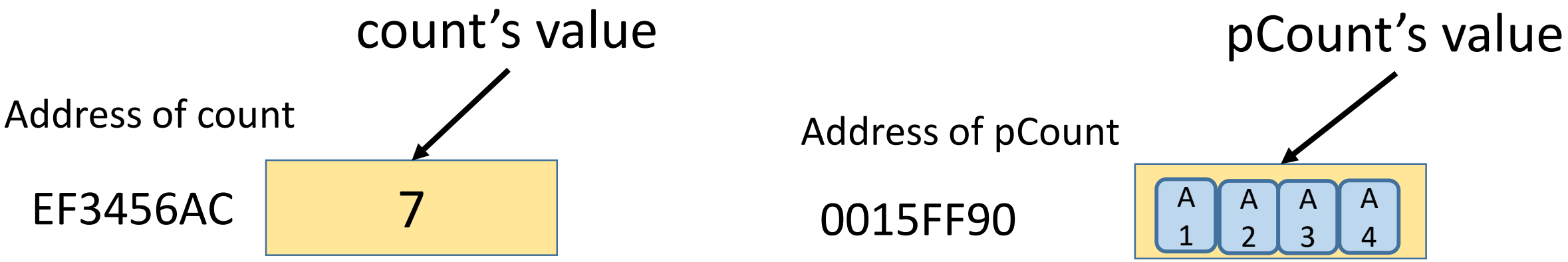
# Pointers

dataType* pVarName;  // declaration

int count = 7;

…

**int*** pCount = &count; // a pointer points to an integer



# Dereferencing

Indirection: Referencing a value from a pointer

*pointer //

# Dereferencing

Indirection: Referencing a value from a pointer

```
*pointer          // indirection; indirect reference
```

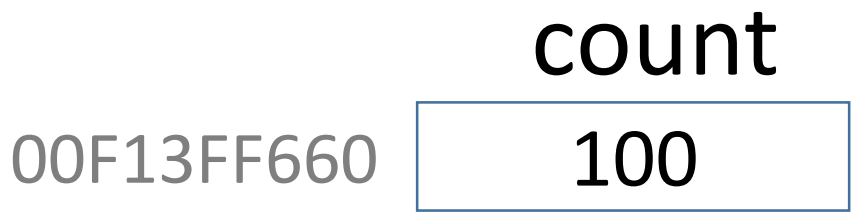
```
int count = 100;
count++;          // direct reference
                  // increment the value in count by 1

int *pCount = &count;
(*pCount)++;  // indirect reference
                  // the value in the memory pointed by pCount is
                  // incremented by 1
```

*pCount is an alias of count.

# Dereferencing

Indirection: Referencing a value from a pointer

```
*pointer        // indirection; indirect reference
```



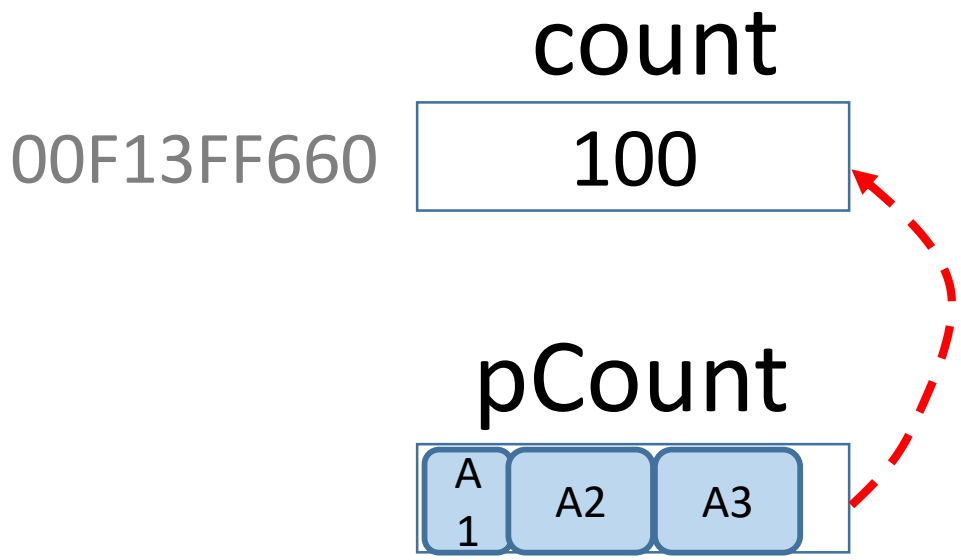

```
int count = 100;
count++;        // direct reference
                // increment the value in count by 1

int *pCount = &count;
(*pCount)++;  // indirect reference
                // the value in the memory pointed by pCount is
                // incremented by 1
```

*pCount is an alias of count.

# Dereferencing

Indirection: Referencing a value from a pointer

```
*pointer        // indirection; indirect reference
```




```
int count = 100;
count++;        // direct reference
                // increment the value in count by 1

int *pCount = &count;
(*pCount)++;  // indirect reference
                // the value in the memory pointed by pCount is
                // incremented by 1
```

*pCount is an alias of count.

# Pointers

The address of the variable of the same type can be assigned.

**int** area = 1;

**int*** pIntArea = &area**;** **//ok. Same type**

**double*** pArea = &area; // Not same type. Syntax error

# Pointers

The address of the variable of the same type can be assigned.

```
int area = 1;

int* pIntArea = &area;        //ok. Same type

double* pArea = &area; // Not same type. Syntax error
```

pArea is a pointer to a double number
&area is the address of an integer

# Pointer Initialization

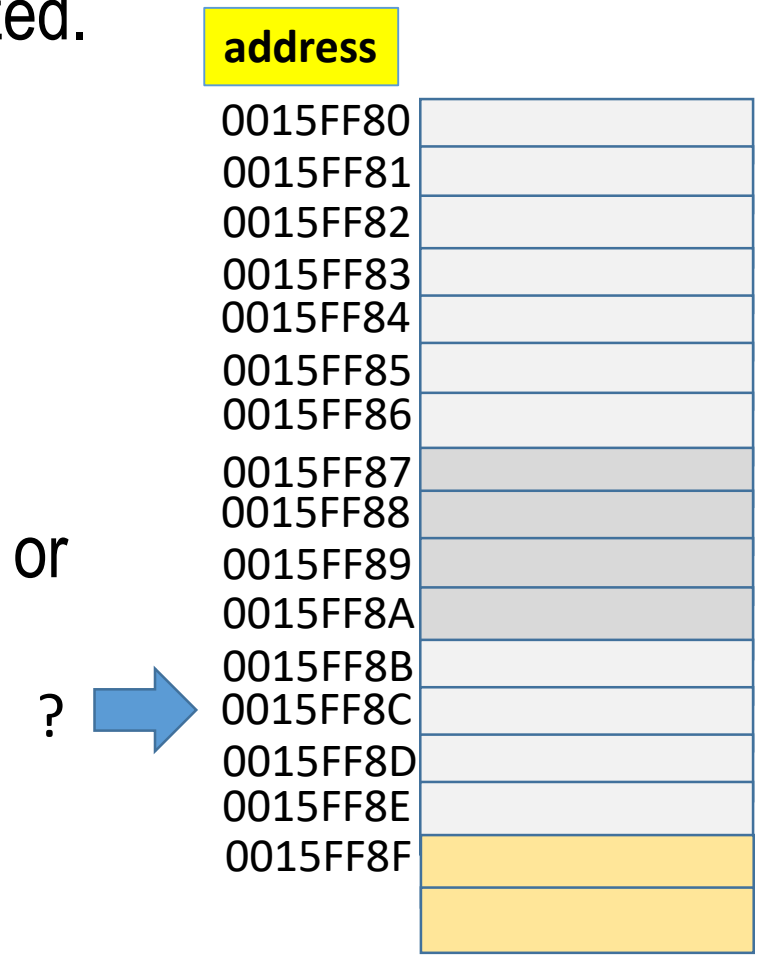
A local pointer contains an arbitrary value if it is not initialized.

A pointer with 0 value: Indicate that it points to nothing.

```
pInt = 0; // or   pInt = nullptr;
```

Dereferencing an uninitialized pointer leads to a fatal error or modifies data.

```
int *p;

*p = 1340;      // p is not initialized.

                // Where do we store 1340?
```



# Pointer Initialization

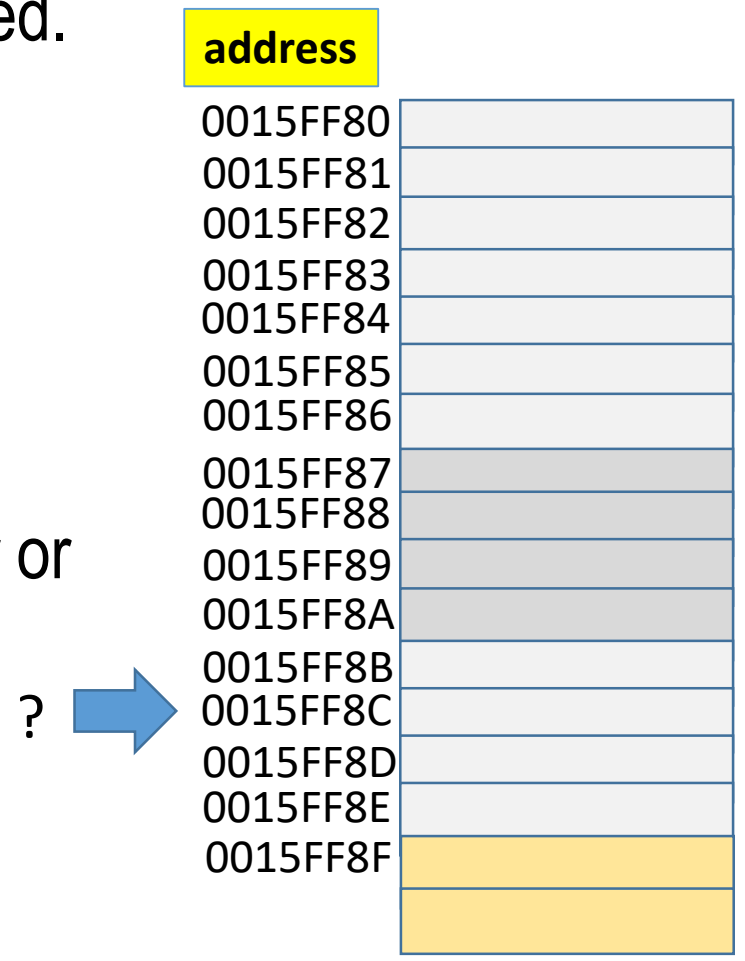
A local pointer contains an arbitrary value if it is not initialized.

A pointer with 0 value: Indicate that it points to nothing.

```
pInt = 0; // or   pInt = nullptr;
```

Dereferencing an uninitialized pointer leads to a **fatal error** or **modifies data**.

```
int *p;

*p = 1340;      // p is not initialized.

                // Where do we store 1340?
```



# Common problems

Declare two variables on the same line.

```
int i = 0, j = 1;
```

What are pI and pJ in the following declarations?

```
int* pI, pJ;

int *pI, pJ;    // What are data types of pI and pJ?
```

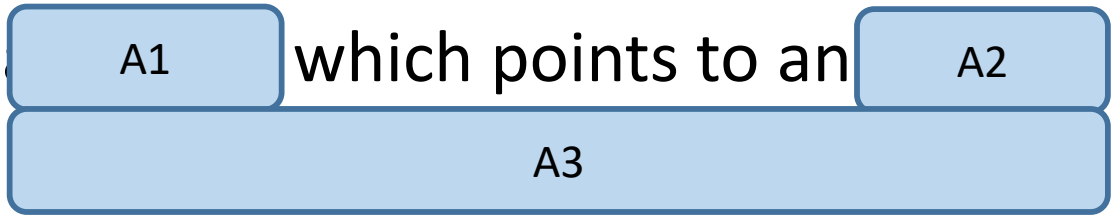
# Common problems

Declare two variables on the same line.

```
int i = 0, j = 1;
```

What are pI and pJ in the following declarations?

```
int* pI, pJ;

int *pI, pJ;    // What are data types of pI and pJ?
                // pI is a
                // pJ is
```



# typedef

Use typedef to define a synonymous type

```
typedef  existingType newType;
```

Example:

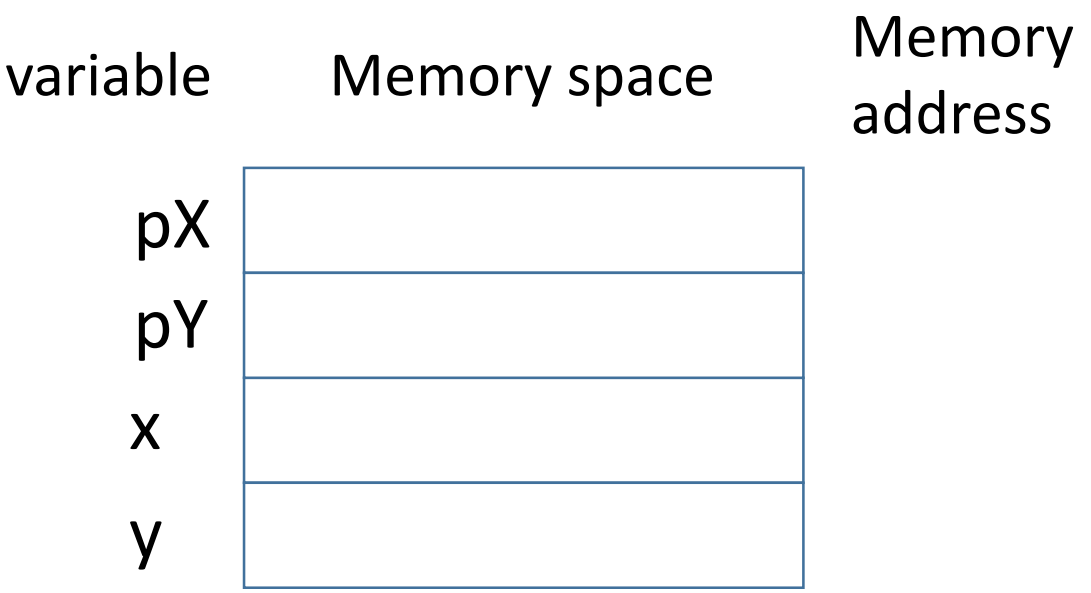
```
typedef  int  integer;
typedef  int* intPointer;
```

integer value = 40;
int *b1, b2;  // b1 is a pointer. b2 is A1
intPointer p1, p2;  // Both p1 and p2 are A2

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
pX = pY;        // what is the meaning?

*pX = *pY;      // what is the meaning?
```

# Assignment instructions




```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
pX = pY;        // what is the meaning?

*pX = *pY;      // what is the meaning?
```

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
pX = pY;        // what is the meaning?

*pX = *pY;      // what is the meaning?
```

| variable | Memory space | Memory address |
|---|---|---|
| pX | ?????? | |
| pY | ?????? | |
| x | 5 | |
| y | 6 | |

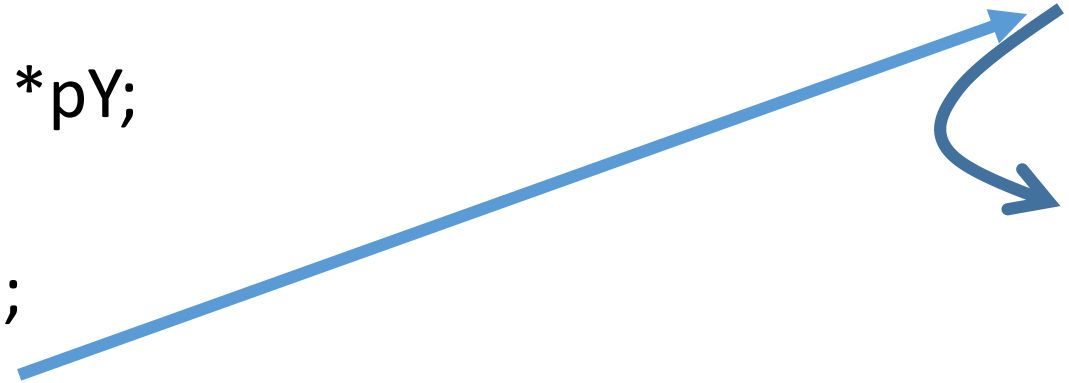
# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
pX = pY;        // what is the meaning?

*pX = *pY;
```

| variable | Memory space | Memory address |
|---|---|---|
| pX | ?????? | FF11A080 |
| pY | ?????? | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Assignment instructions

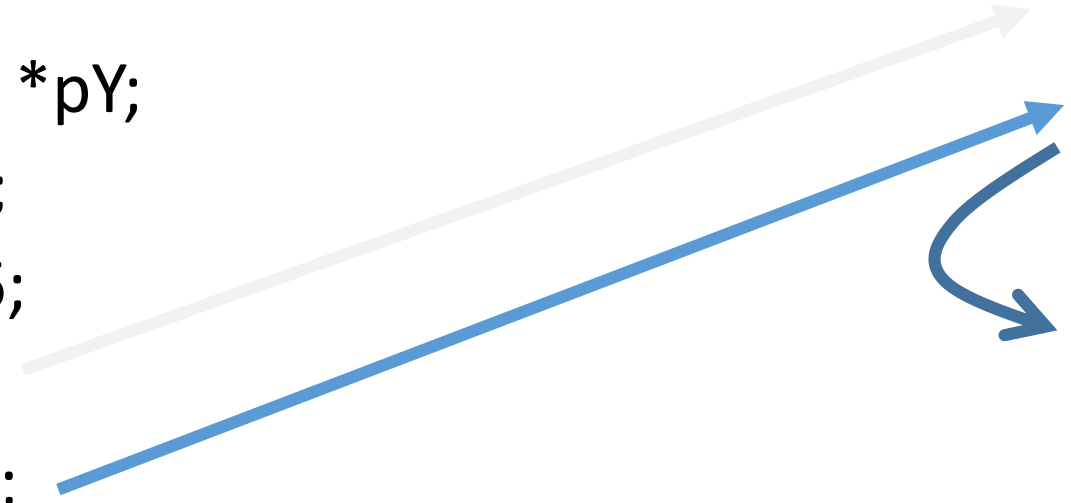
```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
pX = pY;        // what is the meaning?

*pX = *pY;
```



| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | ?????? | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Assignment instructions

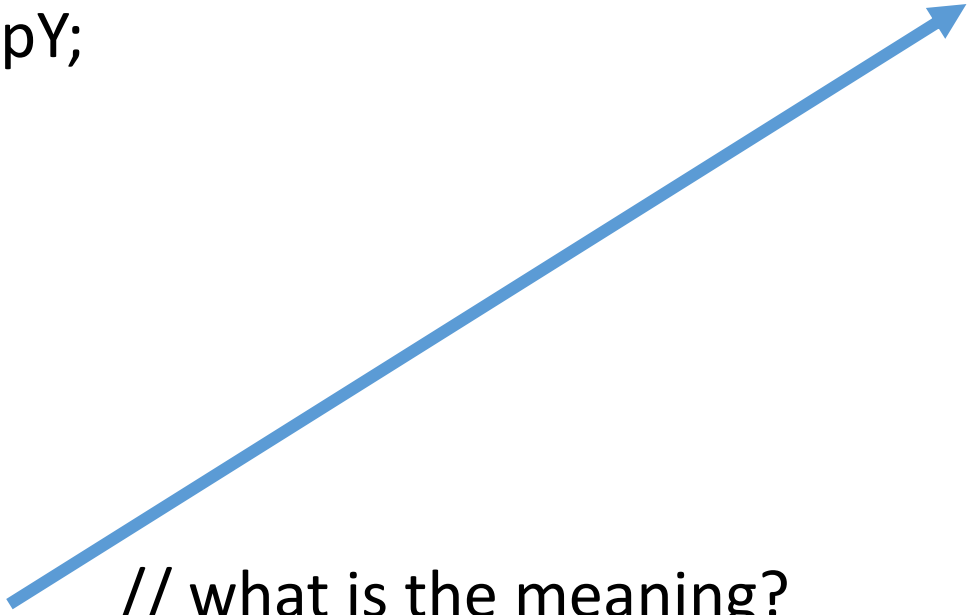
```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
pX = pY;        // what is the meaning?

*pX = *pY;
```



| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
...
pX = pY;        // what is the meaning?

*pX = *pY;
```

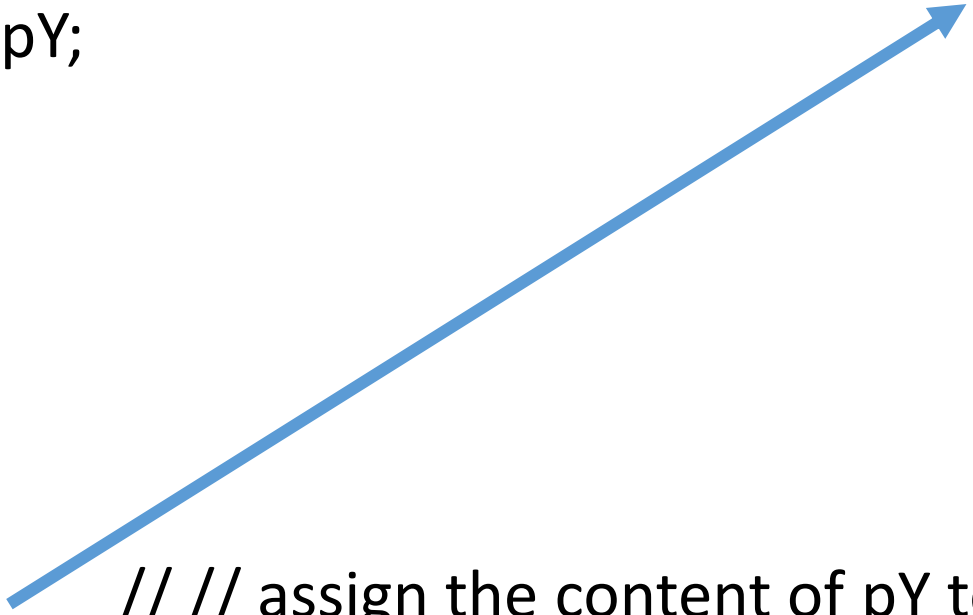


| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
pX = pY;        // // assign the content of pY to pX
```

*pX = *pY;



| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Assignment instructions

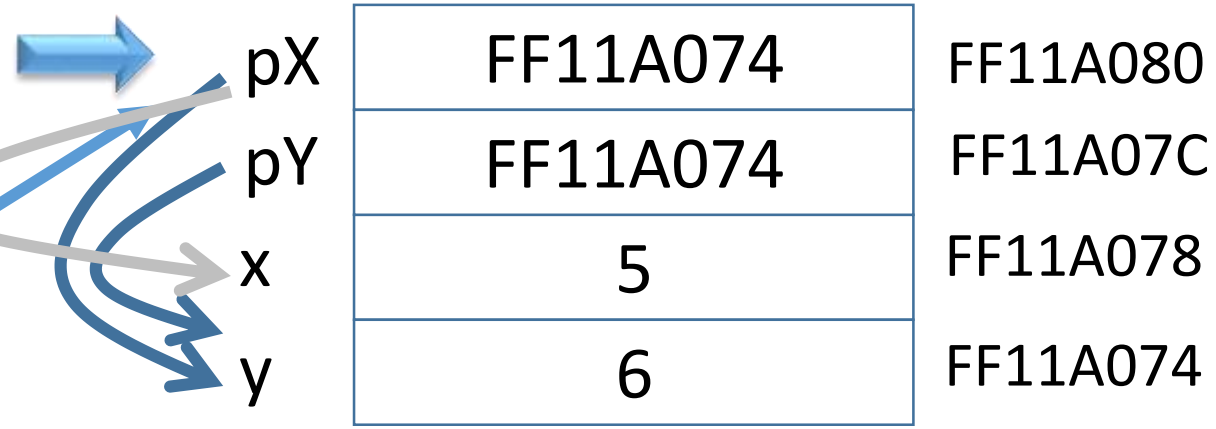
```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
pX = pY;        // // assign the content of pY to pX

*pX = *pY;
```

| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
...
pX = pY;        // // assign the content of pY to pX
```

*pX = *pY;



# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
```

*pX = *pY; // what is the purpose?

| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
```

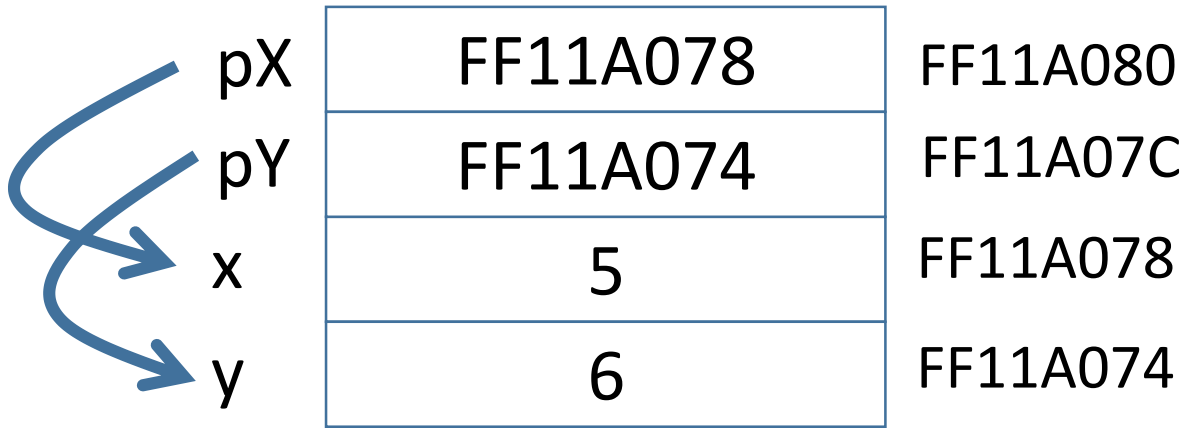
| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

```
*pX = *pY;
```

// dereferencing
//
// A1 the A2 of the variable pointed to by A3
// to the variable pointed to by A4

# Assignment instructions

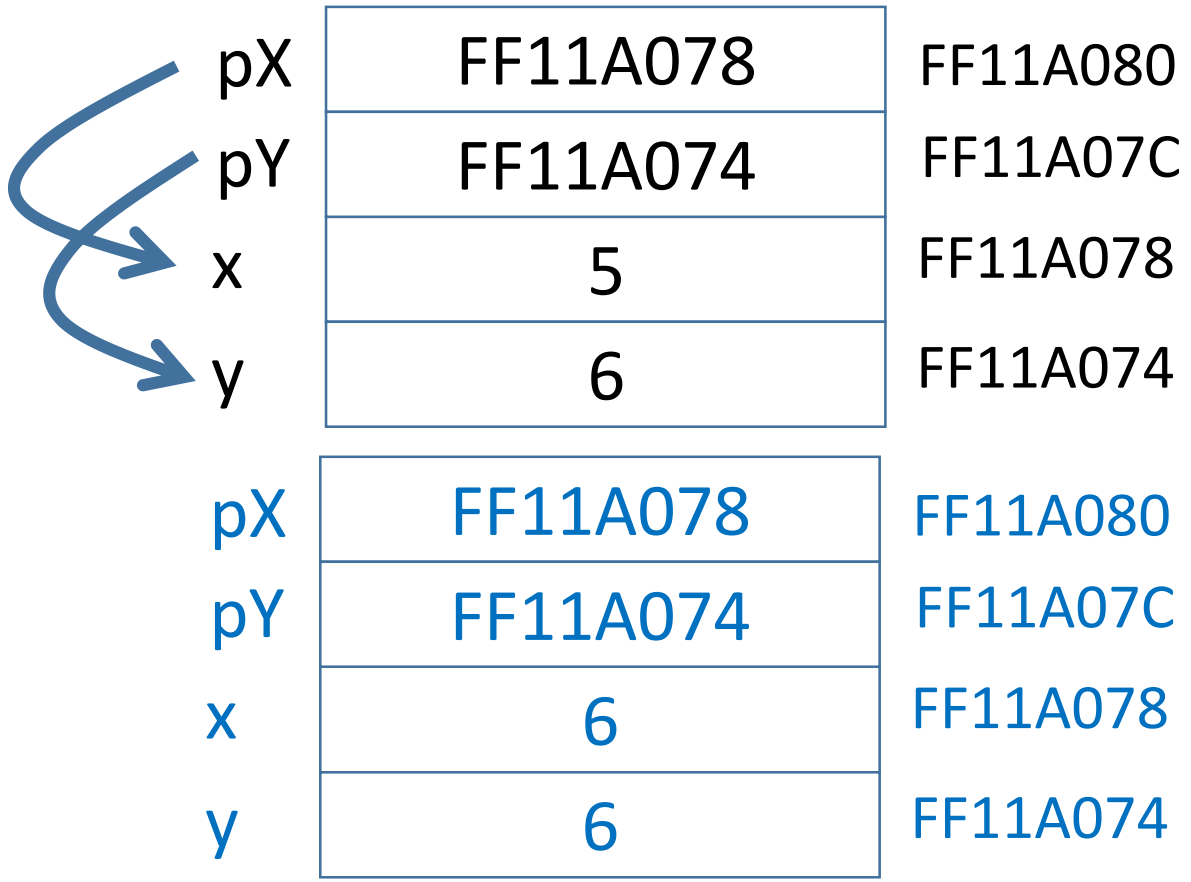
```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
```




```
*pX = *pY;      // dereferencing
                //
                // Assign the value of the variable pointed to by pY
                // to the variable pointed to by pX
```

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
…
```




```
*pX = *pY;    // dereferencing
              //
              // Assign the value of the variable pointed to by pY
              // to the variable pointed to by pX
```

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
...
```

*pX = *pY
1. *pY is the variable y
2. *pY gets the value of y
3. *pX is the variable x
4. Assign the value of y (*pY) to x (*pX)

| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

| | | |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 6 | FF11A078 |
| y | 6 | FF11A074 |

```
*pX = *pY;    // dereferencing
              //
              // Assign the value of the variable pointed to by pY
              // to the variable pointed to by pX
```

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
...
```

(*pX) = (*pY)

*pX = *pY

1. *pY is the variable y
2. *pY gets the value of y
3. *pX is the variable x
4. Assign the value of y (*pY) to x (*pX)

| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

| | | |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 6 | FF11A078 |
| y | 6 | FF11A074 |

```
*pX = *pY;    // dereferencing
              //
              // Assign the value of the variable pointed to by pY
              // to the variable pointed to by pX
```

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
...
```

(*pX) = (*pY)

*pX = *pY
1. *pY is the variable y
2. *pY gets the value of y
3. *pX is the variable x
4. Assign the value of y (*pY) to x (*pX)

| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

| | | |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 6 | FF11A078 |
| y | 6 | FF11A074 |

```
*pX = *pY;   // dereferencing
             //
             // Assign the value of the variable pointed to by pY
             // to the variable pointed to by pX
```

# Assignment instructions

```
int *pX, *pY;
int x =5;
int y = 6;
pX =&x;
pY = &y;
...
```

(*pX) = (*pY)

*pX = *pY
1. *pY is the variable y
2. *pY gets the value of y
3. *pX is the variable x
4. Assign the value of y (*pY) to x (*pX)

| variable | Memory space | Memory address |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

| | | |
|---|---|---|
| pX | FF11A078 | FF11A080 |
| pY | FF11A074 | FF11A07C |
| x | 6 | FF11A078 |
| y | 6 | FF11A074 |

```
// dereferencing
//
// Assign the value of the variable pointed to by pY
// to the variable pointed to by pX
*pX = *pY;
```

=> x = y

Assign the value of y to x.

# Arrays and Pointers

```
int p[10];   //      declare an array with name p
             //      p is the starting address of the array.
```

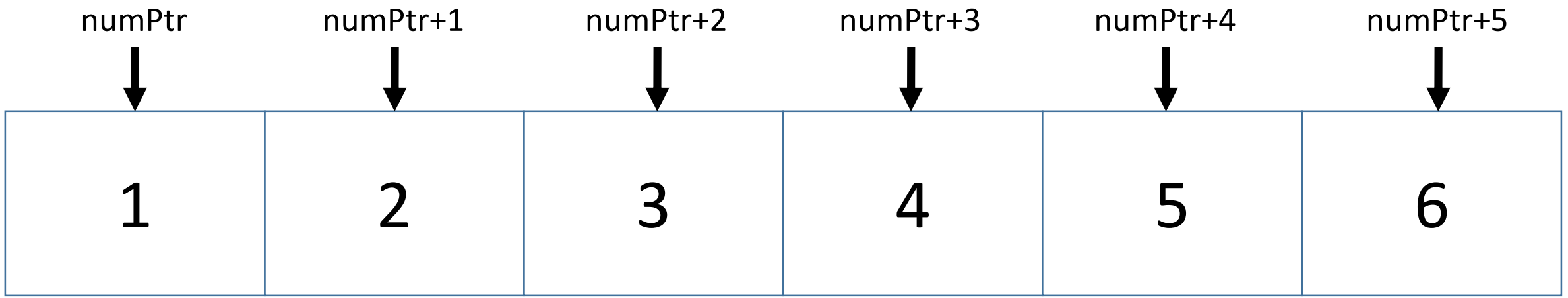
```
// numPtr is a pointer but the address of its content is fixed.
int numPtr[6] = {1, 2, 3, 4, 5, 6};
```

# Arrays and Pointers

```
int p[10];   //      declare an array with name p
             //      p is the starting address of the array.
```

// numPtr is a pointer but the address of its content is fixed.

```
int numPtr[6] = {1, 2, 3, 4, 5, 6};
```



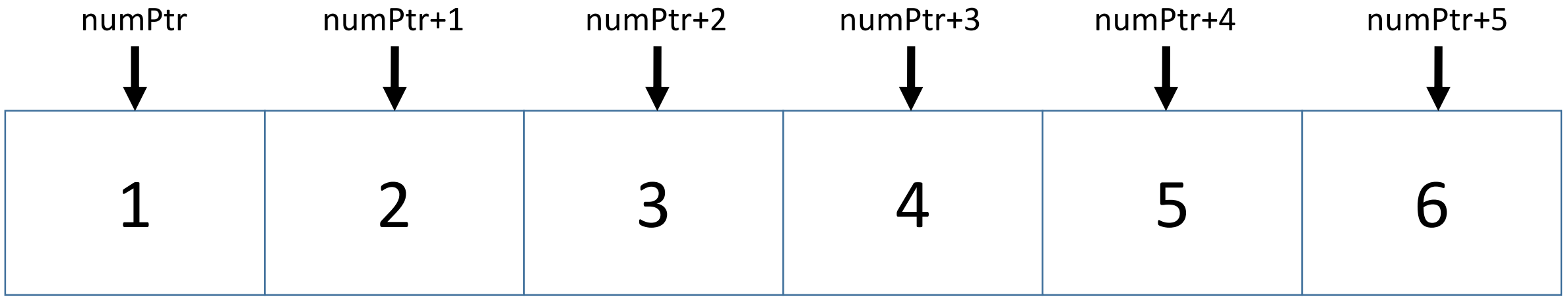
# Arrays and Pointers

```
int p[10];   //      declare an array with name p
             //      p is the starting address of the array.
```

// numPtr is a pointer but the address of its content is fixed.

```
int numPtr[6] = {1, 2, 3, 4, 5, 6};
```

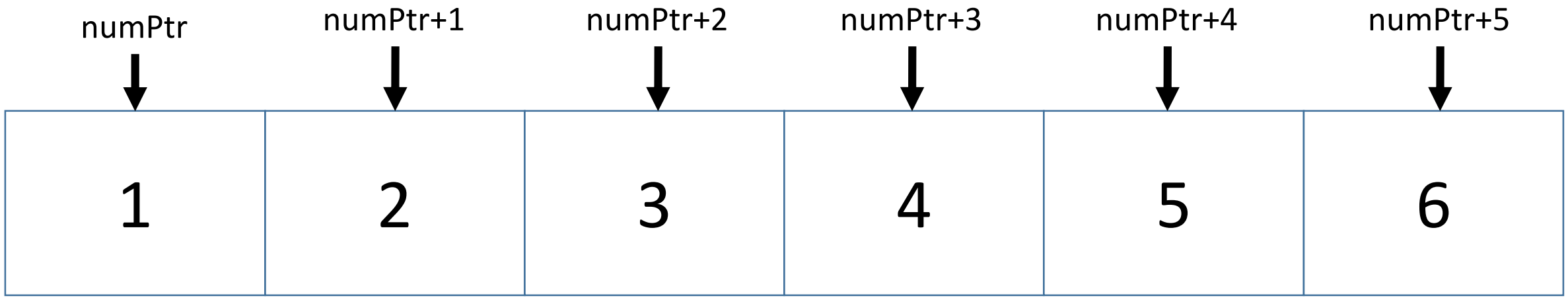
All are pointers



# Arrays and Pointers

**int** numPtr[6] = {1, 2, 3, 4, 5, 6};

| | | |
|---|---|---|
| numPtr[0] = ? | (numPtr+0)[0] = ? | *(numPtr+0) = ? |
| numPtr[1] = ? | (numPtr+1)[0] = ? | *(numPtr+1) = ? |
| numPtr[5] = ? | (numPtr+5)[0] = ? | *(numPtr+5) = ? |
| numPtr[6] = ? | (numPtr+1)[2] = ? | *(numPtr+3) +6 = ? |



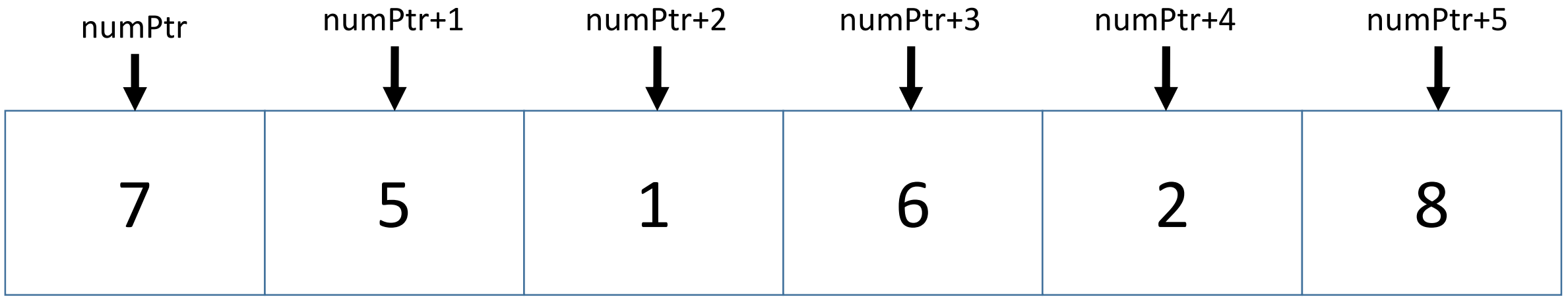
# Arrays and Pointers

**int** numPtr[6] = {7, 5, 1, 6, 2, 8};

numPtr[0] = A1
numPtr[1] = A2
numPtr[5] = A3
numPtr[6] = A4

(numPtr+0)[0] = A5
(numPtr+1)[0] = A6
(numPtr+5)[0] = A7
(numPtr+1)[2] = A8

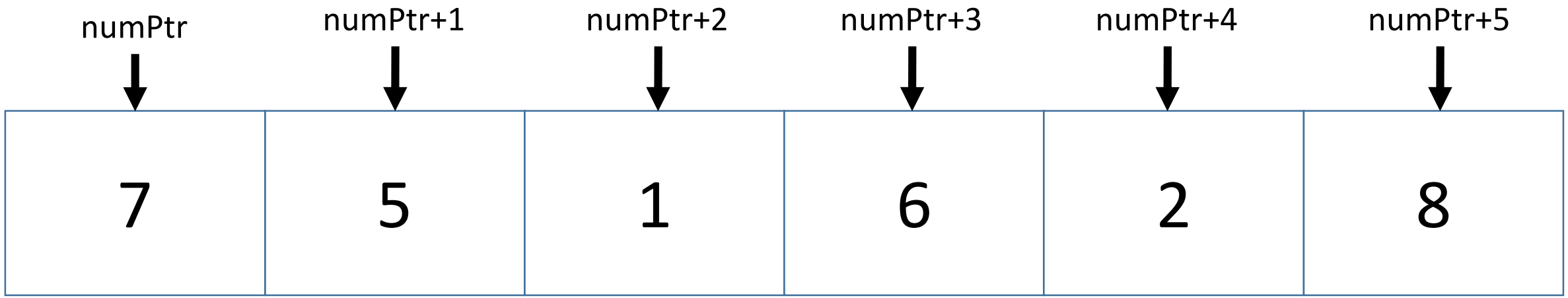
*(numPtr+0) = A9
*(numPtr+1) = A10
*(numPtr+5) = A11
*(numPtr+3) +6 = A12



# Arrays and Pointers

**int** numPtr[6] = {7, 5, 1, 6, 2, 8};

| | | |
|---|---|---|
| numPtr[0] = 7 | (numPtr+0)[0] = 7 | *(numPtr+0) = 7 |
| numPtr[1] = 5 | (numPtr+1)[0] = 5 | *(numPtr+1) = 5 |
| numPtr[5] = 8 | (numPtr+5)[0] = 8 | *(numPtr+5) = 8 |
| numPtr[6] = ?? | (numPtr+1)[2] = 6 | *(numPtr+3) +6 = 12 |



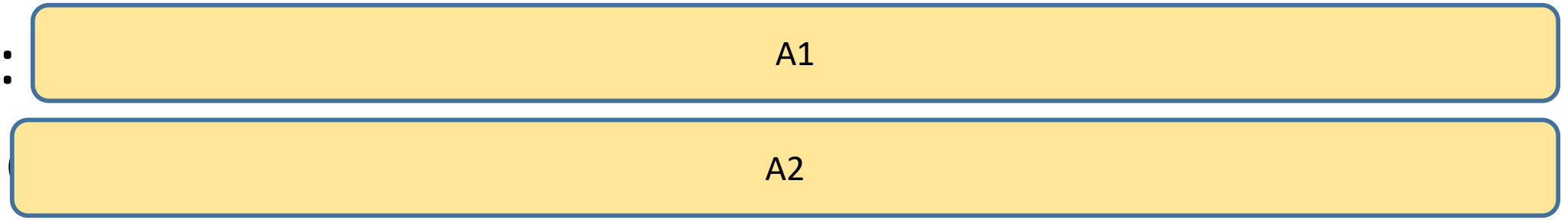
# Array Pointer

int list[ 10 ] = {1, 2, 3, 4};

*(list + 1) v.s. *list + 1

*(list +1):

*list + 1:




What is *(list+1)?

What is *list + 1?

# Array Pointer

int list[ 10 ] = {1, 2, 3, 4};

*(list + 1) v.s. *list + 1

*(list +1): points to the next element of the element pointed by list.

*list + 1: deferencing list and then add the value with 1

# Pointer Arguements

Pass-by-value
pass-by-reference

**void** f(**int*** p1, **int*** &p2)
which is equivalent to

**typedef int*** intPointer;
**void** f(intPointer p1, intPointer& p2)

// p1 is pass-by-value
// p2 is pass-by-reference

# Array parameter or pointer parameter

```
int func ( int *a, int size );

is equivalent to

int func ( int a [], int size );
```

```
int func ( char *a, int size );

is equivalent to

int func ( char a [], int size );
```

# const parameter

```
void foo( const int a ) {
……
}
```

```
void score( const STUDENT &s ) {
……
}
```

# const parameter

```
void foo( const int a ) {
……
}
```

```
void score( const STUDENT &s ) {
……
}
```

Pass-by-value
For simple value, do not need to use const

# const parameter

If an object value does not change,

declare it const to prevent it from being modified accidentally.

const int b = 10;

```
void foo( const int a ) {
……
}
```

```
void score( const STUDENT &s ) {
……
}
```

Pass-by-value

For simple value, do not need to use const

# Functions return a pointer

```
int *foo ( )
{
        // new: Allocation of a memory space
        int *a = new int;
        return a;
}

X *p ( )
{
        // new: Allocation of a memory space
        int *a = new X;                 // instantiate an object of X
        return a;
}
```

# Functions return a pointer

Can a function return a pointer?

The answer is yes.

```
int *foo ( )
{
        int *a = new int[100]; // A1 memory allocation
        return a;
}
```

# Functions return a pointer

Can a function return a pointer?

The answer is yes.

```
int *foo ( int size )
{
        int *a = new int[size ];
        return a;
}

int *arr = foo( n );
```

# Functions return a pointer

Can a function return a pointer?

The answer is yes.

```
int *foo ( int size )
{
        if ( size > 0 ) {
                int *a = new int[size ];
                return a;
        }
        return 0; // NULL
}

int *arr = f( n );
```

# Array Functions

min_element: return the minimal element in an array

max_element:  return the maximal element in an array

sort: sort an array,

random_shuffle: randomly shuffle an array

find: find an element in an array.

# Dynamic Memory Allocation

Generate elements of an array randomly.

```
int *foo( int n )
{
        int *array = A1 A2  // dynamic memory allocation
        for ( int i = 0; i < n; ++i ) array[i] = rand( );
        return array;
}
```

# Dynamic Memory Allocation

```
int* result = new int[6]; // Allocate an array of elements

delete [] result; // Deallocate each element of the array and itself


int* p = new int; // Allocate

delete p; // Deallocate
```

# Creating Dynamic Objects

Create objects dynamically on the heap

```
ClassName* pObject = new ClassName(); // no-arg constructor is invoked
ClassName *pObject = new ClassName; // no-arg constructor is invoked
ClassName* pObject = new ClassName(arguments);
```

```
// Create an object using the default constructor
vector<int>* p = new vector<int>;
// Create an object using the constructor with an arguments
vector<int>* p = new vector<int>(128);  // 128 is the size
```

# Accessing Dynamic Objects

```
CLASS_A   *x = new CLASS_A;
(*x).data_member                    // access a data member
(*&*x).data_member                   // access a data member
x->data_member                      // access a data member

(*x).foo( )                         // call a method
x->foo( )                           // call a method

(&(*x))->foo( )                     // call a method
(&*x)->foo( )                       // call a method

.     : dot operator
->    : arrow operator
```

# Accessing Dynamic Objects

```
CLASS_A   *x = new CLASS_A;
(*x).data_member                    // access a data member
(*&*x).data_member                   // access a data member
x->data_member                      // access a data member

(*x).foo( )                         // call a method
x->foo( )                           // call a method

(&(*x))->foo( )                     // call a method
(&*x)->foo( )                       // call a method
                                    &(*(&*x))->foo( )
.     : dot operator
->    : arrow operator
```

# Accessing Dynamic Objects

```
CLASS_A   *x = new CLASS_A;
(*x).data_member                    // access a data member
(*&*x).data_member                   // access a data member
x->data_member                      // access a data member

(*x).foo( )                         // call a method
x->foo( )                           // call a method

(&(*x))->foo( )                     // call a method
(&*x)->foo( )                       // call a method

.     : dot operator
->    : arrow operator
```

```
&(*(&*x))->foo( )
&(*(&(*(&(*(&*x))))->foo( )
```

# Accessing Dynamic Objects

```
CLASS_A   *x = new CLASS_A;
(*x).data_member                    // access a data member
(*&*x).data_member                   // access a data member
x->data_member                      // access a data member

(*x).foo( )                         // call a method
x->foo( )                           // call a method

(&(*x))->foo( )                     // call a method
(&*x)->foo( )                       // call a method

.     : dot operator
->    : arrow operator
```

```
&(*(&*x))->foo( )
&(*(&(*(&(*(&*x))))->foo( )
*(&(*(&(*(&(*(&*x)))))).foo( )
```

# *this* Pointer

Purpose: reference a class's hidden data field in a function.

```
class myClass {
     public:
     void foo(int a);
     int a;
}
void myClass::foo( int a )
{
     this->a = a;
}

myClass b;
b.foo( 10 );
```

The name of the formal parameter a is the same as the data member a.

# *this* Pointer

Purpose: reference a class's hidden data field in a function.

```
class myClass {
      public:
      void foo(int a);
      int a;
}
void myClass::foo( int a )
{
      this->a = a;
}

myClass b;
b.foo( 10 );
```

The name of the formal parameter a is the same as the data member a.

```
void myClass::foo( int a )
{
      a = a; // both are the formal parameter
}
```

# Destructors

Object creation: A constructor is invoked

Object destruction: A destructor is invoked

Every class has a default destructor if the destructor is not explicitly defined. The default constructor releases the memory space of the data members only but not the allocated dynamic memory.

```
class A {
  public:
        ~A( );          // put a tilde character (~)
};
```

# An example

```
class RECORD {
  int num;
  int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  ~RECORD( ) {
        if ( scoreArr != 0 ) { delete [ ] scoreArr; }
  }
  void input( ) {
        cout << “Enter the number of scores:” << endl;
        cin >> num;
        scoreArr  = new int[ num ];
  }
}
```

```
class RECORD {
  int num;
  int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  ~RECORD( ) {
        if ( scoreArr != 0 ) { delete [ ] scoreArr; }
  }
  void input( ) {
        cout << “Enter the number of scores:” << endl;
        cin >> num;
        scoreArr  = new int[ num ];
  }
};
```

```
void foo( )
{
   RECORD r;
   r.input( );
}
```

```
class RECORD {
int num;
int *scoreArr;
   RECORD( ) { num = 0; scoreArr = 0;}
   ~RECORD( ) {
      if ( scoreArr != 0 ) { delete [ ] scoreArr; }
   }
   void input( ) {
      cout << “Enter the number of scores:” << endl;
      cin >> num;
      scoreArr  = new int[ num ];
   }
};
```

```
void foo( )
{
   RECORD *r;
   r = new RECORD;
   r->input( );
}
```

```
class RECORD {
int num;
int *scoreArr;
   RECORD( ) { num = 0; scoreArr = 0;}
   ~RECORD( ) {
      if ( scoreArr != 0 ) { delete [ ] scoreArr; }
   }
   void input( ) {
      cout << “Enter the number of scores:” << endl;
      cin >> num;
      scoreArr  = new int[ num ];
   }
};
```

Assignment?

```
void foo( )
{
   RECORD r0, r1;
   r0.input( );
   r1 = r0;
}
```

```
class RECORD {
int num;
int *scoreArr;
   RECORD( ) { num = 0; scoreArr = 0;}
   ~RECORD( ) {
      if ( scoreArr != 0 ) { delete [ ] scoreArr; }
   }
   void input( ) {
      cout << “Enter the number of scores:” << endl;
      cin >> num;
      scoreArr  = new int[ num ];
   }
};
```

Assignment?

```
void foo( )
{
   RECORD r0, r1;
   r0.input( );
   r1 = r0;
}
```

Simple copy (shallow copy):

```
r1.num = r0.num;
r1.scoreArr = r0.scoreArr;
```

```
class RECORD {
int num;
int *scoreArr;
   RECORD( ) { num = 0; scoreArr = 0;}
   ~RECORD( ) {
      if ( scoreArr != 0 ) { delete [ ] scoreArr; }
   }
   void input( ) {
      cout << “Enter the number of scores:” << endl;
      cin >> num;
      scoreArr  = new int[ num ];
   }
};
```

```
Assignment?
void foo( )
{
   RECORD r0, r1;
   r0.input( );
   r1 = r0;
}
```

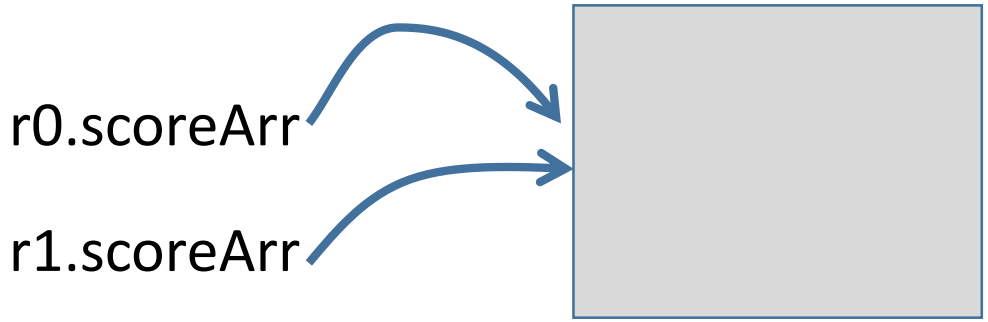
```
class RECORD {
int num;
int *scoreArr;
……
}
```

r0 and r1 share the same content of scoreArr.

Simple copy (shallow copy):

```
r1.num = r0.num;
r1.scoreArr = r0.scoreArr;
```

r0.scoreArr and r1.scoreArr point to the same memory space.



```
Assignment?
void foo( )
{
   RECORD r0, r1;
   r0.input( );
   r1 = r0;
}
```

```
class RECORD {
int num;
int *scoreArr;
……
}
```

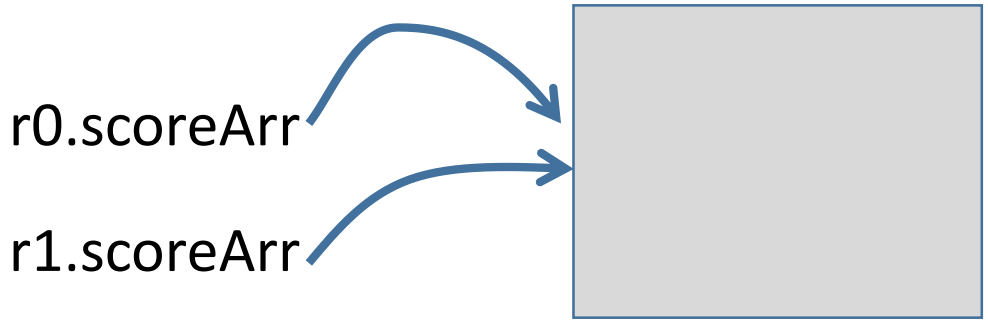
r0 and r1 share the same content of scoreArr.

Simple copy (shallow copy):

```
r1.num = r0.num;
r1.scoreArr = r0.scoreArr;
```

Assignment only.
Not copy elements

r0.scoreArr and r1.scoreArr point to the same memory space.



Assignment?

```
void foo( )
{
   RECORD r0, r1;
   r0.input( );
   r1 = r0;
}
```

```
class RECORD {
int num;
int *scoreArr;
……
}
```

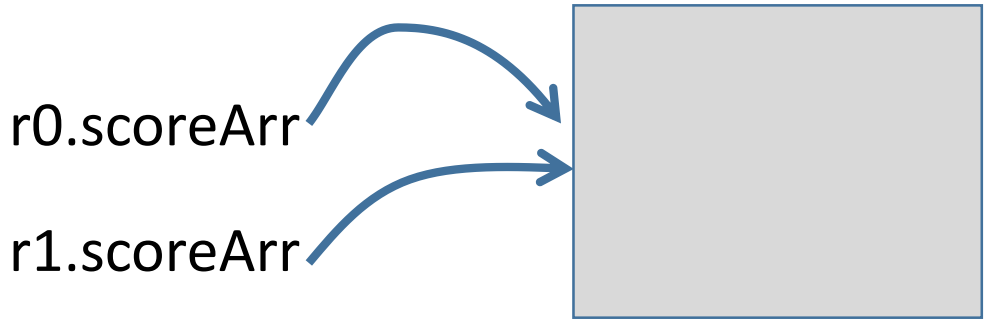
r0 and r1 share the same content of scoreArr.

Simple copy (shallow copy):

```
r1.num = r0.num;
r1.scoreArr = r0.scoreArr;
```

Assignment only.
Not copy elements

r0.scoreArr and r1.scoreArr point to the same memory space.




But you want to copy the elements of r0.scoreArr to r1.scoreArr. What should you do?

# Copy Constructors

Purpose: Create an object and initialize it with another object’s data

Signature of the copy constructor: ClassName(const ClassName&)

Simple copy: The default copy constructor performs it.

Example:

SQUARE ( const SQUARE &)

# Shallow Copy vs. Deep Copy

```
Assignment?
void foo( )
{
   RECORD r0, r1;
   r0.input( );
   r1 = r0;
}
```

shallow copy:

```
r1.num = r0.num;
r1.scoreArr = r0.scoreArr;
```

# Shallow Copy vs. Deep Copy

```
Assignment?
void foo( )
{
   RECORD r0, r1;
   r0.input( );
   r1 = r0;
}
```

shallow copy:

```
r1.num = r0.num;
r1.scoreArr = r0.scoreArr;
```

Assignment only.
Not copy elements

# Shallow Copy vs. Deep Copy

Shallow copy: It is adopted in the default copy constructor or assignment operator for copying.

Shallow copy: if the field is a pointer to some object, the address of the pointer is copied but not the content.

This may cause a run time error when objects are released from memory.

Assignment?

```
void foo( )
{
   RECORD r0, r1;
   r0.input( );
   r1 = r0;
}
```

shallow copy:

```
r1.num = r0.num;
r1.scoreArr = r0.scoreArr;
```

Assignment only.
Not copy elements

# Deep Copy

- Allocate a new memory space
- Copy the content to the new space

```
class RECORD {
int num;
int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  …
  RECORD ( const RECORD &r) {
    num = r.num;
    if ( num > 0 ) {
      scoreArr = new int[ num  ] ;
      for ( int i = 0; i < num; ++i ) {
        scoreArr[i] = r.scoreArr[ i ] ;
      } // for
    } // if
  }
};
```

# Deep Copy

- Allocate a new memory space
- Copy the content to the new space

```
class RECORD {
int num;
int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  …
 RECORD ( const RECORD &r) {
    num = r.num;
    if ( num > 0 ) {
       scoreArr = new int[ num  ] ;
       for ( int i = 0; i < num; ++i ) {
           scoreArr[i] = r.scoreArr[ i ] ;
       } // for
    } // if
 }
};
```

```
RECORD a;
a.input( );

RECORD b  = a;



RECORD c;

c = a;  //
        //
        //
        //
```

```
class RECORD {
int num;
int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  …
  RECORD ( const RECORD &r) {
    num = r.num;
    if ( num > 0 ) {
      scoreArr = new int[ num  ] ;
      for ( int i = 0; i < num; ++i ) {
        scoreArr[i] = r.scoreArr[ i ] ;
      } // for
    } // if
  }
};
```

```
RECORD a;
a.input( );

RECORD b  = a;      //
                    //

RECORD c;

c = a;  //
        // It's an initialization
        // it's an
        // copy is invoked.
```

A1. Purpose?

A2. Assignment? Copy constructor?

A3

A4

A5. What is invoked?

```
class RECORD {
int num;
int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  …
  RECORD ( const RECORD &r) {
    num = r.num;
    if ( num > 0 ) {
      scoreArr = new int[ num  ] ;
      for ( int i = 0; i < num; ++i ) {
        scoreArr[i] = r.scoreArr[ i ] ;
      } // for
    } // if
  }
};
```

```
RECORD a;
a.input( );
                        //declaration
RECORD b  = a;          //initialization
                        //copy constructor

RECORD c;

c = a;  //
        // It's not an initialization
        // it's an assignment.
        // Shallow copy is invoked.
```

```
class RECORD {
int num;
int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  …
  RECORD ( const RECORD &r) {
    num = r.num;
    if ( num > 0 ) {
      scoreArr = new int[ num  ] ;
      for ( int i = 0; i < num; ++i ) {
        scoreArr[i] = r.scoreArr[ i ] ;
      } // for
    } // if
  }
};
```

```
RECORD a;
a.input( );
                          //declaration
RECORD b  = a;       //initialization.
                          //copy constructor


RECORD c;


c = a;  //
        // It's not an initialization
        // it's an assignment.
        // Shallow copy is invoked.
```

```
class RECORD {
int num;
int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  …
  RECORD ( const RECORD &r) {
    num = r.num;
    if ( num > 0 ) {
      scoreArr = new int[ num  ] ;
      for ( int i = 0; i < num; ++i ) {
        scoreArr[i] = r.scoreArr[ i ] ;
      } // for
    } // if
  }
};
```

```
RECORD a;
a.input( );
                          //declaration
RECORD b  = a;       //initialization.
                          //copy constructor


RECORD c;

c = a;  //
        // It's not an initialization
        // it's an assignment.
        // Shallow copy is invoked.
```

Need to implement the assignment operator = to have deep copy.

```
class RECORD {
int num;
int *scoreArr;
  RECORD( ) { num = 0; scoreArr = 0;}
  …
  RECORD ( const RECORD &r) {
    num = r.num;
    if ( num > 0 ) {
      scoreArr = new int[ num  ] ;
      for ( int i = 0; i < num; ++i ) {
        scoreArr[i] = r.scoreArr[ i ] ;
      } // for
    } // if
  }
};
```

We need to implement the copy constructor to have deep copy.

Deep copy is not given.

# Function pointers

# Function pointers

A function pointer points to a function.

```
void (*fPtr)();

void g( ) {
cout << 8 << endl;

}

fPtr = g;
fPtr( );
```

```
void (*fPtr)(int);

void g( int a) {
   cout << a << endl;
}

fPtr = g;
fPtr( 5 );
fPtr( 6 );
```

```
void g( int a) {
   cout << a << endl;
}
void h(void (*fPtr)(int), int num) {
   fPtr(num);
}

h(g, 5);
h(g,10);
```

# Function pointers

A function pointer points to a function.

```
void (*fPtr)();

void g( ) {
cout << 8 << endl;

}

fPtr = g;
fPtr( );
```

```
void (*fPtr)(int);

void g( int a) {
   cout << a << endl;
}

fPtr = g;
fPtr( 5 );
fPtr( 6 );
```

```
void g( int a) {
   cout << a << endl;
}
void h( void (*fPtr)(int), int num) {
   fPtr(num);
}

h(g, 5);
h(g,10);
```

# Intended Learning Outcomes

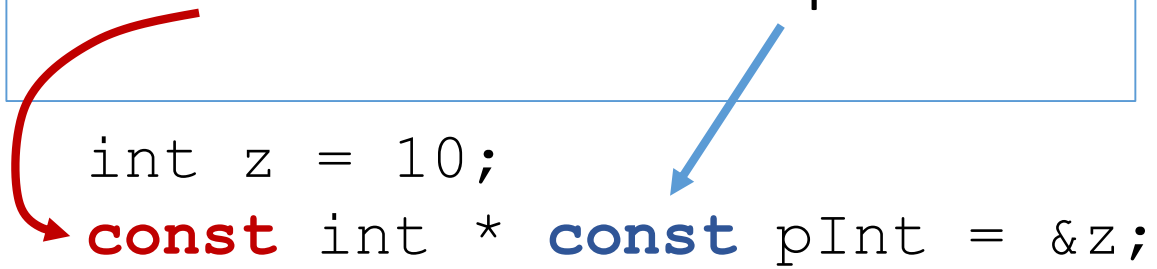
- Distinguish between little endian and big endian
- Describe how to use pointers
- Define a copy constructor structure
- Distinguish between shallow copy and deep copy

# Supplemental Material

# Learn the followings:

- What's a pointer?
- Declare a pointer
- Dereferencing
- Pointer type
- **typedef** existingType newType;
- Assignments using pointers

Constant data  Constant pointer



```
int z = 10;
const int * const pInt = &z;
```

- Arrays and pointers

  int *ptr; int b[10]; ptr = b

- Returning a pointer from a function
- Dynamic memory allocation
- this
- Dynamic objects
- Copy constructor
  - Deep and shallow copy
- Destructor

# const

A constant cannot be changed after it is declared.

```
const double pi = 3.14;                 // a constant double number
double radius = 5;
double* const pValue = &radius;   // a constant pointer

const double* pValue = &radius;   // a pointer points to a variable
```

```
const double * const value = &radius;
```

Constant data

Constant pointer

Exercises

```
int b[] = {1, 2, 3, 4};          //What is b? Its element values?
int *p, a = 10;                  // L1
p = &a;                          // L2
*p = 11 + a - *p;                // L3
p = new int[16];                 // L4
*p = 1;                          // L5          p[0], p[1], p[2]=?
*(p+1) = 2;                      // L6          p[0], p[1], p[2]=?
*(p+2) = *p + *p + 1;            // L8          p[0], p[1], p[2]=?
*(p+16) = 100;                   // L9
```

# Exercise

```
int *a, *b;
int x =5;
int y = 6;
a =&y;
b = &x;
…
*b = ++(*a); // what are the values of a and b?
```

# Exercise

```
int *a, *b;
int x =5;
int y = 6;
a =&y;
b = &x;
…
*b = ++(*a); // what are the values of a and b?
```

# Exercise

```
int *a, *b;
int x =5;
int y = 6;
a =&y;
b = &x;
…
*b = ++(*a); // what are the values of a and b?
```

| variable | Memory space | Memory address |
|---|---|---|
| a | ???? | FF11A080 |
| b | ???? | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Exercise

```
int *a, *b;
int x =5;
int y = 6;
a =&y;
b = &x;
…
*b = ++(*a); // what are the values of a and b? x and y?
```

| variable | Memory space | Memory address |
|---|---|---|
| a | FF11A074 | FF11A080 |
| b | FF11A078 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Exercise

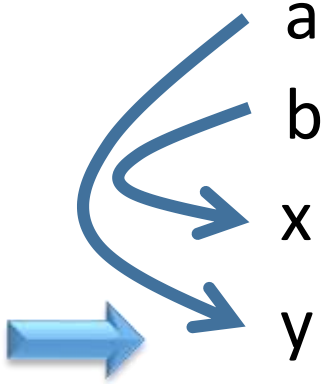
```
int *a, *b;
int x =5;
int y = 6;
a =&y;
b = &x;
...
*b = ++(*a); // what are the values of a and b? x and y?
```

| variable | Memory space | Memory address |
|---|---|---|
| a | FF11A074 | FF11A080 |
| b | FF11A078 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

# Exercise

```
int *a, *b;
int x =5;
int y = 6;
a =&y;
b = &x;
...
```

***b = ++(*a);** // what are the values of a and b? x and y?



| variable | Memory space | Memory address |
|---|---|---|
| a | FF11A074 | FF11A080 |
| b | FF11A078 | FF11A07C |
| x | 5 | FF11A078 |
| y | 6 | FF11A074 |

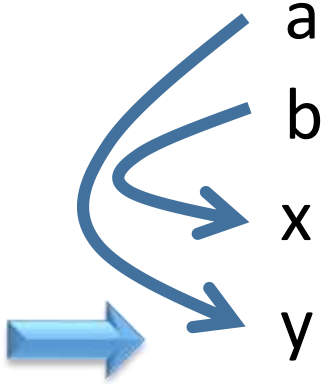
Dereferencing a

To get the A1 pointed to by a

It is **incorrect** to say:
"get the value pointed to by a."
Cannot perform ++ on a value.

# Exercise

```
int *a, *b;
int x =5;
int y = 6;
a =&y;
b = &x;
…
*b = ++(*a); // what are the values of a and b? x and y?
```



| variable | Memory space | Memory address |
|---|---|---|
| a | FF11A074 | FF11A080 |
| b | FF11A078 | FF11A07C |
| x | 5 | FF11A078 |
| y | 7 | FF11A074 |

Get the variable which is pointed to by a,
and then increase its value by one

# Exercise

```
int *a, *b;
int x =5;
int y = 6;
a =&y;
b = &x;
…
```

***b = ++(*a);** // what are the values of a and b? x and y?

| variable | Memory space | Memory address |
|---|---|---|
| a | FF11A074 | FF11A080 |
| b | FF11A078 | FF11A07C |
| x | 7 | FF11A078 |
| y | 7 | FF11A074 |

Assign the right result to the variable pointed to by b

# Exercises: Shallow Copy vs Deep Copy

```
MyClass a, b;

a = b; // what is called for performing the assignment?

SHAPE x;
SHAPE y = x; // what is called in here?

SHAPE *z;              // Does z point to an object of SHAPE?
z = new SHAPE;         // Does z point to an object of SHAPE?
z->input( );           // How do we call the method input of z?
                       // Alternatives 1, 2, 3..?
```

# Exercises: Shallow Copy vs Deep Copy

```
MyClass a, b;

a = b; // what is called for performing the assignment?
```

Shallow copy if no assignment operator

```
SHAPE x;
SHAPE y = x; // what is called in here?

SHAPE *z;                  // Does z point to an object of SHAPE?
z = new SHAPE;             // Does z point to an object of SHAPE?
z->input( );               // How do we call the method input of z?
(*z).input();              // Alternatives 1, 2, 3..?
(&(*z))->input();
(*(&(*z))).input();
&(*z).foo();               // Assume that there is no error. What is the meaning of this line?
```

# Exercises
# What're the contents of arrays x and y?

```
int x[32] = {1};                    // L0
int y[32] = {};                     // L1
```

// Write a program and output the elements of x and y to see the results.

# Exercise

```
int a[10]; //static array: size is fixed; address is fixed; name is fixed
int *p = a;
p[0] = 1;
a[0] = 1;

p[1] = 11;              //?
p[10] = 101;            //?
p[11] = 100;            //?
```

# Others

MyClass a, *b;

a = b;  // Line 1

*b = a;  // Line 2

b = &a;  // Line 3

(*b), a, b->  a.x = 10,  b->x = 10

**Exercise**

**What is the problem?**

```
class RECORD {
   int num;
   int *scoreArr;
public:
     RECORD( ) { num= 10; scoreArr = new int[num];}
   ~ RECORD() { if (scoreArr) delete [ ] scoreArr; }
}

void foo( ) {
        RECORD r0, r1;
        r1 = r0;
}
```

**Assume**

A1 **copy**

A2 **copy)**

**Exercise**
**What is the problem?**

```
class RECORD {
   int num;
   int *scoreArr;
public:
      RECORD( ) { num= 10; scoreArr = new int[num];}
   ~ RECORD() { if (scoreArr) delete [ ] scoreArr; }
}

void foo( ) {
         RECORD r0, r1;
         r1 = r0;
}
```

**Assume**
**simple copy**
**(shallow copy)**