Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Chapter 8: Pointers

Introduction to Computer Programming
(C language)

Nguyễn Tiến Thịnh, Ph.D.

Email: ntthinh@hcmut.edu.vn

2022 – 2023, Semester 1

Course Content

- C.1. Introduction to Computers and Programming
- C.2. C Program Structure and its Components
- C.3. Variables and Basic Data Types
- C.4. Selection Statements
- C.5. Repetition Statements
- C.6. Functions
- C.7. Arrays
- **C.8. Pointers**
- C.9. File Processing

References

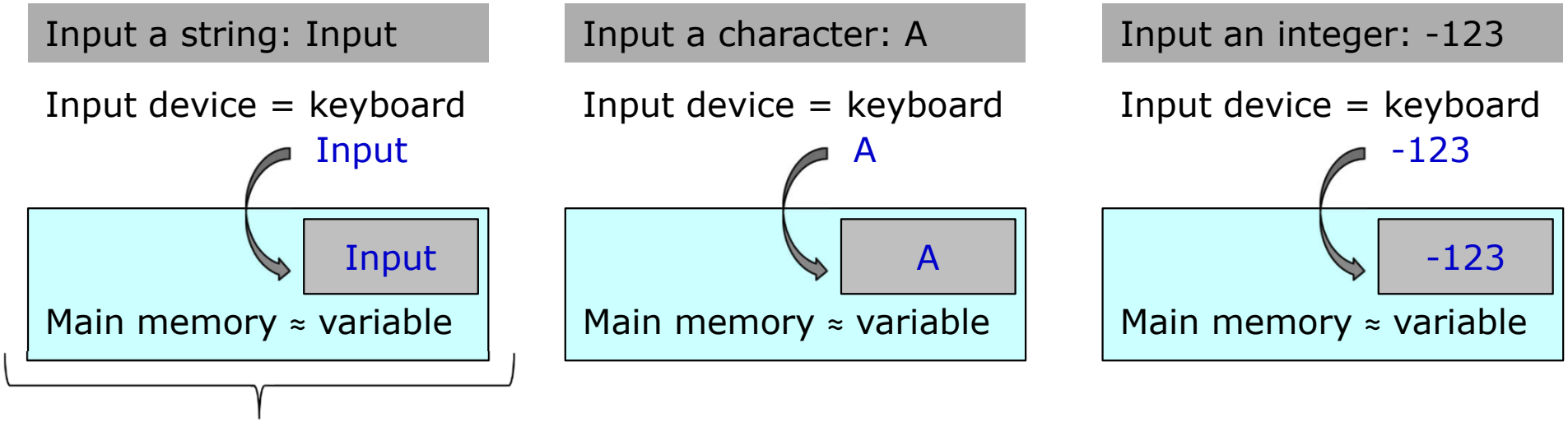
- [1] "*C: How to Program*", 7th Ed. – Paul Deitel and Harvey Deitel, Prentice Hall, 2012.
- [2] "*The C Programming Language*", 2nd Ed. – Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1988
- and others, especially those on the Internet

Content

- Introduction
- Declare and initialize pointers
- Operations on pointers
- Pointers and arrays
- Variable storage and heap memory
- Memory allocation and de-allocation
- Pointers and structures
- Pass pointers to a function
- Function pointers
- Summary

Introduction - Recall - Chapter 2

- Main memory is addressable continuously.
- scanf() for input data from input devices to main memory



Varying size: user-predefined

Fixed sizes: character = 1 byte, integer = 4 bytes, …

```
char aString[5];
…
scanf("%s", aString)

printf("%s", aString)
```

```
char aChar;
…
scanf("%c", &aChar)

printf("%c", aChar)
```

```
int anInteger;
…
scanf("%d", &anInteger)

printf("%d", anInteger)
```

Introduction - Recall - Chapter 3

- Built-in data types (primitive/fundamental)
  - char (signed char), unsigned char
  - short int, unsigned short, int, unsigned int, long int, unsigned long int, long long int, unsigned long long
  - float, double, long double
  - void
  - enum (enumerated data associated with integers)
- Derived data types
  - arrays [] of objects of a given type
  - pointers * to objects of a given type
  - structures struct containing objects of other types
  - union containing any one of several objects of various types

We haven't discussed this derived data type in detail yet!!!

Introduction - Recall - Chapter 3

Name	Operator	Description	Example
sizeof	**sizeof**(type), **sizeof**(variable)	Returns the size (bytes) of a type or a variable	**sizeof**(**char**) **int** anInt = 0; **sizeof**(anInt);
address	**&**Variable	Returns the address of the memory named Variable	**char** aChar; **char*** ptrChar; ptrChar = **&**aChar;
Dereferencing	***Pointer**	Returns the value of the memory Pointer points to	aChar = ***ptrChar + 1;

Index	Variable**[..]**	Returns the element at the index	**int** intArray[3];
			
			intArray**[**0**]** = 0;
			intArray**[**1**]** = 1;
			intArray**[**2**]** = 2;
			anInt = intArray**[**1**]**;

Structure member	Structure_ name**.**member	Refers to a member of a particular structure	**struct** point pt; pt**.**x = 10;

Introduction - Recall - Chapter 6

A function to swap two integer numbers

```
void swap(int a, int b){
      int temp;
      temp = a;
      a = b;
      b = temp;
}
```

a and **b** will be passed by values of int type.

```
void swap(int *a, int *b){
      int temp;
      temp = *a;
      *a = *b;
      *b = temp;
}
```

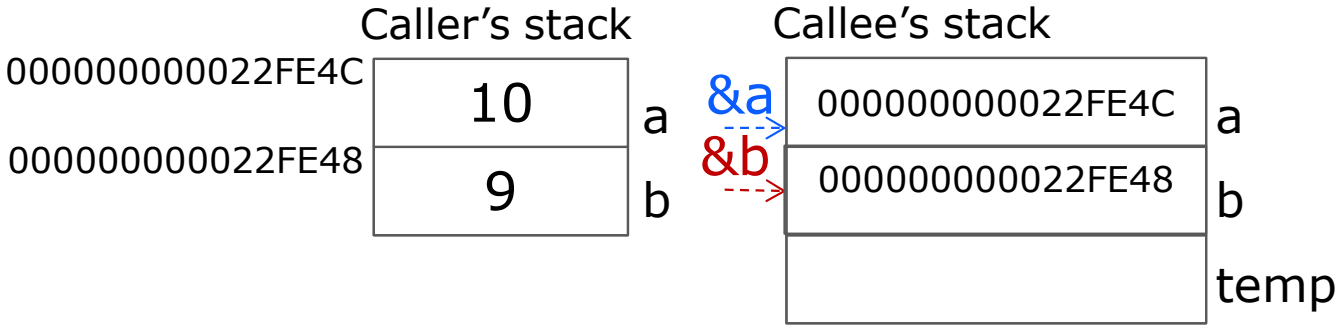
a and **b** will be passed by pointers to int values, i.e. addresses of the memory that contains int values.

Introduction - Recall - Chapter 6

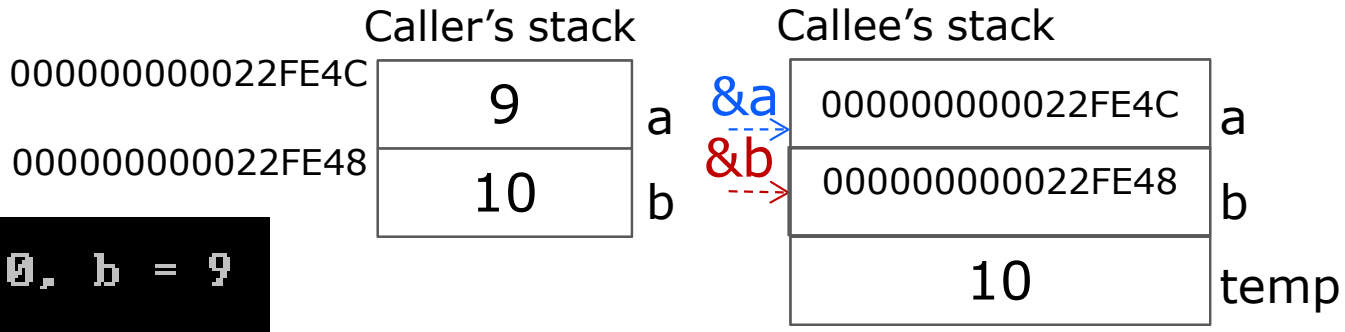
```c
#include <stdio.h>

void swap (int *a, int *b) {
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

void main() {
    int a = 10, b = 9;

    printf("\nBefore swapping, a = %d, b = %d \n", a, b);

    swap(&a, &b);

    printf("\nAfter swap
}
```

```
Before swapping, a = 10, b = 9

After swapping, a = 9, b = 10
```

Stack's values when a=10 and b=9 in the main() function



Stack's values before the callee ends

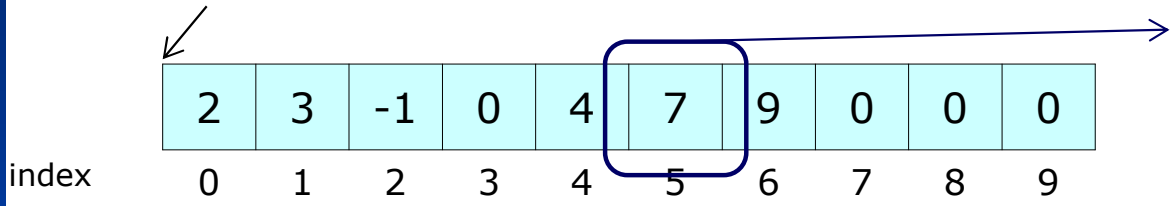


Introduction - Recall - Chapter 7

```
The b matrix:

  2   3  -1   0   4
  7   9   0   0   0
  6  11  -2   5   0

b[2] = 000000000022FE18

&b[2][0] = 000000000022FE18
```

int a[10] = {2, 3, -1, 0, 4, 7, 9};

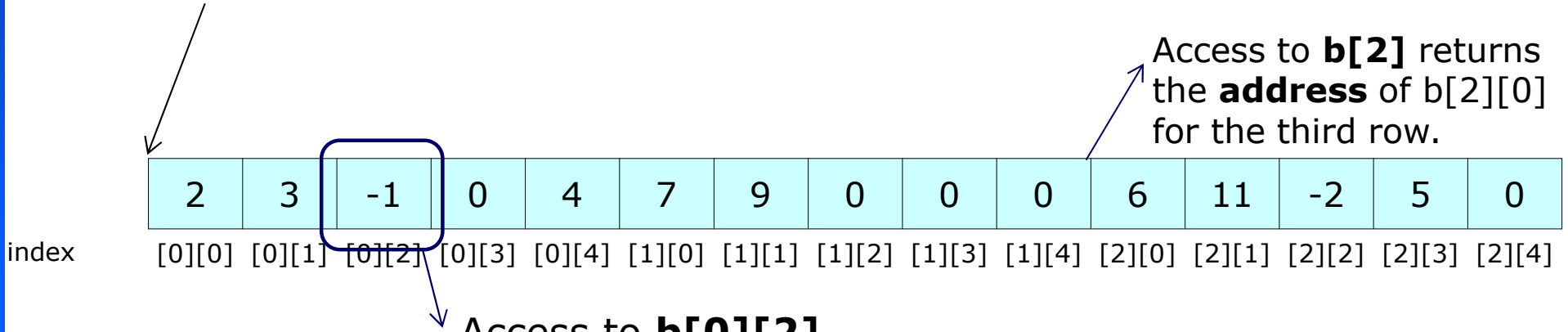
a, array name, is the address of the first **int** memory location.



Access to **a[5]** returns an **int** value: 4.

int b[3][5] = {{2, 3, -1, 0, 4}, {7, 9}, {6, 11, -2, 5}};

b, array name, is the address of the first **int** memory location.



Access to **b[2]** returns the **address** of b[2][0] for the third row.

Access to **b[0][2]** returns an **int** value: -1.

Introduction - Recall - Chapter 7

- Pass a value of an element at index i of a one-dimension array a to functions
  - Call to function *func*: *func*(a[i], …)

  Value passing - unchanged
- Pass all the values of the elements of a one-dimension array a to functions
  - Call to function *func*: *func*(a, …)

  Address passing - changeable
- Pass a value of an element at indices i and j of a two-dimension array b to functions
  - Call to function *func*: *func*(b[i][j], …)

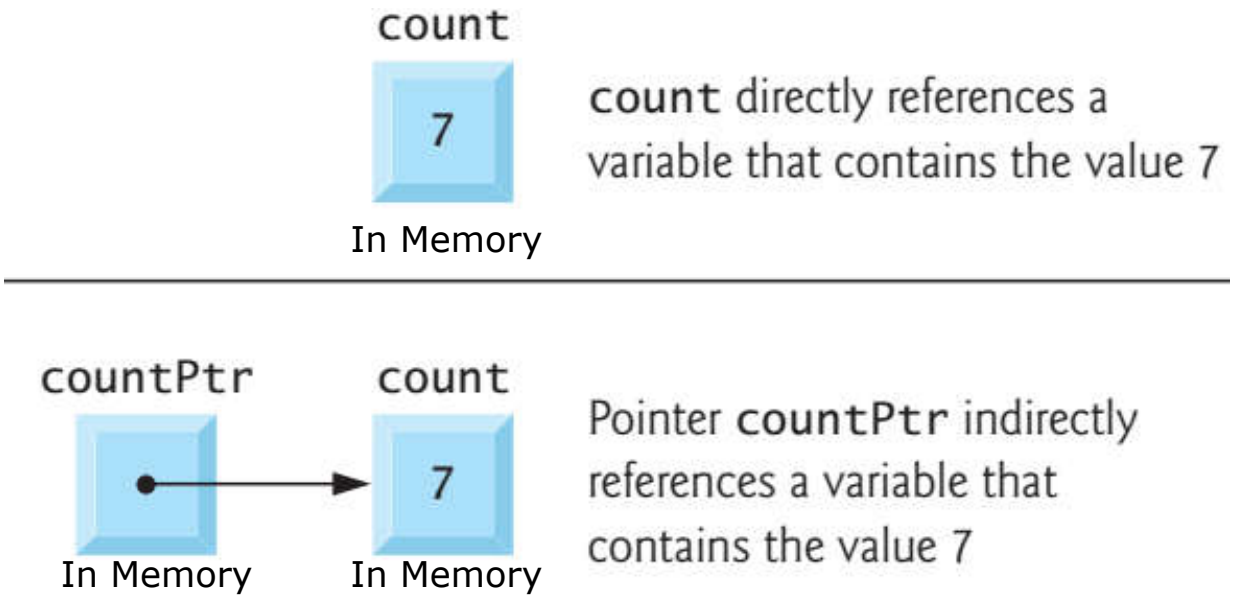
  Value passing - unchanged
- Pass a row at index i of a two-dimension array b to functions
  - Call to function *func*: *func*(b[i], …)

  Address passing - changeable
- Pass all the values of the elements of a two-dimension array b to functions
  - Call to function *func*: *func*(b, …)

  Address passing - changeable

Introduction

- Related to physical memory addresses, pointers are provided as
  - A means for manipulation on memory
  - A means for pass-by-reference implementation
  - A means for dynamic data structures that can grow and shrink at execution time



[1], Figure 7.1, pp. 254

Declare and initialize pointers

- A pointer of a data type
  - A variable whose value is an address of memory location that contains a value of a data type
  - Must be declared before its use
    - optionally with an initial value

type_name variable_name* $=_{opt}$ *expression*$_{opt}$;

- *variable_name*: a valid identifier for a pointer

- *type_name*: a valid data type (basic, derived, a new one with **typedef**)

- *type_name**: a pointer type for pointers that point to memory location of a data type *type_name*

- *expression*: an initial value (0, NULL, an address)

Declare and initialize pointers

type_name variable_name* $=_{opt}$ *expression*$_{opt}$;

```
char aChar = 'a';
char* pChar1;
char* pChar2 = 0;
char* pChar3 = NULL;
char* pChar4 = &aChar;
```

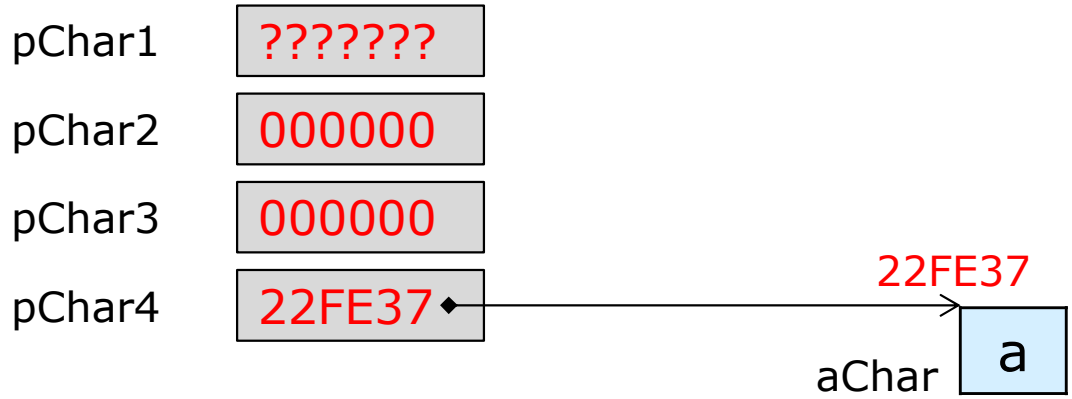
// a variable of the *char* data type

// an un-initialized pointer pointing to a char

// a null pointer pointing to a char

// a null pointer pointing to a char

// a pointer pointing to a char with an initialized pointer to aChar



Declare and initialize pointers

*type_name** *variable_name* =$_{opt}$ *expression*$_{opt}$;

```
int anInt = 10;
int* pInt1;
int* pInt2 = 0;
int* pInt3 = NULL;
int* pInt4 = &anInt;
```

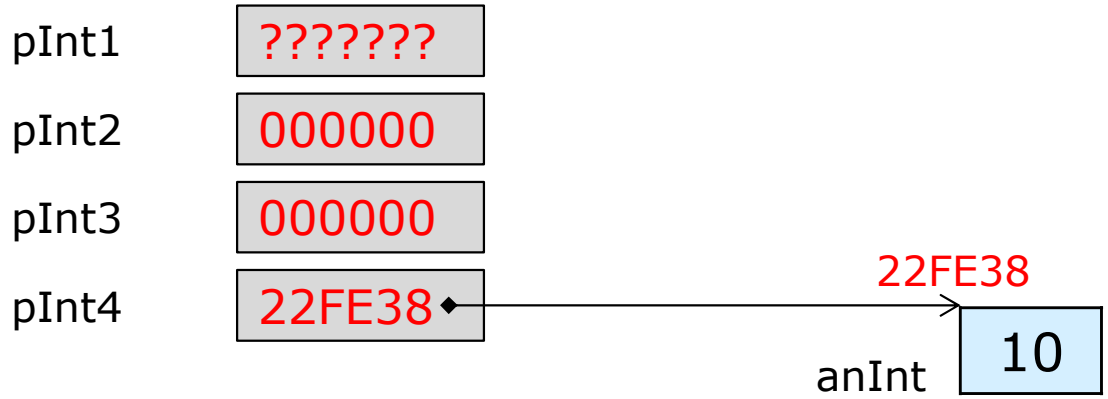
// a variable of the *int* data type

// an un-initialized pointer pointing to an int

// a null pointer pointing to an int

// a null pointer pointing to an int

// a pointer pointing to an int with an initialized pointer to anInt



Declare and initialize pointers

*type_name** *variable_name* =$_{opt}$ *expression*$_{opt}$;

```
float aFloat = 10.5;
float* pFloat1;
float* pFloat2 = 0;
float* pFloat3 = NULL;
float* pFloat4 = &aFloat;
```

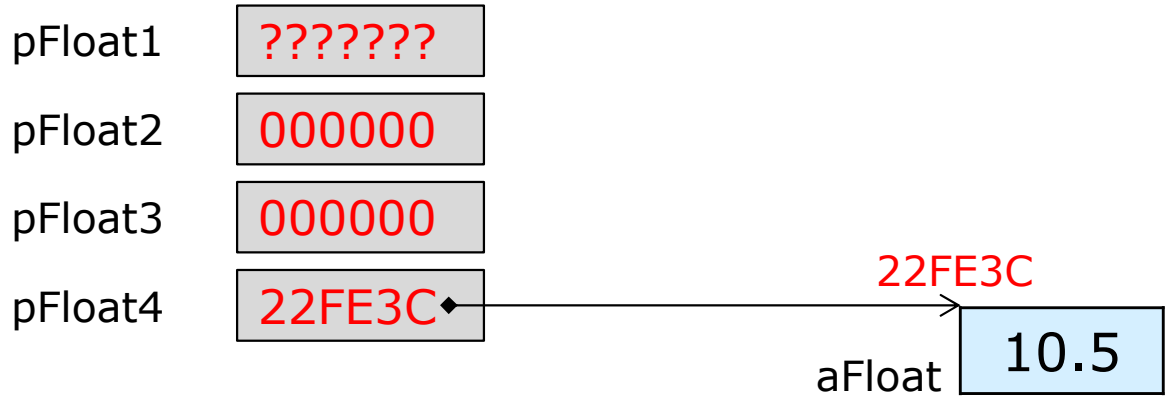
// a variable of the *float* data type

// an un-initialized pointer pointing to a float

// a null pointer pointing to a float

// a null pointer pointing to a float

// a pointer pointing to a float with an initialized pointer to aFloat

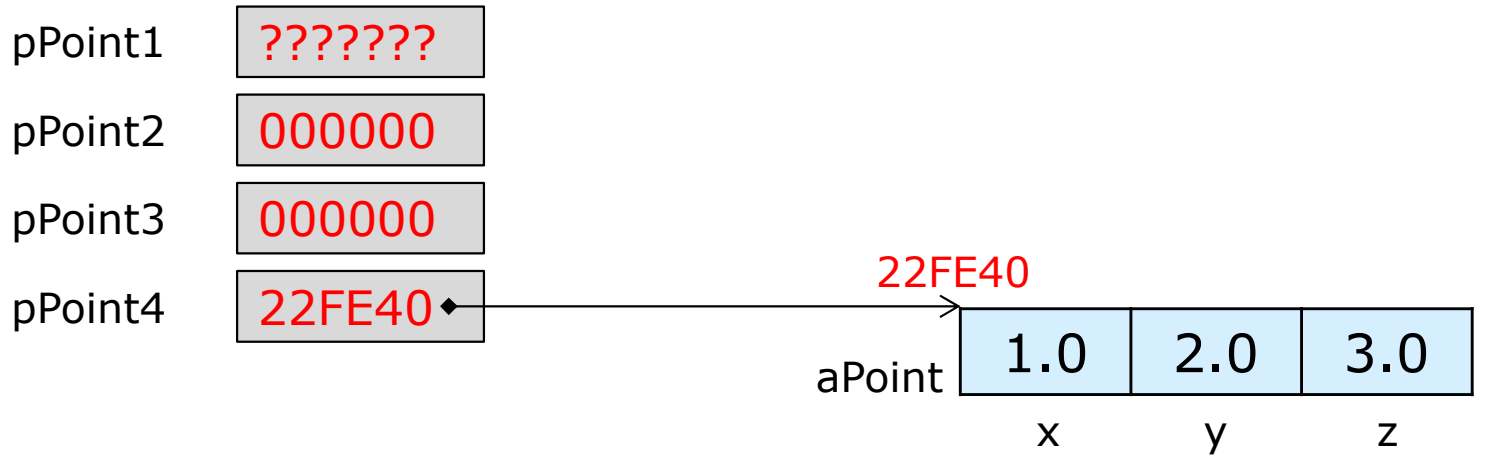


Declare and initialize pointers

type_name variable_name* $=_{opt}$ *expression*$_{opt}$;

```
struct point3D {
        float x;
        float y;
        float z;
    };
```

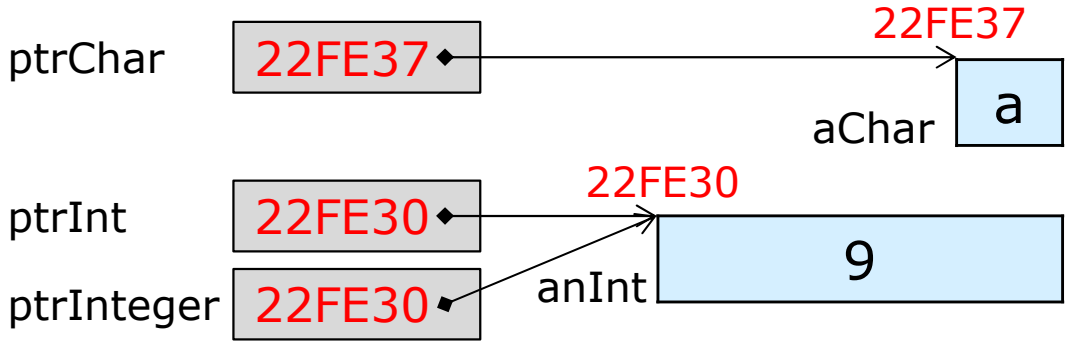
```
struct point3D aPoint = {1.0, 2.0, 3.0};// a variable of the point3D data type
struct point3D* pPoint1;                // an un-initialized pointer pointing to a point3D
struct point3D* pPoint2 = 0;            // a null pointer pointing to a point3D
struct point3D* pPoint3 = NULL;         // a null pointer pointing to a point3D
struct point3D* pPoint4 = &aPoint;      // a pointer pointing to a point3D with an
                                        initialized pointer to aPoint
```



Declare and initialize pointers

```
#include <stdio.h>

typedef int Integer;

void main() {

    char aChar = 'a';

    char* ptrChar = &aChar;

    int anInt = 9;

    int* ptrInt;

    ptrInt = &anInt;

    Integer* ptrInteger = &anInt;


    printf("\naChar = \'%c\' at address %p\n", aChar, &aChar);

    printf("\nptrC                                    \'%c\'\n",

    printf("\nanInt = %d at address %p\n", anInt, &anInt);

    printf("\nptrInt = &anInt = %p with value %d\n", ptrInt, *ptrInt);

    printf("\npt                                      n", ptrInteger,
}
```

```
aChar = 'a' at address 000000000022FE37
ptrChar = &aChar = 000000000022FE37 with value 'a'
anInt = 9 at address 000000000022FE30
ptrInt = &anInt = 000000000022FE30 with value 9
ptrInteger = &anInt = 000000000022FE30 with value 9
```



an *int* value in the memory location named anInt: 9

an *address* of the memory location named anInt: 22FE30

an *address* of the memory location pointed by ptrInt: 22FE30

an *int* value in the memory location pointed by ptrInt: 9

Declare and initialize pointers

- A pointer of a data type
  - A variable whose value is an address of memory location that contains a value of a data type

→ What if a particular data type is not told?

→ Pointers to void

void * pointer;

Declare and initialize pointers

- Pointers to void
  - The generic pointer type which is a flexible means to manipulate memory for any data type
    - Adaptive data structures
    - Function definitions
  - A pointer to any data type can be assigned directly to a pointer of type void *.
  - A pointer of type void * can be assigned directly to a pointer to any data type.
  - A pointer of type void * can be casted to any data type.
  - A void * pointer cannot be dereferenced.

Pointers to void

```
anInt = 9 at address 000000000022FE34
pInt = &anInt = 000000000022FE34 with value 9
pVoid = pInt = 000000000022FE34 with value 9
pFloat = pVoid = 000000000022FE34 with value 0.000000 while anInt = 9
```

```c
#include <stdio.h>

void main() {

    int anInt = 9;

    int* pInt = &anInt;

    void* pVoid = NULL;

    float* pFloat = NULL;

    pVoid = pInt;

    pFloat = pVoid;

    printf("\nanInt = %d at address %p\n", anInt, &anInt);

    printf("\npInt = &anInt = %p with value %d\n", pInt, *pInt);

    //printf("\npVoid = pInt = %p with value %d\n", pVoid, *pVoid); //error with *pVoid

    printf("\npVoid = pInt = %p with value %d\n", pVoid, *((int*)pVoid));

    printf("\npFloat = pVoid = %p with value %f while anInt = %d\n", pFloat, (float)*pFloat, anInt);
}
```

Dereference of a **void*** pointer:

pVoid** should be: ***((**int)pVoid)

Declare and initialize pointers

- A pointer of a data type
  - A variable whose value is an address of memory location that contains a value of a data type

→ What if a data type is a pointer type?

→ Pointers to pointers

  - A pointer that points to the memory location whose value is an address

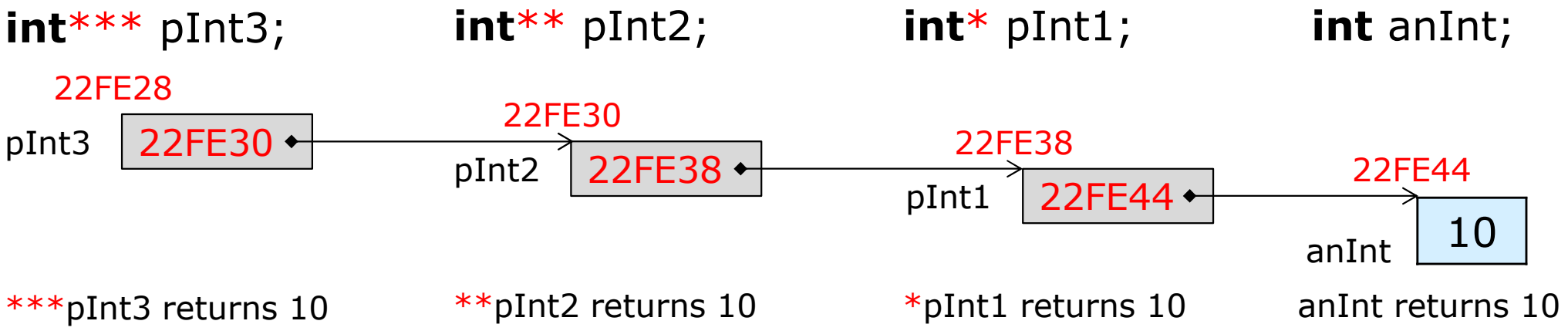
```
int anInt = 10;
```
→ A conventional variable

```
int* pInt1 = &anInt;
```
→ A pointer to an **int** memory location

```
int** pInt2 = &pInt1;

int*** pInt3 = &pInt2;
```
→ Pointers to pointers

```
anInt = 10 at address 000000000022FE44
pInt1 = &anInt = 000000000022FE44 points to anInt whose value = 10
pInt2 = &pInt1 = 000000000022FE38 points to pInt1 whose value = 000000000022FE44

pInt3 = &pInt2 = 000000000022FE30 points to pInt2 whose value = 000000000022FE38

An integer number indirectly referred by pInt1 = *pInt1 = 10
An integer number indirectly referred by pInt2 = **pInt2 = 10
An integer number indirectly referred by pInt3 = ***pInt3 = 10
```

```c
#include <stdio.h>

void main() {

    int anInt = 10;

    int* pInt1 = &anInt;

    int** pInt2 = &pInt1;

    int*** pInt3 = &pInt2;

    printf("\nanInt = %d at address %p\n", anInt, &anInt);

    printf("\npInt1 = &anInt = %p points to anInt whose value = %d\n", pInt1, *pInt1);

    printf("\npInt2 = &pInt1 = %p points to pInt1 whose value = %p\n", pInt2, *pInt2);

    printf("\npInt3 = &pInt2 = %p points to pInt2 whose value = %p\n", pInt3, *pInt3);

    printf("\nAn integer number indirectly referred by pInt1 = *pInt1 = %d\n", *pInt1);

    printf("\nAn integer number indirectly referred by pInt2 = **pInt2 = %d\n", **pInt2);

    printf("\nAn integer number indirectly referred by pInt3 = ***pInt3 = %d\n", ***pInt3);
}
```



Declare and initialize pointers

- **const**
  - A qualifier to inform the compiler that the value of a particular variable should not be modified
  - Recall – Constant variables in Chapter 3

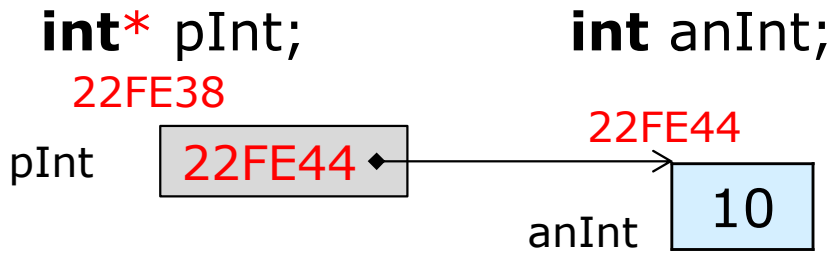
const short MAX = 50;

INVALID!

Modification on constant variable: MAX = 55;

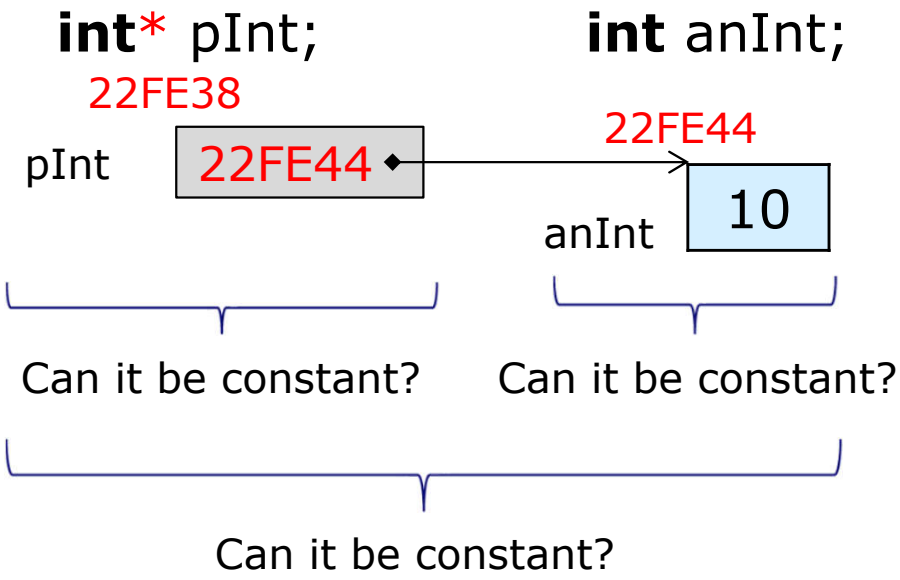
- A pointer of a data type
  - A variable whose value is an address of memory location that contains a value of a data type

How to apply **const** to a pointer?



Declare and initialize pointers

How to apply **const** to a pointer?

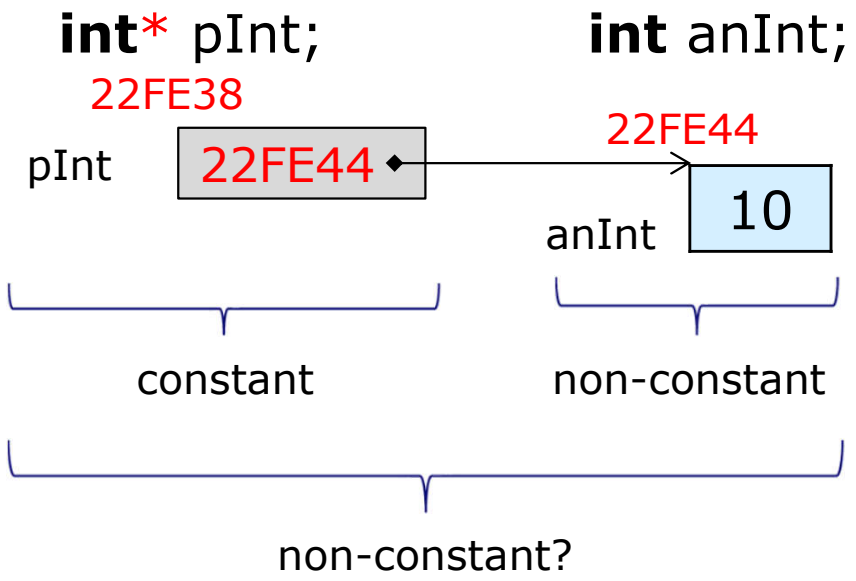


- A non-constant pointer to non-constant data
  - As normal with no **const** qualifier
- A constant pointer to non-constant data
- A non-constant pointer to constant data
- A constant pointer to constant data

Declare and initialize pointers

How to apply **const** to a pointer?

```
int anInt = 10;

int a[5] = {1, 2, 3, 4, 5};

int* const pInt = &anInt;

int* pInt3;
```



Read from right to left: pInt is a constant pointer to an **int** location.

anInt += 5; //OK as modification is on a modifiable variable anInt

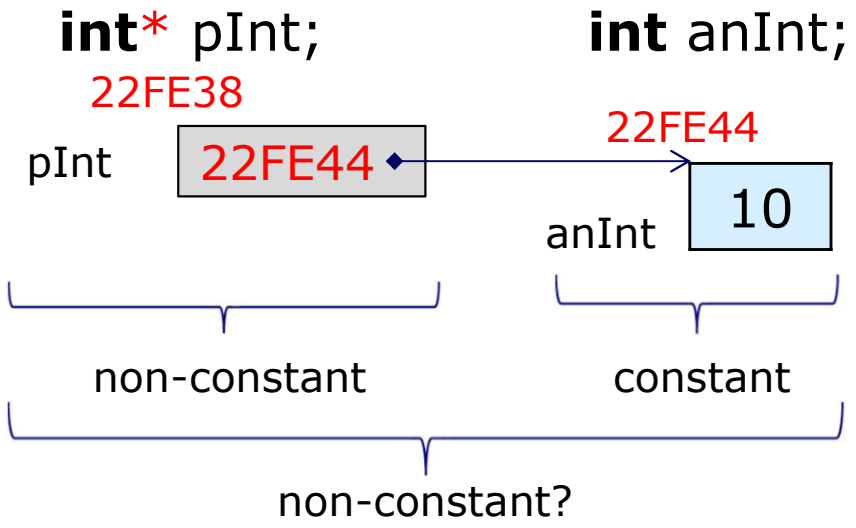
*pInt = anInt + 5; //OK as modification is on a modifiable **int** location.

pInt = &a[1]; //INVALID as modification is on a constant pointer.

pInt3 = pInt; //OK as modification is on a modifiable pointer

Declare and initialize pointers

How to apply **const** to a pointer?




```
int anInt = 10;

int a[5] = {1, 2, 3, 4, 5};

const int* pInt = &anInt;

int* pInt3;
```

Any modification on a constant location through a pointer is not allowed.

Read from right to left: pInt is a pointer to a constant **int** location.

anInt += 5; //OK as modification is on a modifiable variable anInt

*pInt = anInt + 5; //INVALID as modification is on a constant **int** location.

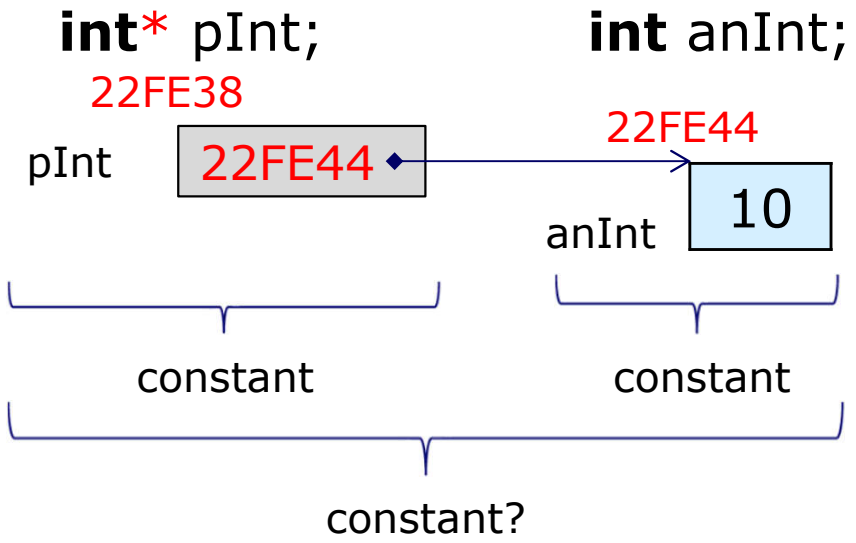
pInt = &a[1]; //OK as modification is on a modifiable pointer.

pInt3 = pInt; //Warning: assignment discards 'const' qualifier from pointer target type

Declare and initialize pointers

How to apply **const** to a pointer?

```
int anInt = 10;

int a[5] = {1, 2, 3, 4, 5};

const int* const pInt = &anInt;

int* pInt3;
```



Any modification on a constant location through a pointer is not allowed.

Read from right to left: pInt is a constant pointer to a constant **int** location.

anInt += 5; //OK as modification is on a modifiable variable anInt

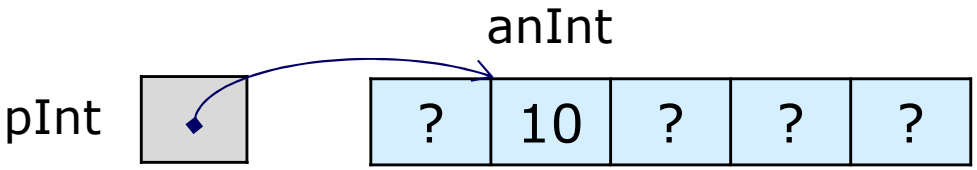
*pInt = anInt + 5; //INVALID as modification is on a constant **int** location.

pInt = &a[1]; //INVALID as modification is on a constant pointer.

pInt3 = pInt; //Warning: assignment discards 'const' qualifier from pointer target type

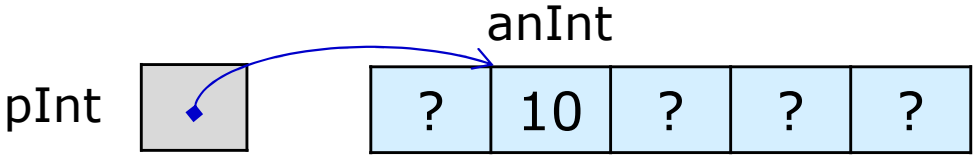
Operations on pointers

```
int anInt = 10;
int* pInt = &anInt;
```



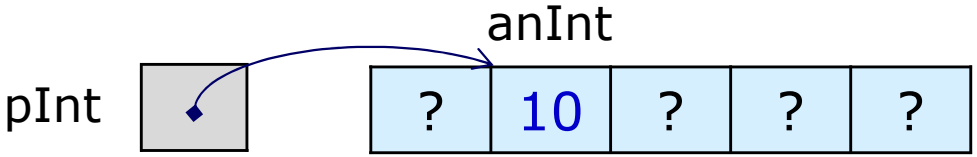
- Get an address of the memory location pointed by a pointer

```
pInt
```



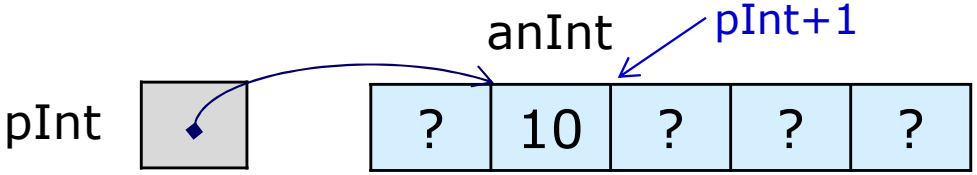
- Get a value in the memory location pointed by a pointer

```
*pInt
```



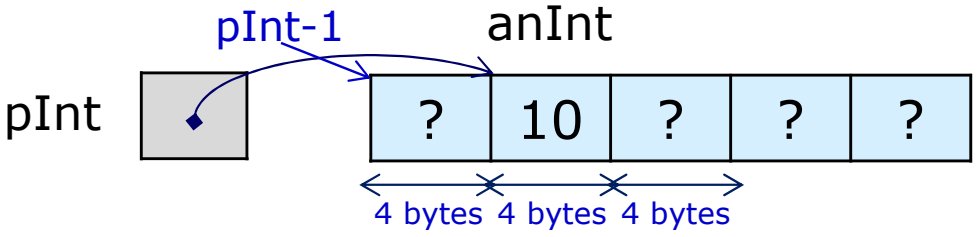
- Get an address of the memory location next to the memory location pointed by a pointer

```
pInt+1
```



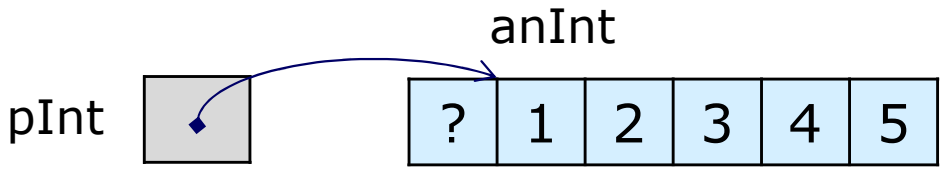
- Get an address of the memory location before the memory location pointed by a pointer

```
pInt-1
```



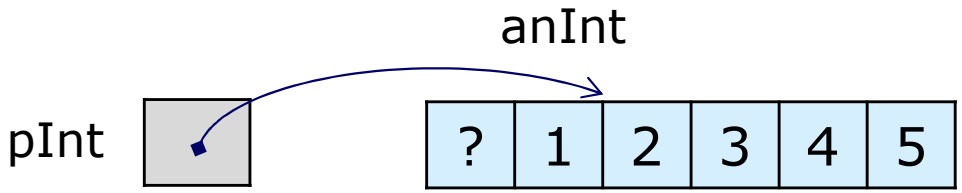
Operations on pointers

```
int anInt[5] = {1, 2, 3, 4, 5};
int* pInt = anInt;
```



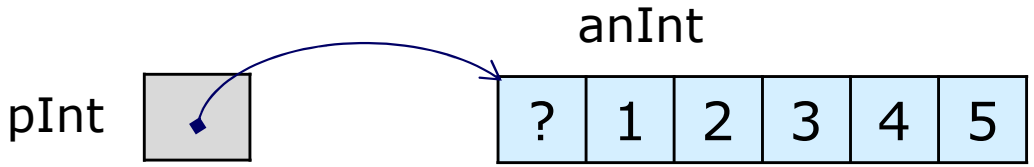
- Shift the pointer to the memory location next to the memory location currently pointed by the pointer

```
pInt++;
++pInt;
```



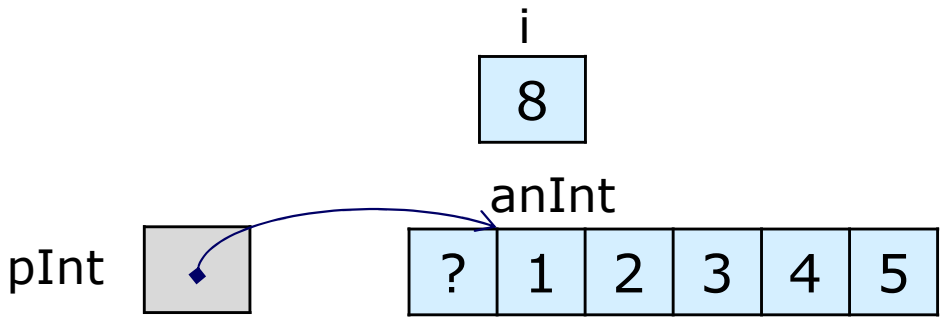
- Move the pointer back to the memory location before the memory location currently pointed by a pointer

```
pInt--;
--pInt;
```



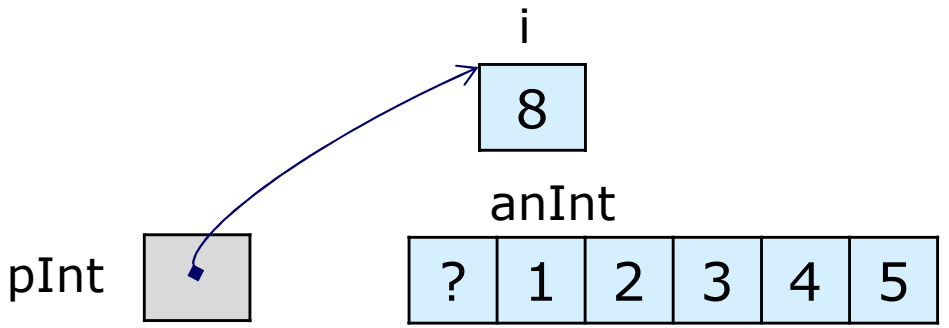
Operations on pointers

```
int i = 8;
int anInt[5] = {1, 2, 3, 4, 5};
int* pInt = anInt;
```

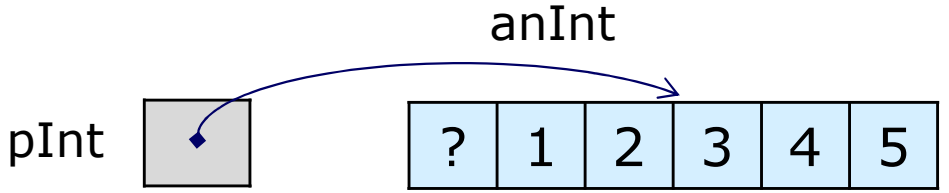


- More assignments

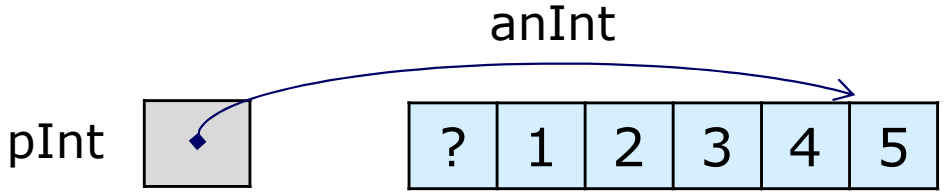
```
pInt = &i;
```




```
pInt = &anInt[2];
```

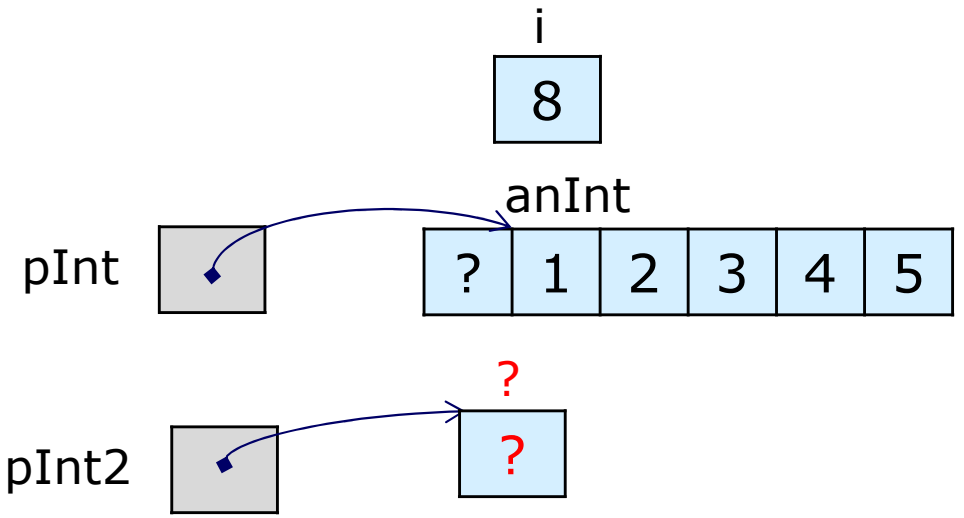



```
pInt += 2;
```



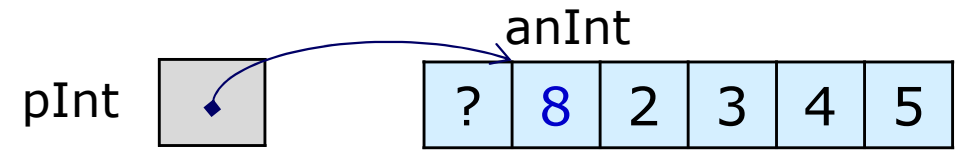
Operations on pointers

```
int i = 8;
int anInt[5] = {1, 2, 3, 4, 5};
int* pInt = anInt;
int* pInt2; //un-initialized!!!
```

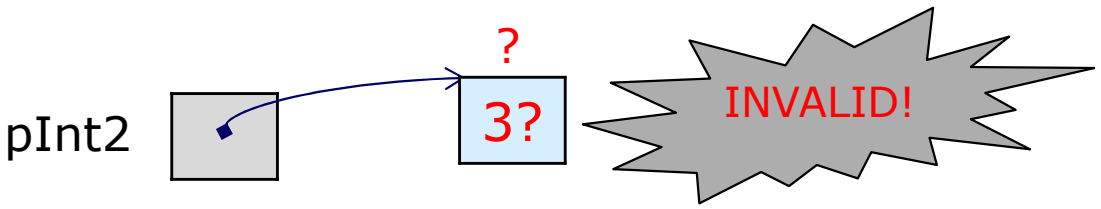


- More assignments

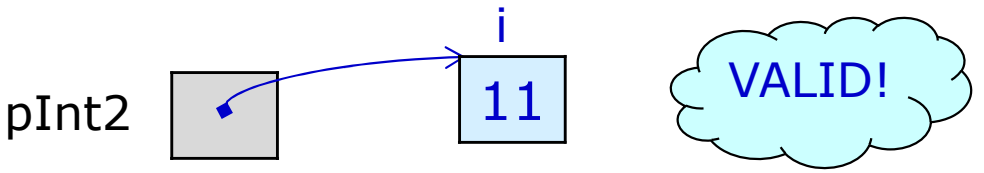
```
*pInt = i;
```




```
*pInt2 = anInt[2];
```

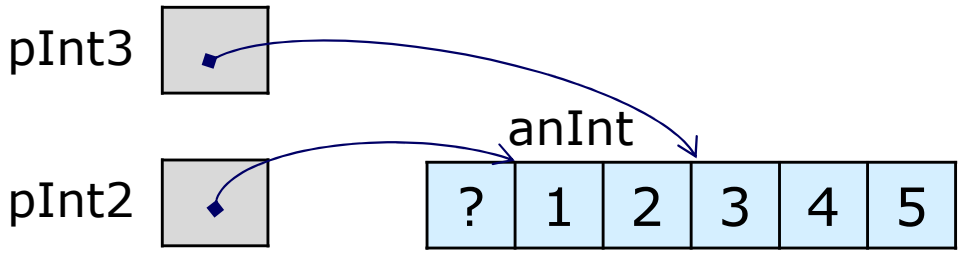



```
pInt2 = &i;
*pInt2 = *pInt + anInt[2];
```



Operations on pointers

```
int anInt[5] = {1, 2, 3, 4, 5};
int* pInt2 = anInt;
int* pInt3 = &anInt[2];
```



- Comparisons

pInt2 <= pInt3	results in:	1 (TRUE)
pInt2 == &anInt[1]	results in:	0 (FALSE)
pInt2 != pInt3 - 1	results in:	1 (TRUE)
pInt2 > anInt	results in:	0 (FALSE)
pInt2 != NULL	results in:	1 (TRUE)
pInt2 != 2	*results in:*	*1 (TRUE) //warning!!!*
pInt2 == 0	results in:	0 (FALSE)

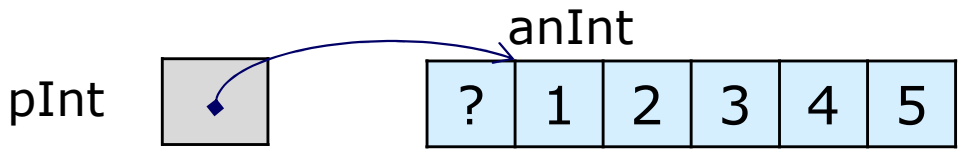
Pointers and arrays

- Arrays
  - A group of contiguous memory locations that all have the same type
  - Array name is the address of the first location among these contiguous memory locations.
    - A constant pointer that points to the first element
- Pointers
  - A variable whose value is an address of memory location that contains a value of a data type

→ Arrays and pointers are closely related and might be used interchangeably.

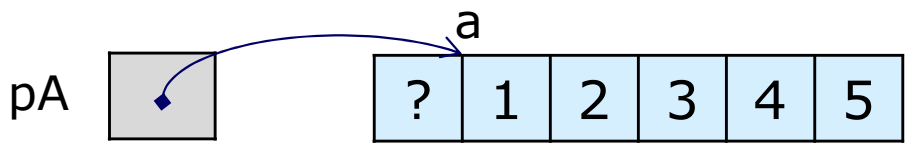
Pointers and arrays

```
int anInt[5] = {1, 2, 3, 4, 5};
int* pInt = anInt;
```



- pInt: returns the address of the first element
  - pInt == &anInt[0] : TRUE
- *pInt or pInt[0]: returns the value of the first element
  - *pInt == pInt[0] == anInt[0] : TRUE
- pInt+i: returns the address of the (i+1)-th element
  - pInt+i == &anInt[i] : TRUE
- *(pInt+i) or pInt[i]: returns the value of the (i+1)-th element
  - *(pInt+i) == pInt[i] == anInt[i] : TRUE

Pointers and arrays



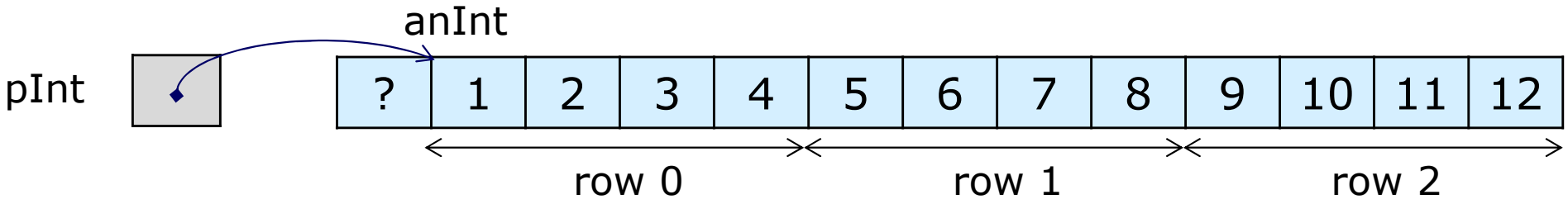

```
int a[5] = {1, 2, 3, 4, 5};

int* pA = a;

int i;

printf("\nArray a[i]\n\n");
for (i=0; i<5; i++) printf("a[%d] = %d\n", i, a[i]);

printf("\nArray pA[i]\n\n");
for (i=0; i<5; i++) printf("pA[%d] = %d\n", i, pA[i]);

printf("\nArray *(a+i)\n\n");
for (i=0; i<5; i++) printf("*(a+%d) = %d\n", i, *(a+i));

printf("\nArray *(pA+i)\n\n");
for (i=0; i<5; i++) printf("*(pA+%d) = %d\n", i, *(pA+i));
```

```
Array a[i]

a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5

Array pA[i]

pA[0] = 1
pA[1] = 2
pA[2] = 3
pA[3] = 4
pA[4] = 5

Array *(a+i)

*(a+0) = 1
*(a+1) = 2
*(a+2) = 3
*(a+3) = 4
*(a+4) = 5

Array *(pA+i)

*(pA+0) = 1
*(pA+1) = 2
*(pA+2) = 3
*(pA+3) = 4
*(pA+4) = 5
```

Equivalent access ways based on indices and addresses

Pointers and arrays

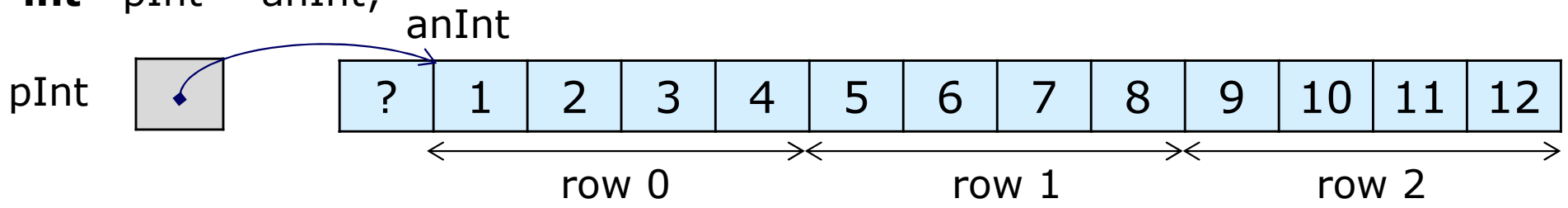
```
int anInt[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
int* pInt = anInt;
```



- pInt: returns the address of the first element at row 0 and column 0
  - pInt == &anInt[0][0] : TRUE
- *pInt: returns the value of the first element at row 0 and column 0
  - *pInt == anInt[0][0] : TRUE

Pointers and arrays

```
int anInt[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
int* pInt = anInt;
```



- pInt + column_size*i + j: returns the address of the element at row i and column j
  - pInt + column_size*i + j == &anInt[i][j] : TRUE
- *(pInt + column_size*i + j): returns the value of the element at row i and column j
  - *(pInt + column_size*i + j) == anInt[i][j] : TRUE

pInt + 4*1 + 1 = pInt + 5: returns &anInt[1][1]

pInt + 4*2 + 1 = pInt + 9: returns &anInt[2][1]

*(pInt+5): returns anInt[1][1], i.e. 6

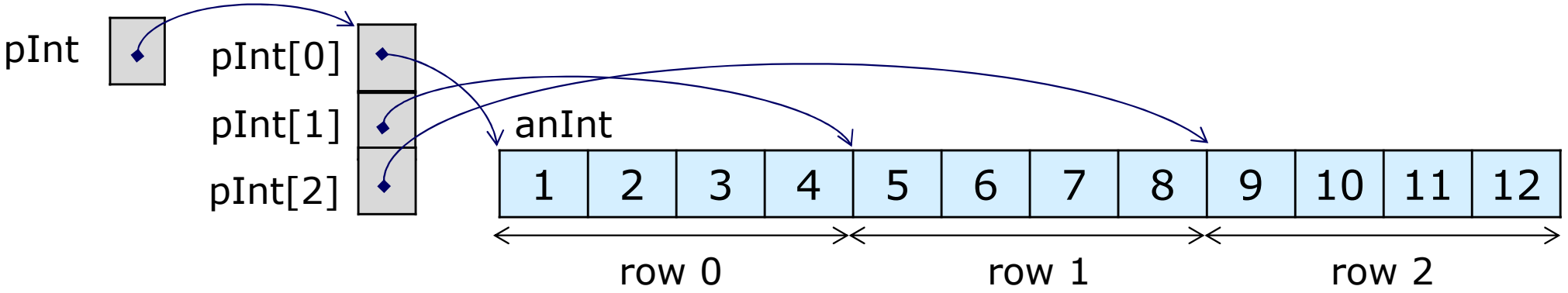
*(pInt+9): returns anInt[2][1], i.e. 10

Pointers and arrays

int anInt[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

int** pInt;

What happens with a pointer to pointer for a multidimensional array?



int** int* int

pInt[0]: returns anInt[0];

pInt[1]: returns anInt[1];

pInt[2]: returns anInt[2];

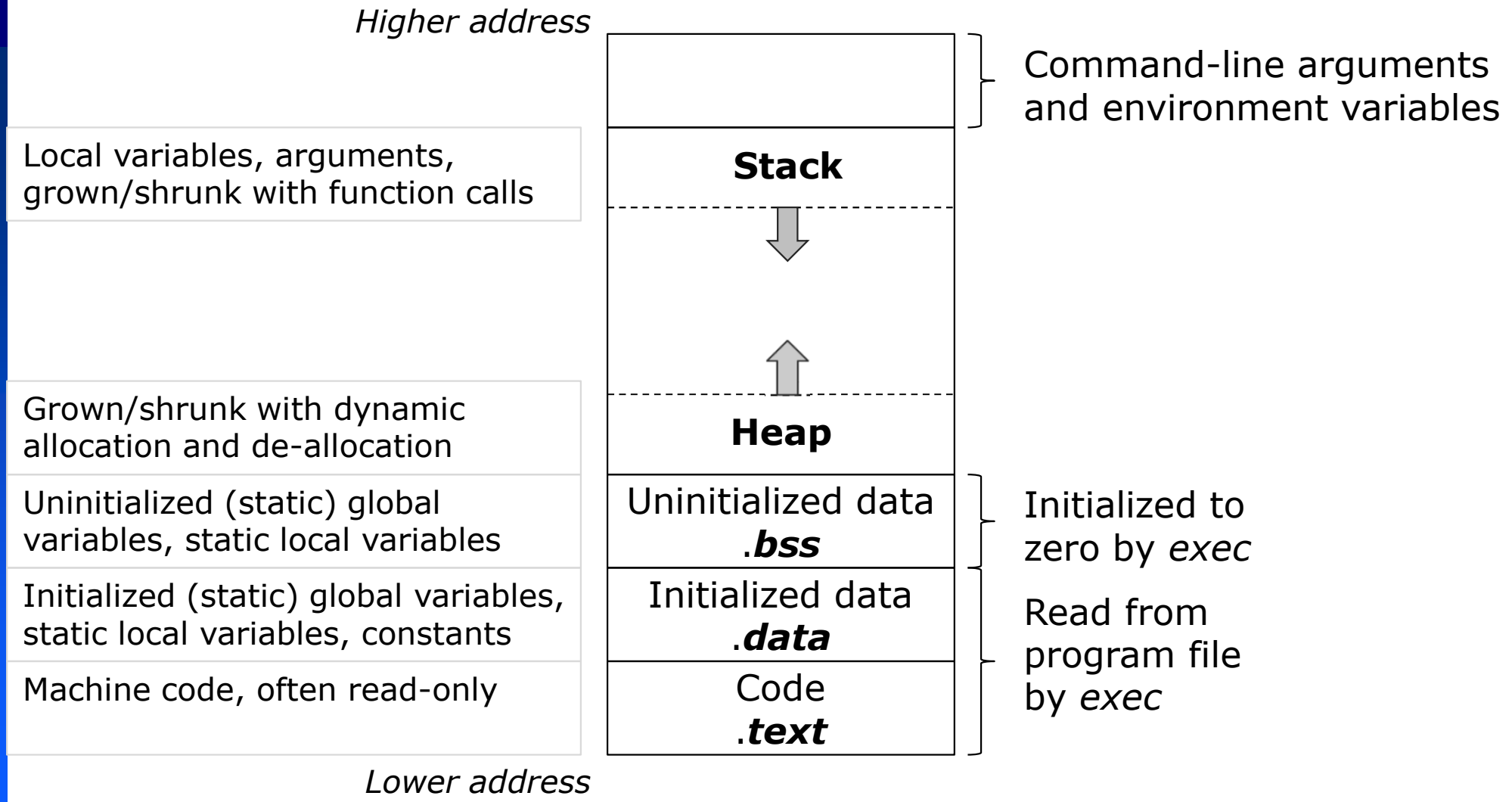
pInt[i][j]: returns anInt[i][j]

NOTE:

- anInt is a constant pointer that points to the first element anInt[0][0].
- pInt is a pointer that points to pointer pInt[0].

Variable storage and heap memory - Recall – Chapter 3

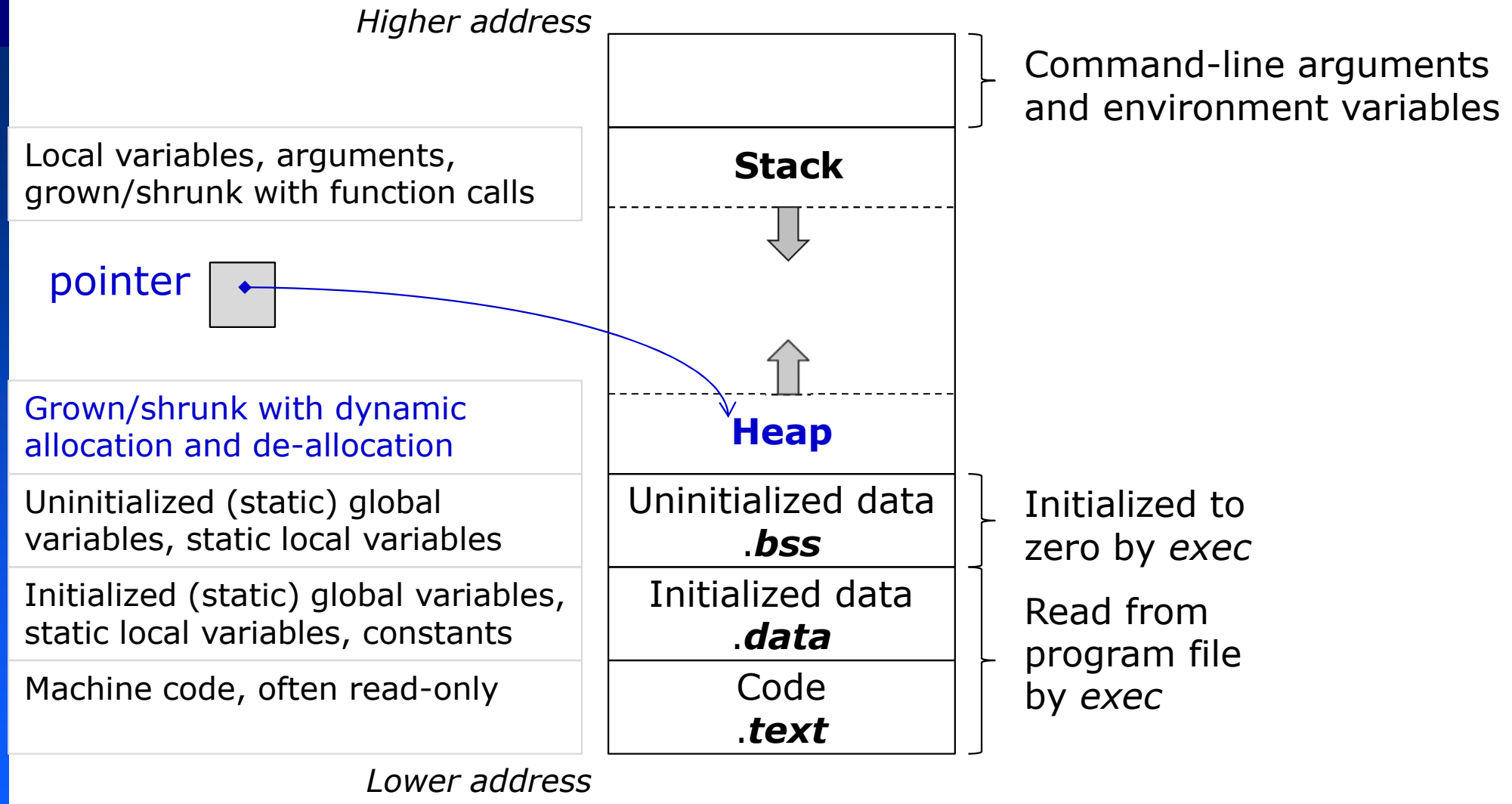
- Memory layout of a C program



bss = block started by symbol, better save space

Variable storage and heap memory

- Memory layout of a C program



bss = block started by symbol, better save space

Memory allocation and de-allocation

- Dynamically allocated arrays
  - Located in heap memory
  - Handled by means of pointers
  - Unknown size at compile time
  - Dynamically grown/shrunk at execution time
  - Allocated/de-allocated explicitly by programmers

Standard library <stdlib.h>

→ Allocation: malloc, calloc, realloc

→ De-allocation: free

Memory allocation and de-allocation

- Allocation
  - void* malloc(size_t size)
    - Allocates the requested memory and returns a pointer to it
  - void* calloc(size_t nitems, size_t size)
    - Allocates the requested memory and returns a pointer to it
  - void* realloc(void* ptr, size_t size)
    - Attemps to resize the memory block pointed to by ptr that was previously allocated with a call to *malloc* or *calloc*
- De-allocation
  - void free(void* ptr)
    - De-allocates the memory previously allocated by a call to *calloc*, *malloc*, or *realloc*

Define an array of 5 integer numbers in heap memory

```c
#include <stdio.h>
#include <stdlib.h>

#define N 5

void main() {

    int* pA;
    int* pB;
    int i;

    pA = (int*)malloc(N*sizeof(int));   ⟶ Allocation
    if (pA == NULL) return;             ⟶ Check if OK

    pB = (int*)calloc(N, sizeof(int));  ⟶ Allocation
    if (pB == NULL) return;             ⟶ Check if OK

    //print pA and pB after allocation
    printf("\npA with malloc after allocation\n");
    for (i=0; i<N; i++) printf("pA[%d] = %d\n", i, *(pA+i));
    printf("\npB with calloc after allocation\n");
    for (i=0; i<N; i++) printf("pB[%d] = %d\n", i, *(pB+i));

    //initialization
    for (i=0; i<N; i++) *(pA+i) = i;
    for (i=0; i<N; i++) pB[i] = i;

    //print pA and pB after initialization
    printf("\npA after initialization\n");
    for (i=0; i<N; i++) printf("pA[%d] = %d\n", i, *(pA+i));
    printf("\npB after initialization\n");
    for (i=0; i<N; i++) printf("pB[%d] = %d\n", i, *(pB+i));

    free(pA); ⟶ De-Allocation
    free(pB); ⟶ De-Allocation
}
```

USE

```
pA with malloc after allocation
pA[0] = 3889488
pA[1] = 0
pA[2] = 3866968
pA[3] = 0
pA[4] = 0

pB with calloc after allocation
pB[0] = 0
pB[1] = 0
pB[2] = 0
pB[3] = 0
pB[4] = 0

pA after initialization
pA[0] = 0
pA[1] = 1
pA[2] = 2
pA[3] = 3
pA[4] = 4

pB after initialization
pB[0] = 0
pB[1] = 1
pB[2] = 2
pB[3] = 3
pB[4] = 4
```

Memory allocation and de-allocation

```
pA = (int*)malloc(N*sizeof(int));
```

Type casting
for the appropriate data type of each value in a memory unit

The number of bytes to be allocated
= the number of memory units * memory unit size

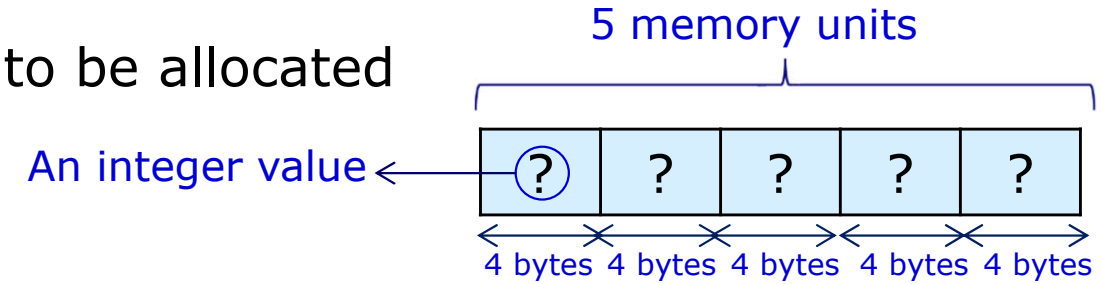
```
pB = (int*)calloc(N, sizeof(int));
```

Type casting
for the appropriate data type of each value in a memory unit

The number of memory units

Memory unit size

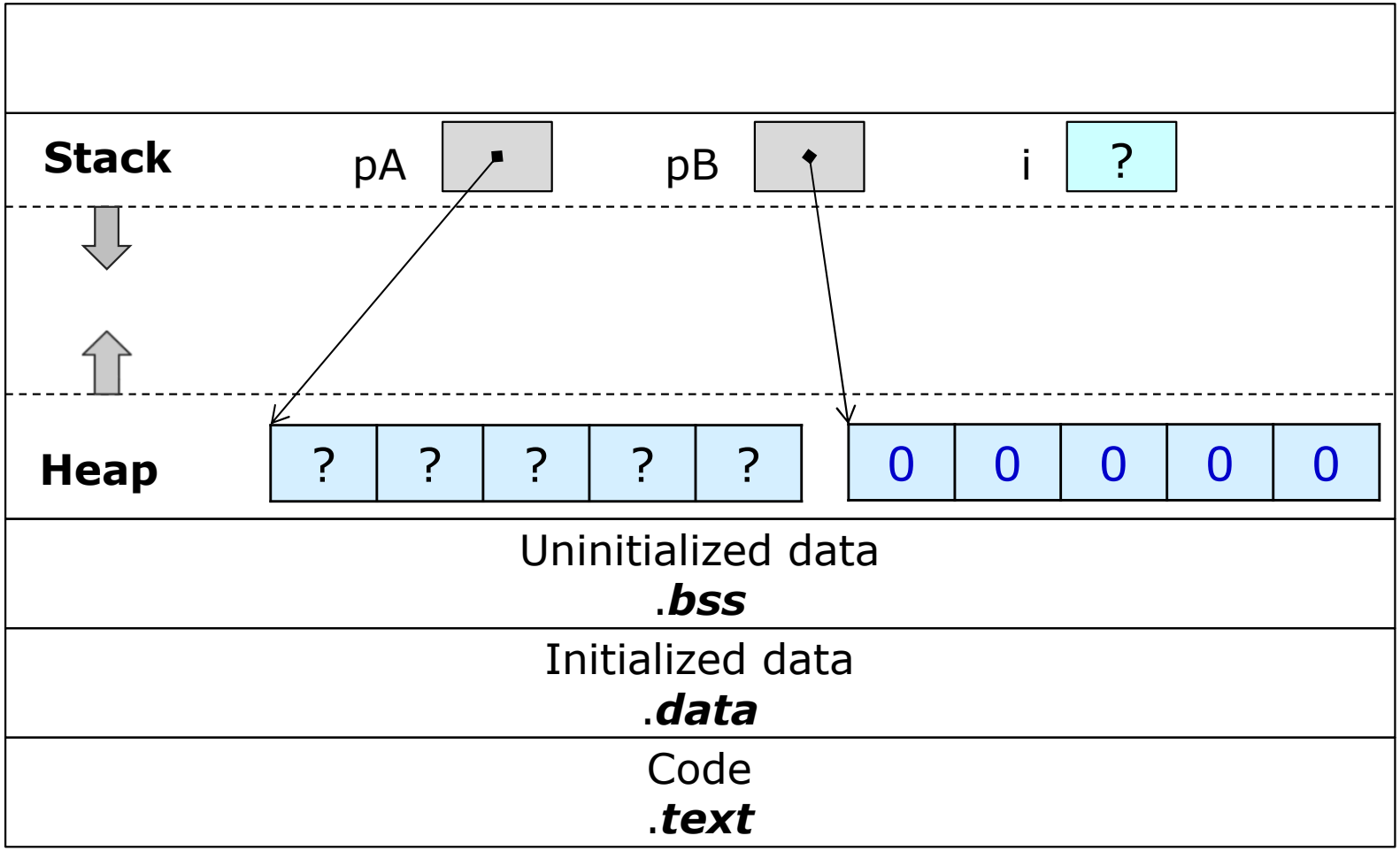
Memory to be allocated



Memory unit size of an integer
= 4 bytes

Memory allocation and de-allocation

```
pA = (int*)malloc(N*sizeof(int));
pB = (int*)calloc(N, sizeof(int));
```



Memory layout

Zero-filled bytes with *calloc*

Calculate an averaged grade of the courses you have taken. It is noted that the number of courses might be different from student to student.

```c
#include <stdio.h>

void main() {

    int n; //the number of courses
    float* yrCourse;
    float avgGrade = 0.0;
    int i;

    do {
        printf("\nEnter the number of your courses n = ");
        scanf("%d", &n);
        fflush(stdin);
    }
    while (n<=0);

    yrCourse = (float*)calloc(n, sizeof(float));

    for (i=0; i<n; i++) {
        printf("\nEnter the %d-th grade = ", i+1);
        scanf("%f", yrCourse+i);
        fflush(stdin);
    }

    printf("\nThe grades of your courses are as follows:\n");
    for (i=0; i<n; i++) {
        printf("\nThe %d-the Grade = %.2f\n", i+1, *(yrCourse+i));
        avgGrade += *(yrCourse+i);
    }
    avgGrade /= n;

    printf("\nYour averaged grade is %.2f.\n", avgGrade);

    free(yrCourse);
}
```

```
Enter the number of your courses n = 3
Enter the 1-th grade = 9.5
Enter the 2-th grade = 10
Enter the 3-th grade = 9.2
The grades of your courses are as follows:
The 1-the Grade = 9.50
The 2-the Grade = 10.00
The 3-the Grade = 9.20
Your averaged grade is 9.57.
```

Memory allocation and de-allocation

```c
#include <stdio.h>
#include <stdlib.h>

#define n 5

void main() {

    int a[n] = {1, 2, 3, 4, 5};

    int* pA = (int*)malloc(n*sizeof(int));

    int i;
    for(i=0; i<n; i++) *(pA+i) = a[i]*2;

    printf("\nArray a[i] in the stack:\n");
    for (i=0; i<n; i++) printf("a[%d] = %d\n", i, a[i]);

    printf("\nArray *(pA+i) in the heap:\n");
    for (i=0; i<n; i++) printf("*(pA+%d) = %d\n", i, *(pA+i));

    free(pA);
}
```

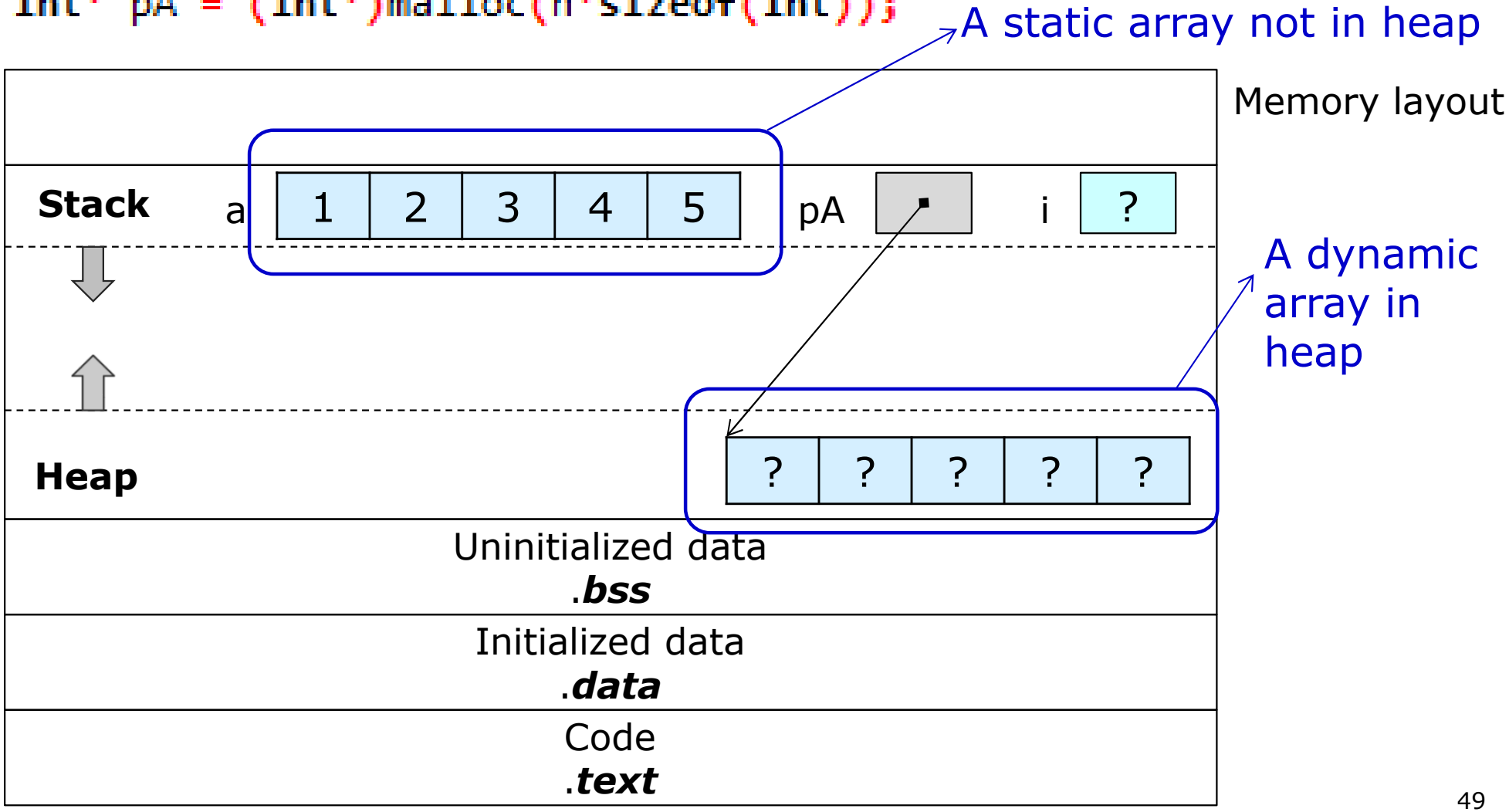
Define an array of integer numbers in the stack: a[i]

Define an array of integer numbers in the heap: *(pA+i)

```
Array a[i] in the stack:
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5

Array *(pA+i) in the heap:
*(pA+0) = 2
*(pA+1) = 4
*(pA+2) = 6
*(pA+3) = 8
*(pA+4) = 10
```

Memory allocation and de-allocation

```
int a[n] = {1, 2, 3, 4, 5};

int* pA = (int*)malloc(n*sizeof(int));
```



Memory allocation and de-allocation

What are the differences?

A static array

- Allow initialization at declaration of array
- Access via array name
  - Possible addresses
- Located not in the heap
- Fixed size thru existence
- No extra storage
- No need to free the memory location
- **sizeof**(array name): array size in bytes

A dynamic array

- No initialization at declaration of pointer
- Access via pointers
  - Possible indices from zero
- Located in the heap
- Varying size with *realloc*
- Extra storage for pointer
- Need to free the memory location via pointer
- **sizeof**(pointer): pointer size, not array size

Memory allocation and de-allocation

- Strings are one-dimension arrays of characters ended by '\0'.
- Strings can be dynamically allocated in the heap memory via pointers.
  - The length of a string is not required to be known in advance.
  - The length of a string can be varied.
  - Allocation: One extra byte is needed for '\0'.

Use: **char*** aString;

Instead of: **char** aString**[10]**;

or **char** aString**[]** = "a string";

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int isSubstring(const char sub[], const char* aStr);

void main() {

    char s1[] = "Computer";
    char s2[] = "Programming";
    char sLang[] = ": C language";

    char* s3 = (char*)calloc(strlen(s1) + 1 + strlen(s2) + 1, sizeof(char));

    if (s3==NULL) return;

    strcpy(s3, s1);

    strcat(s3, " ");

    strcat(s3, s2);

    if (isSubstring(sLang, s3)) printf("\nDone: \"%s\"\n", s3);
    else {
        printf("\nUndone: \"%s\"\n", s3);

        s3 = (char*)realloc(s3, (strlen(s3) + strlen(sLang) + 1)*sizeof(char));

        if (s3==NULL) return;

        strcat(s3, sLang);

        printf("\nDone: \"%s\"\n", s3);
    }

    free(s3);
}
```

s3: a pointer points to an array of characters ended by '\0' in the heap

a dynamic string: an array of characters ended by '\0' in the heap

(**char***)calloc(*length*+1, **sizeof**(**char**))

s3: a pointer points to a dynamic string

a dynamic string: resized in the heap

(**char***)realloc(pointer, (*length*+1)***sizeof**(**char**))

```
Undone: "Computer Programming"
Done: "Computer Programming: C language"
```

char s1[] = "Computer";

char s2[] = "Programming";

Generate s3 as a concatenation of s1 and s2.

If s3 does not contain ": C language",

then extend s3 with ": C language" at the end.

Check if a string sub is a substring of a string aStr:

int isSubstring(**const char** sub[], **const char*** aStr);

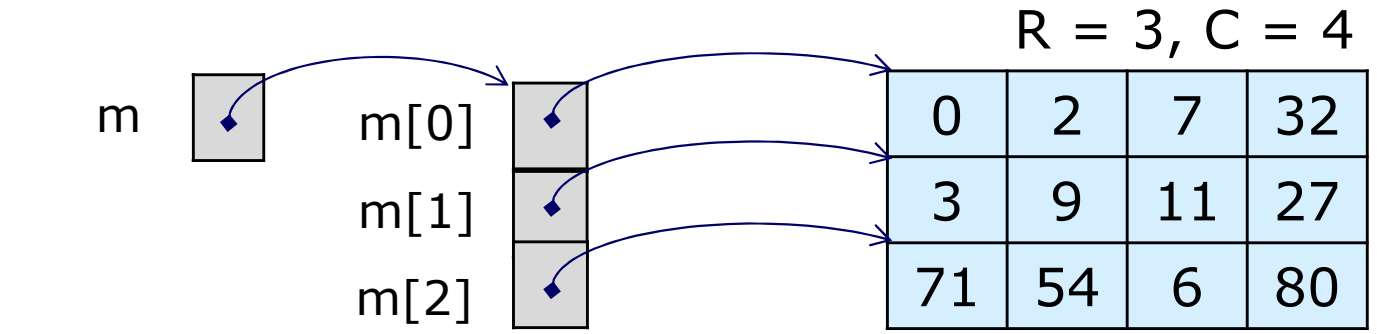
```
//return: 0 for not a substring; 1 for a substring
int isSubstring(const char sub[], const char* aStr) {

    int p=0, sLen=strlen(aStr), subLen=strlen(sub);

    while (p<sLen) {
        if (sub[0] == aStr[p]) {
            int sP = 1;
            while (sP<subLen)
                if (sub[sP] == aStr[p+sP]) sP++;
                else break;
            if (sP==subLen) return 1;
        }
        p++;
    }

    return 0;
}
```

sub = ":C language"
aStr = "Computer Programming"
isSubstring(sub, aStr) returns 0.

sub = ":C language"
aStr = "Computer Programming: C language"
isSubstring(sub, aStr) returns 1.

Memory allocation and de-allocation

- Multidimensional arrays in the heap
  - Generate a dynamic random matrix RxC of integer numbers in the range of [0, 100)
    - R and C corresponding to the number of rows and the number of columns input by a user



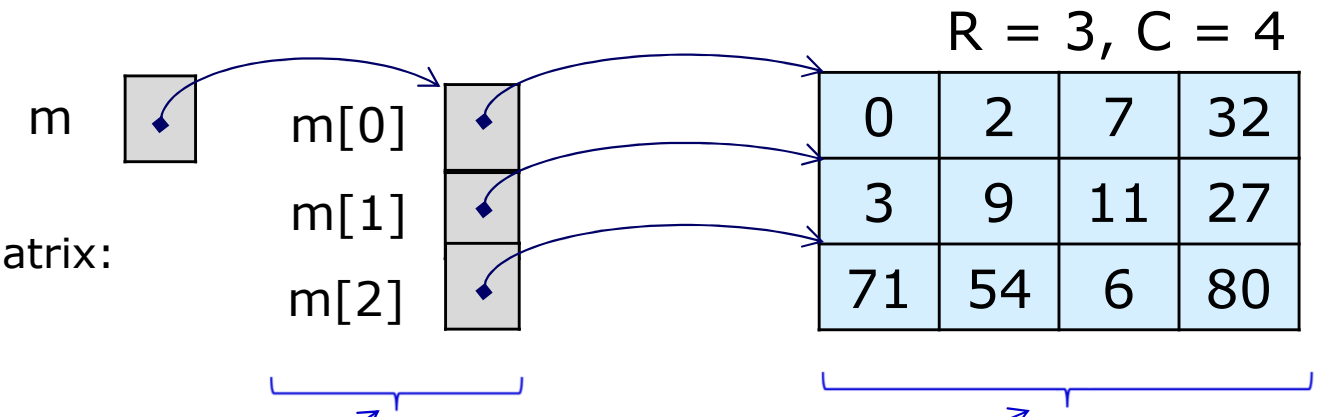

```
int** m;
```

```
int* m[0];
int* m[1];
int* m[2];
```

```
int m[0][0];
int m[0][1];
…
int m[2][3];
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main() {

    int R, C, i, j;
    int** m;

    time_t t;
    srand((unsigned)time(&t));

    do {
        printf("\nEnter row# and column#: ");
        scanf("%d%d", &R, &C);
        fflush(stdin);
    }
    while (R<=0 || C<=0);

    m = (int**)calloc(R, sizeof(int*));

    for (i=0; i<R; i++)
        m[i] = (int*)calloc(C, sizeof(int));

    printf("\nMatrix after allocation\n");
    for (i=0; i<R; i++) {
        for (j=0; j<C; j++) printf("%5d", m[i][j]);
        printf("\n");
    }

    for (i=0; i<R; i++)
        for (j=0; j<C; j++) m[i][j] = (int)rand()%100;

    printf("\nMatrix after randomization\n");
    for (i=0; i<R; i++) {
        for (j=0; j<C; j++) printf("%5d", m[i][j]);
        printf("\n");
    }

    for (i=0; i<R; i++) free(m[i]);

    free(m);
}
```

A pointer to the matrix: pointer to pointer



Memory allocation for pointers to rows

Memory allocation for the elements in each row

Generate a dynamic random matrix RxC of integer numbers in the range of [0, 100)

```
Enter row# and column#: 3 4

Matrix after allocation
    0    0    0    0
    0    0    0    0
    0    0    0    0

Matrix after randomization
   84   60   31   63
   21   44   20   88
   51   49    3   24
```

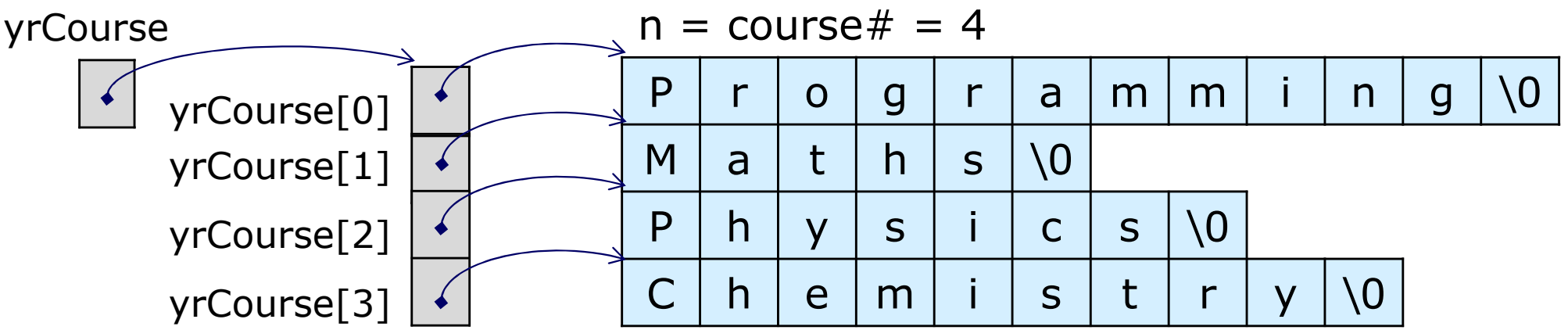
De-allocate the memory location of each row

De-allocate the memory location of pointers to rows

Memory allocation and de-allocation

- Multidimensional arrays in the heap
  - Enter the names of your courses. Your course list might be different from your friend's.
  - Check if any course is input more than once. If yes, simply ignore the duplicate.

```
char** yrCourse;
```




```
char* yrCourse[0];
……
char* yrCourse[3];
```

```
char yrCourse[0][0];
…
char yrCourse[3][6];
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main() {
    int n=0, i=0;
    char** yrCourse;

    while (n<=0) {
        printf("\nEnter the number of your courses: ");
        scanf("%d", &n);
        fflush(stdin);
    }

    yrCourse = (char**)calloc(n, sizeof(char*));

    while (i<n) {
        char aName[10];
        printf("\nEnter the %d-th course name: ", i+1);
        scanf("%s", aName);
        fflush(stdin);

        //check for duplicate
        int j=0;
        while (j<i) {
            if (strcmp(yrCourse[j], aName)==0) break;
            j++;
        }

        //no duplicate, update for a new course
        if (j==i) {
            yrCourse[j] = (char*)calloc(strlen(aName)+1, sizeof(char));
            strcpy(yrCourse[j], aName);
            i++;
        }
    }

    printf("\nYour courses are listed as follows:\n");
    for (i=0; i<n; i++) printf("%d. \"%s\"\n", i+1, yrCourse[i]);

    for (i=0; i<n; i++) free(yrCourse[i]);
    free(yrCourse);
}
```

A pointer to an array of strings: pointer to pointer

Allocate the memory location of the pointers to strings

Allocate the memory location of each string

Deallocate the memory location of each string

Deallocate the memory location of the pointers to strings

Enter the names of your courses. Your course list might be different from your friend's.
Check if any course is input more than once. If yes, simply ignore the duplicate.

```
Enter the number of your courses: 4

Enter the 1-th course name: Programming

Enter the 2-th course name: Maths

Enter the 3-th course name: Physics

Enter the 4-th course name: Maths

Enter the 4-th course name: Chemistry

Your courses are listed as follows:
1. "Programming"
2. "Maths"
3. "Physics"
4. "Chemistry"
```

Memory allocation and de-allocation

- Problems with dynamic memory allocation
  - Multiple allocations for a single pointer
  - Allocation with no explicit de-allocation
    - At multiple levels of details in case of pointers to pointers
  - **sizeof**(pointer) for the number of allocated bytes
  - Reference to the de-allocated memory location
    - A pointer type for a return type of a function
  - Missing one extra byte for '\0' in dynamic strings
  - ...

struct Data Type - Recall

```
struct {
        int x;
        int y;
};
```

```
struct point {
        int x;
        int y;
};
```

```
struct student {
        char        IdStudent[10];
        char        Name[50];
        int         IdMajor;
        int         EntranceYear;
        struct point  Location;
};
```

Structure declaration

```
struct {
        int x;
        int y;
} aPoint1, aPoint2;
```

```
struct point {
        int x;
        int y;
} aPoint4, aPoint5;
```

```
struct student aStudent;
```

```
struct {
        int x;
        int y;
} aPoint3 = {0, 0};
```

```
struct point aPoint6 = {0, 0};
```

Variable declaration

```
aPoint3.x
```

```
aPoint6.y
```

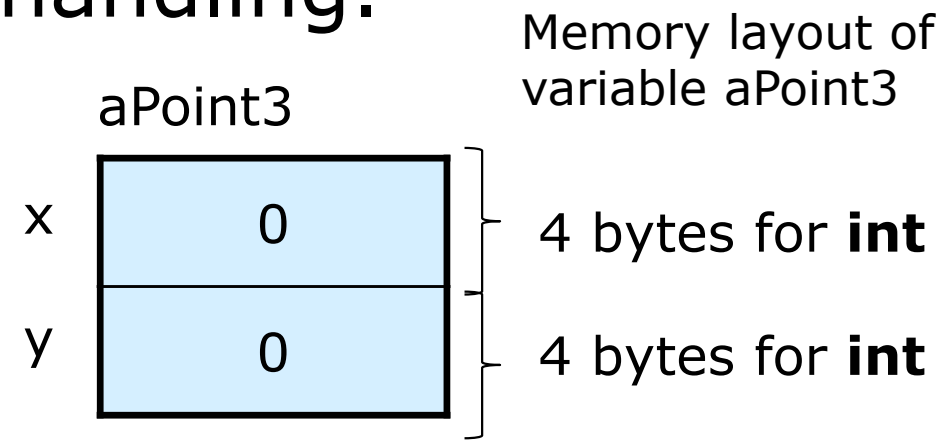
```
aStudent.EntranceYear
aStudent.Location.x
```

Member access

Pointers and structures

- A structure is a collection of one or more variables grouped together under a single name for convenient handling.

```
struct {
        int x;
        int y;
} aPoint3 = {0, 0};
```



aPoint3.x returns 0, an **int** value of member x;

aPoint3.y returns 0, an **int** value of member y;

&aPoint3.x returns an *address* of member x;

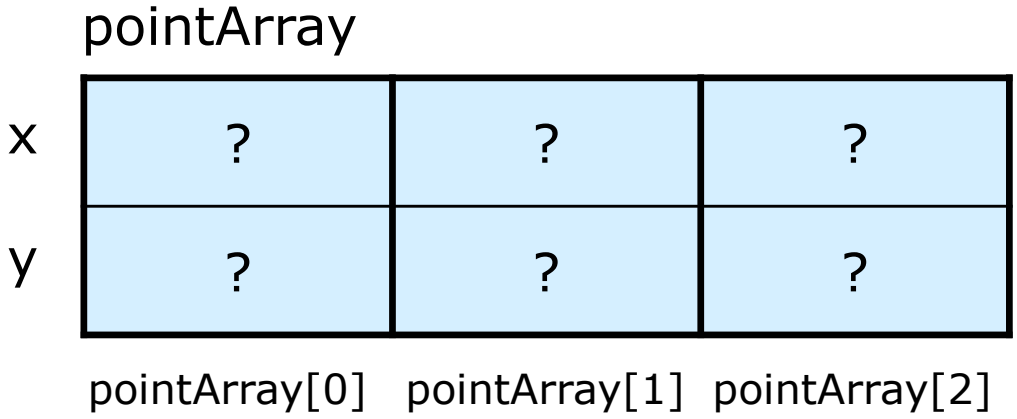
&aPoint3.y returns an *address* of member y;

&aPoint3 returns an address of the variable aPoint3;

Pointers and structures

- An array of structures
  - A group of contiguous memory locations of a **struct** data type

```
struct {
        int x;
        int y;
} pointArray[3];
```



pointArray[0].x returns an **int** value of member x of the first element of array;

pointArray[0].y returns an **int** value of member y of the first element of array;

&pointArray[0].x returns an *address* of member x of the first element of array;

&pointArray[0].y returns an *address* of member y of the first element of array;

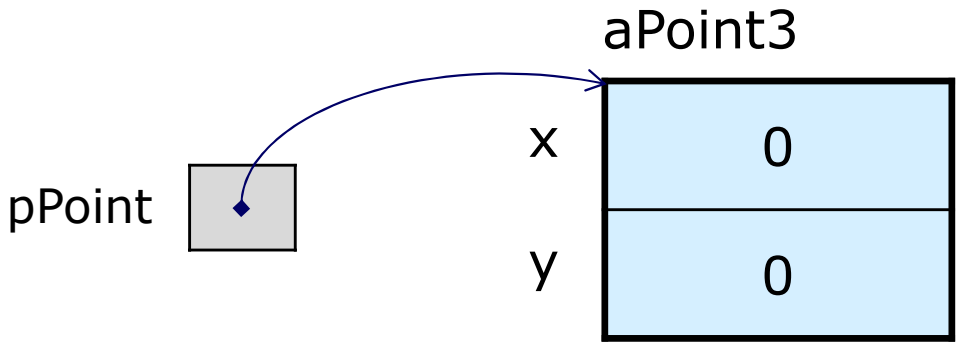
&pointArray[0] returns an address of the first element of array;

&pointArray[1] returns an address of the second element of array; …

Pointers and structures

- A pointer to a structure
  - A variable whose value is an address of a memory location of a **struct** data type

```
struct {
        int x;
        int y;
} * pPoint = &aPoint3;
```



pPoint returns an address of the variable aPoint3;

*pPoint returns a structure value of the variable aPoint3;

(*pPoint).x returns 0, an **int** value of member x;

(*pPoint).y returns 0, an **int** value of member y;

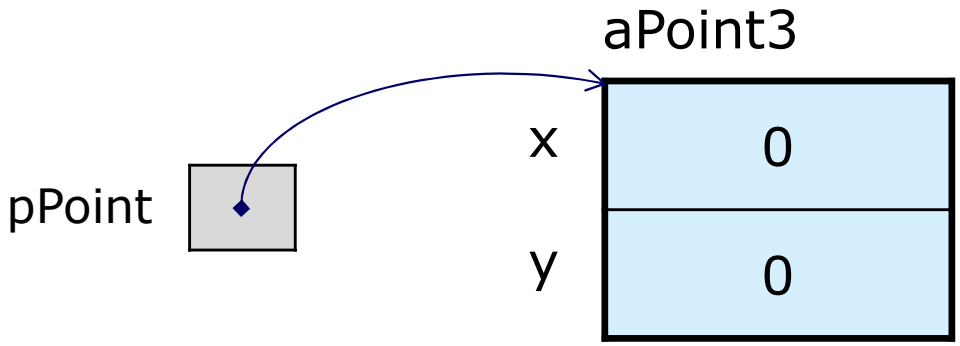
&(*pPoint).x returns an *address* of member x;

&(*pPoint).y returns an *address* of member y;

Pointers and structures

- A pointer to a structure
  - A variable whose value is an address of a memory location of a **struct** data type

```
struct {
        int x;
        int y;
} * pPoint = &aPoint3;
```



Equivalent access:

```
pPoint->x
pPoint->y
```

or

```
(*pPoint).x
(*pPoint).y
```

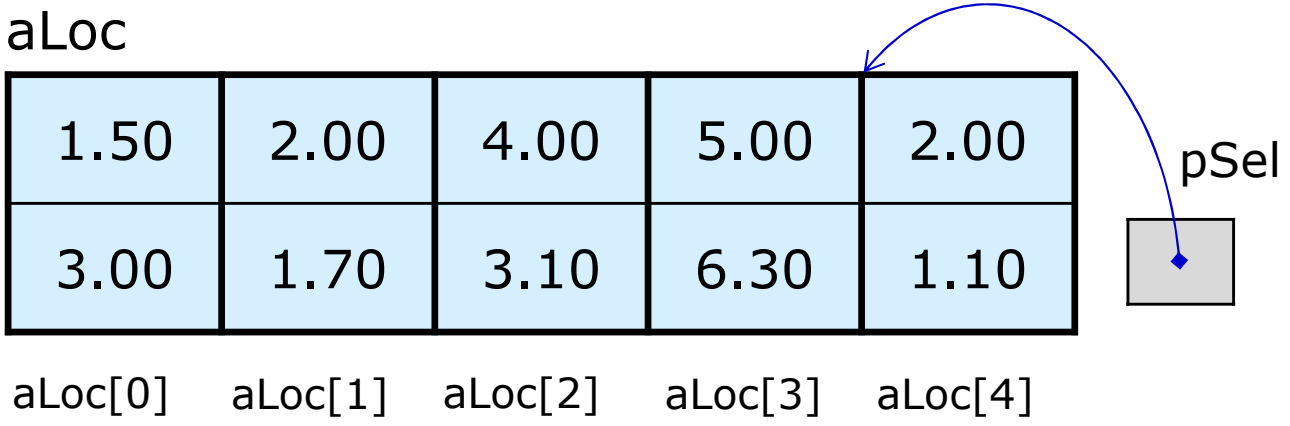
Use **->** for access of a pointer to a member of a structure:

pointer**->***member_of_structure*

Pointers and structures

- Given 5 locations, randomly select one location and then print the selected location and its farthest locations

```
struct location {
        float x;
        float y;
};
```




```
struct location aLoc[n] = {{1.5, 3}, {2, 1.7}, {4, 3.1}, {5, 6.3}, {2, 1.1}};
struct location* pSel = NULL;

pSel = &aLoc[4];
```

pSel->x returns 2.00.

pSel->y returns 1.10.

Given 5 locations, randomly select one location and then print the selected location and its farthest locations

```
Given locations are listed as follows:
The 1-the location: x = 1.50     y = 3.00
The 2-the location: x = 2.00     y = 1.70
The 3-the location: x = 4.00     y = 3.10
The 4-the location: x = 5.00     y = 6.30
The 5-the location: x = 2.00     y = 1.10
Randomly selected location: x = 2.00     y = 1.10
Max distance = 6.00
Farthest location: x = 5.00      y = 6.30
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define n 5

struct location {
    float x;
    float y;
};

void main() {
    struct location aLoc[n] = {{1.5, 3}, {2, 1.7}, {4, 3.1}, {5, 6.3}, {2, 1.1}};
    struct location* pSel = NULL;
    float farLoc[n], maxDis = 0;
    int i;

    printf("\nGiven locations are listed as follows:\n");
    for(i=0; i<n; i++)
        printf("\nThe %d-the location: x = %.2f\t y = %.2f\n", i+1, aLoc[i].x, aLoc[i].y);

    time_t t;
    srand((unsigned)time(&t));

    pSel = &aLoc[(int)rand()%5];

    for(i=0; i<n; i++) {
        float curDis = sqrt((pSel->x-aLoc[i].x)*(pSel->x-aLoc[i].x)+(pSel->y-aLoc[i].y)*(pSel->y-aLoc[i].y));
        if (curDis>maxDis) maxDis = curDis;
        farLoc[i] = curDis;
    }

    printf("\nRandomly selected location: x = %.2f\t y = %.2f\n", pSel->x, pSel->y);
    printf("\nMax distance = %.2f\n", maxDis);
    for(i=0; i<n; i++)
        if (farLoc[i]>=maxDis) printf("\nFarthest location: x = %.2f\t y = %.2f\n", aLoc[i].x, aLoc[i].y);
}
```

Access to member of a structure via an element of array: **.**

Access to member of a structure via pointer: **->**

Pointers and structures

- Dynamic structures in the heap memory
  - Single structure
  - Array of structures

Given n locations, find the nearest locations from your current location.

A pointer to an array of structures in heap

A pointer to a structure in heap

```
struct location * aLoc, * curLoc;

aLoc = (struct location*) calloc(n, sizeof(struct location));

curLoc = (struct location*) calloc(1, sizeof(struct location));
```

Given n locations, find the nearest locations from your current location.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct location {
    float x;
    float y;
};

void main() {
    struct location* aLoc, * curLoc;
    float* aDis, minDis;
    int n=0, i;

    while (n<=0) {
        printf("\nEnter the number of locations: ");
        scanf("%d", &n);
        fflush(stdin);
    }

    aLoc = (struct location*)calloc(n, sizeof(struct location));
    curLoc = (struct location*)calloc(1, sizeof(struct location));
    aDis = (float*)calloc(n, sizeof(float));

    for(i=0; i<n; i++) {
        printf("\nEnter the %d-th location: ", i+1);
        scanf("%f%f", &aLoc[i].x, &aLoc[i].y);
        fflush(stdin);
    }
    printf("\nEnter your current location: ");
    scanf("%f%f", &(curLoc->x), &(curLoc->y));

    aDis[0] = sqrt((curLoc->x-aLoc[0].x)*(curLoc->x-aLoc[0].x)+(curLoc->y-aLoc[0].y)*(curLoc->y-aLoc[0].y));
    minDis = aDis[0];
    for(i=1; i<n; i++) {
        float curDis = sqrt((curLoc->x-aLoc[i].x)*(curLoc->x-aLoc[i].x)+(curLoc->y-aLoc[i].y)*(curLoc->y-aLoc[i].y));
        if (curDis<minDis) minDis = curDis;
        aDis[i] = curDis;
    }

    printf("\nMin distance = %.2f\n", minDis);
    for(i=0; i<n; i++)
        if (aDis[i]<=minDis) printf("\nNearest location: x = %.2f\t y = %.2f\n", aLoc[i].x, aLoc[i].y);

    free(aDis);
    free(curLoc);
    free(aLoc);
}
```

```
Enter the number of locations: 5
Enter the 1-th location: 1.5 3.0
Enter the 2-th location: 2.0 1.7
Enter the 3-th location: 4 3.1
Enter the 4-th location: 5 6.3
Enter the 5-th location: 2 1.1
Enter your current location: 4 5.5
Min distance = 1.28
Nearest location: x = 5.00        y = 6.30
```

Calculate an averaged grade of the courses you have taken. It is noted that the number of courses might be different from student to student.

```c
#include <stdio.h>

struct course {
        char name[10];
        float grade;
    };

void main() {

    int n; //the number of courses
    struct course* yrCourse;
    float avgGrade = 0.0;
    int i;

    do {
        printf("\nEnter the number of your courses n = ");
        scanf("%d", &n);
        fflush(stdin);
    }
    while (n<=0);

    yrCourse = (struct course*)calloc(n, sizeof(struct course));

    for (i=0; i<n; i++) {
        printf("\nEnter the %d-th course name = ", i+1);
        scanf("%s", yrCourse[i].name);
        printf("\nEnter the %d-th grade = ", i+1);
        scanf("%f", &yrCourse[i].grade);
        fflush(stdin);
    }

    printf("\nYour courses are as follows:\n");
    for (i=0; i<n; i++) {
        printf("\nCourse name = %s;\t Grade = %.2f\n", yrCourse[i].name, yrCourse[i].grade);
        avgGrade += yrCourse[i].grade;
    }
    avgGrade /= n;

    printf("\nYour averaged grade is %.2f.\n", avgGrade);

    free(yrCourse);
}
```

```
Enter the number of your courses n = 3
Enter the 1-th course name = Programming
Enter the 1-th grade = 9.5
Enter the 2-th course name = Mathematics
Enter the 2-th grade = 10
Enter the 3-th course name = Chemistry
Enter the 3-th grade = 9.2
Your courses are as follows:
Course name = Programming;        Grade = 9.50
Course name = Mathematics;        Grade = 10.00
Course name = Chemistry;          Grade = 9.20
Your averaged grade is 9.57.
```

Pass pointers to a function

- Address passing to a function
  - Allow call-by-reference
  - Allow multiple input values like arrays
  - Allow multiple returned values

Avoid huge data copy

→ Pointers are passed to a function:

  → Pointers

  → Pointers to pointers

Pass pointers to a function

```c
#include <stdio.h>

void swap(int* a, int* b);

void main() {
    int a=10, b=5;

    int* aPtr = &a;
    int* bPtr = &b;

    printf("\n1. Before swapping: a = %d, b = %d\n", a, b);

    swap(&a, &b);

    printf("\n1. After swapping: a = %d, b = %d\n", a, b);

    printf("\n2. Before swapping: a = %d, b = %d\n", a, b);

    swap(aPtr, bPtr);

    printf("\n2. After swapping: a = %d, b = %d\n", a, b);
}

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Swap a and b:

```
1. Before swapping: a = 10, b = 5

1. After swapping: a = 5, b = 10

2. Before swapping: a = 5, b = 10

2. After swapping: a = 10, b = 5
```

Address passing via variables' addresses

Address passing via pointers that point to the variables

Give an array of n integer numbers, calculate their mean and standard deviation.

A *pointer* to an array of the **int** values

The number of elements in the array

A *pointer* to a **float** memory location for the expected output: standard deviation

A *pointer* to a **float** memory location for the expected output: mean

Address passing with *a*, *&mean*, *&sdev*

```c
#include <stdio.h>

int getStat(int* a, int n, float* mean, float* sdev);

void main() {
    int a[] = {2, 4, -3, 0, 5, 6, 8, 9, 1, -7};
    int n = sizeof(a)/sizeof(int), i;
    float mean, sdev;

    getStat(a, n, &mean, &sdev);

    printf("\nA collection of integer numbers: \n");
    for(i=0; i<n; i++) printf("The %d-th number: %5d\n", i+1, a[i]);

    printf("\n\nmean = %.2f; \tstandard deviation = %.2f\n", mean, sdev);
}

//return: -1 for invalid n; 1 for valid n > 0
int getStat(int* a, int n, float* mean, float* sdev) {

    if (n<=0) return -1;

    int i;

    *mean = 0;
    for (i=0; i<n; i++) *mean += a[i];
    *mean /= n;

    *sdev = 0;
    for (i=0; i<n; i++) *sdev += (a[i]-*mean)*(a[i]-*mean);
    *sdev = sqrt(*sdev/n);

    return 1;
}
```

```
A collection of integer numbers:
The 1-th number:     2
The 2-th number:     4
The 3-th number:    -3
The 4-th number:     0
The 5-th number:     5
The 6-th number:     6
The 7-th number:     8
The 8-th number:     9
The 9-th number:     1
The 10-th number:    -7


mean = 2.50;    standard deviation = 4.72
```

Given n locations (randomly generated) and a current location, check if the current location has already been in the list of n locations. If not, insert the current location into the given list.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct {
    float x;
    float y;
} location;

int isFound(location* locs, int* n, location cur);
location genLoc();

void main() {
    location* lstLoc;
    location curLoc;
    int n = 0, i, check;

    while (n<=0) {
        printf("\nEnter the number of locations: ");
        scanf("%d", &n);
        fflush(stdin);
    }

    time_t t;
    srand((unsigned)time(&t));

    curLoc = genLoc();

    lstLoc = (location*)calloc(n, sizeof(location));

    if (lstLoc == NULL) check = -1;
    else {
        for(i=0; i<n; i++) lstLoc[i] = genLoc();
        check = isFound(lstLoc, &n, curLoc);
    }

    for(i=0; i<n; i++)
        printf("\n%d-th location: x = %6.2f; y = %6.2f\n", i+1, lstLoc[i].x, lstLoc[i].y);

    printf("\nCurrent location: x = %6.2f; y = %6.2f\n\n", curLoc.x, curLoc.y);
    switch(check) {
        case 1: printf("\tFOUND\n"); break;
        case 0: printf("\tNOT FOUND, SUCCESSFULLY INSERTED\n"); break;
        default: printf("\tNOT FOUND, UNSUCCESSFULLY INSERTED\n");
    }

    free(lstLoc);
}
```

A *pointer* to the array of structures location

A *pointer* to the variable n as the number of elements in the array might be changed

```
Enter the number of locations: 8

1-th location: x =   9.70; y =   5.30

2-th location: x =   4.10; y =   7.50

3-th location: x =   6.80; y =   9.60

4-th location: x =   6.00; y =   5.00

5-th location: x =   2.60; y =   5.30

6-th location: x =   8.80; y =   7.70

7-th location: x =   8.90; y =   6.90

8-th location: x =   7.60; y =   0.70

9-th location: x =   0.30; y =   1.70

Current location: x =   0.30; y =   1.70

        NOT FOUND, SUCCESSFULLY INSERTED
```

Given n locations (randomly generated) and a current location, check if the current location has already been in the list of n locations. If not, insert the current location into the given list.

```
//return:
//      1: found;
//      0: not found and successfully inserted;
//      -1: not found and unsuccessfully inserted
int isFound(location* locs, int* n, location cur) {

    int i;
    for (i=0; i<*n; i++)
        if (locs[i].x == cur.x && locs[i].y == cur.y) return 1;

    locs = (location*)realloc(locs, (*n+1)*sizeof(location));

    if (locs==NULL) return -1;

    locs[*n].x = cur.x;
    locs[*n].y = cur.y;
    (*n)++;

    return 0;
}

//return: a new structure location
location genLoc() {

    location b = {((int)rand()%100)/10.0, ((int)rand()%100)/10.0};

    return b;
}
```

MATRIX MULTIPLICATION:

mulM = aMxbM

- Sizes of aM and bM are given.
- Elements of aM and bM are randomly generated.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int** mulMatrix(int** m1, int r1, int c1, int** m2, int r2, int c2);
int** genMatrix(int r, int c);
int getNaturalNumber(char* purpose);
void printMatrix(int** m, int r, int c);

void main() {

    int** aM;
    int** bM;
    int** mulM;
    int ar, ac, br, bc, i;

    time_t t;
    srand((unsigned)time(&t));

    ar = getNaturalNumber("row# of matrix a");
    ac = getNaturalNumber("column# of matrix a");
    br = getNaturalNumber("row# of matrix b");
    bc = getNaturalNumber("column# of matrix b");

    aM = genMatrix(ar, ac);
    bM = genMatrix(br, bc);

    printf("\nMatrix a:\n");
    printMatrix(aM, ar, ac);

    printf("\nMatrix b:\n");
    printMatrix(bM, br, bc);

    mulM = mulMatrix(aM, ar, ac, bM, br, bc);

    if (mulM==NULL) {
        printf("\nUNABLE FOR MATRIX MULTIPLICATION!\n");
        return;
    }

    printf("\nThe Resulting Matrix:\n");
    printMatrix(mulM, ar, bc);

    for (i=0; i<ar; i++) {
        free(aM[i]);
        free(mulM[i]);
    }
    for (i=0; i<br; i++) free(bM[i]);
    free(aM);
    free(bM);
    free(mulM);
}
```

```
Enter a natural number for row# of matrix a: 3
Enter a natural number for column# of matrix a: 2
Enter a natural number for row# of matrix b: 2
Enter a natural number for column# of matrix b: 4

Matrix a:
        62        86
        67        11
        20        54

Matrix b:
        81        26        28        64
        60        62        89        95

The Resulting Matrix:
     10182      6944      9390     12138
      6087      2424      2855      5333
      4860      3868      5366      6410
```

Function for matrix multiplication: res = m1xm2

```
//return: res: success; NULL: unsuccess
int** mulMatrix(int** m1, int r1, int c1, int** m2, int r2, int c2) {

    static int** res;
    int i, j, k;

    if (r1<=0 || c1<=0 || r2<=0 || c2<=0 || c1 != r2) return NULL;

    res = (int**)calloc(r1, sizeof(int*));
    if (res==NULL) return NULL;

    for (i=0; i<r1; i++) {
        res[i] = (int*)calloc(c2, sizeof(int));
        if (res[i]==NULL) return NULL;
    }

    for(i=0; i<r1; i++)
        for (j=0; j<c2; j++) {
            res[i][j] = 0;
            for (k=0; k<c1; k++) res[i][j] += m1[i][k]*m2[k][j];
        }

    return res;
}
```

Function for generating a matrix m with size rxc

```
//return: m: success; NULL: unsuccess
int** genMatrix(int r, int c) {

    static int** m;

    int i,j;

    if (r<=0 || c<=0) return NULL;

    m = (int**) calloc(r, sizeof(int*));
    if (m==NULL) return NULL;

    for (i=0; i<r; i++) {
        m[i] = (int*) calloc(c, sizeof(int));
        if (m[i]==NULL) return NULL;
    }

    for (i=0; i<r; i++)
        for (j=0; j<c; j++) m[i][j] = (int)rand()%100;

    return m;
}
```

Functions for inputting a natural number and printing a matrix

```
int getNaturalNumber(char* purpose) {
    int n;
    do {
        printf("\nEnter a natural number for %s: ", purpose);
        scanf("%d", &n);
        fflush(stdin);
    }
    while (n<=0);
    return n;
}

void printMatrix(int** m, int r, int c) {
    int i, j;

    if (r<=0 || c<=0) return;

    for (i=0; i<r; i++) {
        for (j=0; j<c; j++) printf("%10d", m[i][j]);
        printf("\n");
    }
}
```

Function pointers

- Function name is the starting address of the function memory where the code that performs the function's task exists.
- A function pointer is a pointer that points to the function memory location.
  - Containing an address of the function in memory
  - Able to be passed to functions, returned from functions, stored in arrays, assigned to other function pointers in initialization

→ A means for the flexible execution of the program when calling functions via function pointers instead of their function names

Function pointers

- Declaration based on function prototype

return_type **(***pointer_name***)(***argument_list***)** $=_{opt}$ initial_value$_{opt}$;

  - *pointer_name*: an identifier for a pointer that points to a function
  - *return_type*: a data type of the returned value of the function
  - *argument_list*: list of arguments (specifically data types) separated by comma for formal parameters of the function

- Values for initialization and assignment
  - NULL
  - Function names
  - Function pointers
- Function call through a function pointer instead of a function name

Function pointers

- Declaration based on function prototype

return_type **(***pointer_name***)(***argument_list***)** $=_{opt}$ initial_value$_{opt}$;

```
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
```

```
int (*op)(int, int) = add;
int (*opList1[])(int, int) = {add, sub, mul};
int (*opList2[2])(int, int) = {add, sub};
```

A function call to *add* function:

```
add(1, 2);
(*op)(1, 2);
op(1, 2);
(*opList1[0])(1, 2);
opList1[0](1, 2);
(*opList2[0])(1, 2);
opList2[0](1, 2);
```

A text-based menu-driven program for calculation of two integer numbers

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_INT 2147483647

int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div2(int a, int b);
int mod(int a, int b);

void main() {
    int (*maths[])(int, int) = {add, sub, mul, div2, mod};
    int aChoice, num1=0, num2=0;
    char isAgain;

    do {
        do {
            system("cls");
            printf("\nCalculation for a and b:\n");
            printf("\n1. Addition\n");
            printf("\n2. Subtraction\n");
            printf("\n3. Multiplication\n");
            printf("\n4. Division\n");
            printf("\n5. Remainder\n");
            printf("\nYour choice is: ");
            scanf("%d", &aChoice);
            fflush(stdin);
            printf("\na = ");
            scanf("%d", &num1);
            fflush(stdin);
            printf("\nb = ");
            scanf("%d", &num2);
            fflush(stdin);
        }
        while (aChoice > 5 || aChoice < 1);

        (*maths[aChoice-1])(num1, num2);

        printf("\nDo you want to continue? (Y/N) ");
        scanf("%c", &isAgain);
        fflush(stdin);
    }
    while (isAgain!='N');
}
```

An array of function pointers

```
Calculation for a and b:

1. Addition

2. Subtraction

3. Multiplication

4. Division

5. Remainder

Your choice is: 3

a = -3

b = -71

(-3) * (-71) = 213

Do you want to continue? (Y/N) _
```

A call to an appropriate function via a function pointer in the array of function pointers

```c
int add(int a, int b){
    if (a<0 && b>0) printf("\n(%d) + %d = %d\n", a, b, a+b);
    else if (a>0 && b<0) printf("\n%d + (%d) = %d\n", a, b, a+b);
    else if (a<0 && b<0) printf("\n(%d) + (%d) = %d\n", a, b, a+b);
    else printf("\n%d + %d = %d\n", a, b, a+b);
    return a+b;
}
int sub(int a, int b) {
    if (a<0 && b>0) printf("\n(%d) - %d = %d\n", a, b, a-b);
    else if (a>0 && b<0) printf("\n%d - (%d) = %d\n", a, b, a-b);
    else if (a<0 && b<0) printf("\n(%d) - (%d) = %d\n", a, b, a-b);
    else printf("\n%d - %d = %d\n", a, b, a-b);
    return a-b;
}
int mul(int a, int b) {
    if (a<0 && b>0) printf("\n(%d) * %d = %d\n", a, b, a*b);
    else if (a>0 && b<0) printf("\n%d * (%d) = %d\n", a, b, a*b);
    else if (a<0 && b<0) printf("\n(%d) * (%d) = %d\n", a, b, a*b);
    else printf("\n%d * %d = %d\n", a, b, a*b);
    return a*b;
}
int div2(int a, int b) {
    if (a<0 && b>0) printf("\n(%d) / %d = %d\n", a, b, a/b);
    else if (a>0 && b<0) printf("\n%d / (%d) = %d\n", a, b, a/b);
    else if (a<0 && b<0) printf("\n(%d) / (%d) = %d\n", a, b, a/b);
    else printf("\n%d / %d = %d\n", a, b, a/b);
    return a/b;
}
int mod(int a, int b) {
    if (a<0 && b>0) printf("\n(%d) %% %d = %d\n", a, b, a%b);
    else if (a>0 && b<0) printf("\n%d %% (%d) = %d\n", a, b, a%b);
    else if (a<0 && b<0) printf("\n(%d) %% (%d) = %d\n", a, b, a%b);
    else printf("\n%d %% %d = %d\n", a, b, a%b);
    return a%b;
}
```

Function definitions for *addition*, *subtraction*, *multiplication*, *division*, and *remainder* corresponding to

(*maths[0])(int, int),
(*maths[1])(int, int),
(*maths[2])(int, int),
(*maths[3])(int, int),
(*maths[4])(int, int)

Print a matrix

```c
#include <stdio.h>

void printMatrix(int m[][4], int r, int c, void (*print)(int a[], int n));
void printASC(int a[], int n);
void printDES(int a[], int n);

void main() {

    int m[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

    printf("\nPrint each row with indices in descending order:\n");
    printMatrix(m, 3, 4, printDES);
    printf("\n");

    printf("\nPrint each row with indices in ascending order:\n");
    printMatrix(m, 3, 4, printASC);
    printf("\n");
}

void printMatrix(int m[][4], int r, int c, void (*print)(int a[], int n)) {
    int i;
    for (i=0; i<r; i++) (*print)(m[i], c);
}

void printASC(int a[], int n) {
    int i;
    for (i=0; i<n; i++) printf("%d\t", a[i]);
    printf("\n");
}

void printDES(int a[], int n) {
    int i;
    for (i=n-1; i>=0; i--) printf("%d\t", a[i]);
    printf("\n");
}
```

A formal parameter is a function pointer print:

- Input: an array of integer numbers and an integer number
- Output: void

Address of a function is passed as an actual parameter in function call

Call a function through a function pointer instead of its name

```
Print each row with indices in descending order:
4       3       2       1
8       7       6       5
12      11      10      9


Print each row with indices in ascending order:
1       2       3       4
5       6       7       8
9       10      11      12
```

Summary

- Pointers
  - Support for manipulation on physical memory
  - Simulation for pass-by-reference
  - Enabling dynamic data structures in heap memory
    - On demand
    - Size-varying
- Allocation: calloc, malloc, realloc
- De-allocation: free

Summary

- Operations on pointers
  - Addresses vs. non-address values
- Pointers and arrays, structures, functions
- Pointers and other issues
  - Constant
  - Pointers to void
  - Pointers to pointers
  - Dangling references
  - Function pointers

Chapter 8: Pointers