TRƯỜNG ĐẠI HỌC FPT

Pointers

Objectives

After studying this chapter, you should be able to:

- Understand where program's data can be putted
- Explain what are pointers
- Declare pointers in a program
- Discuss about where pointers can be used
- Understand operators on pointers
- Implement functions in which pointers are parameters
- Use build-in functions to allocate data dynamically

Contents

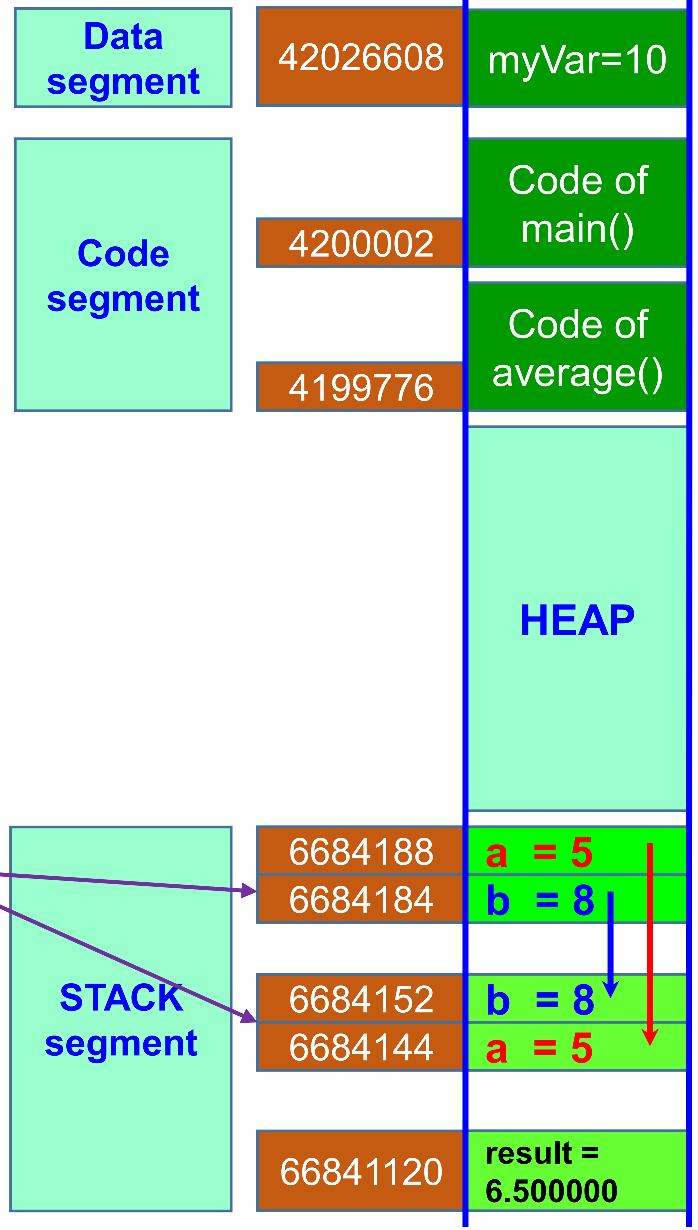
- Review the memory structure of a program
- Where can we put program's data?
- What are pointers?
- Pointer Declarations
- Why are pointers used?
- Pointer operators
- Assign values to pointers
- Access data through pointer
- Explain pointer arithmetic
- Explain pointer comparisons
- Pointers as parameters of a function
- Dynamic Allocated Data

1 - Review the memory structure of a program

```c
#include <stdio.h>

int myVar = 10;

double average(int a, int b){
    double result;
    result = (a+b)/2.0;

    printf("\nIn average function\n");
    printf("%-15s %-15s %-15s\n", "Name", "Address", "Value");
    printf("----------------------------------------\n");
    printf("%-15s %-15u %-15d\n", "a", &a, a);
    printf("%-15s %-15u %-15d\n", "b", &b, b);
    printf("%-15s %-15u %-15lf\n", "result", &result, result);

    return result;
}

int main(){
    int a=5, b=8;

    printf("In main function\n");
    printf("%-15s %-15s %-15s\n", "Name", "Address", "Value");
    printf("----------------------------------------\n");
    printf("%-15s %-15u %-15d\n", "myVar", &myVar, myVar);
    printf("%-15s %-15u %-15d\n", "a", &a, a);
    printf("%-15s %-15u %-15d\n", "b", &b, b);

    printf("Address of main(): %u\n", &main);
    printf("Address of average(...): %u\n", &average);
    printf("Result returned to main: %lf", average(a, b));

    return 0;
}
```

Memory mapping

```
In main function
Name            Address         Value
----------------------------------------
myVar           4206608         10
a               6684188         5
b               6684184         8
Address of main(): 4200002
Address of average(...): 4199776

In average function
Name            Address         Value
----------------------------------------
a               6684144         5
b               6684152         8
result          6684120         6.500000
Result returned to main: 6.500000
```

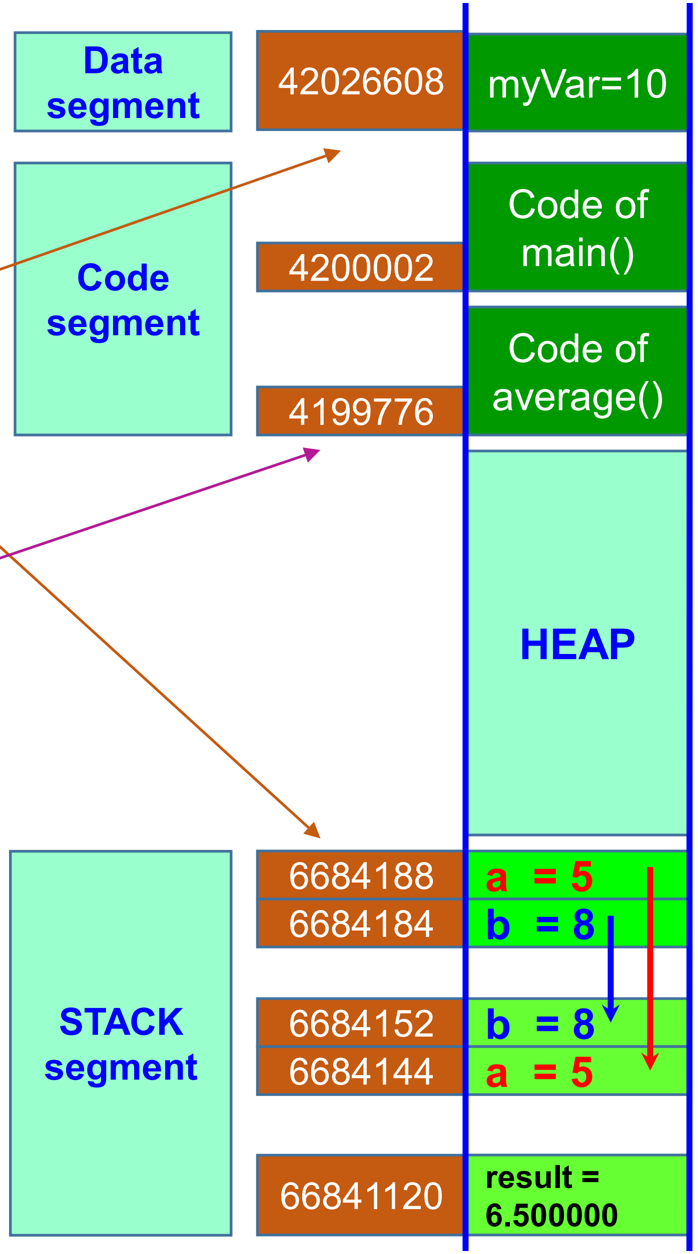


Question

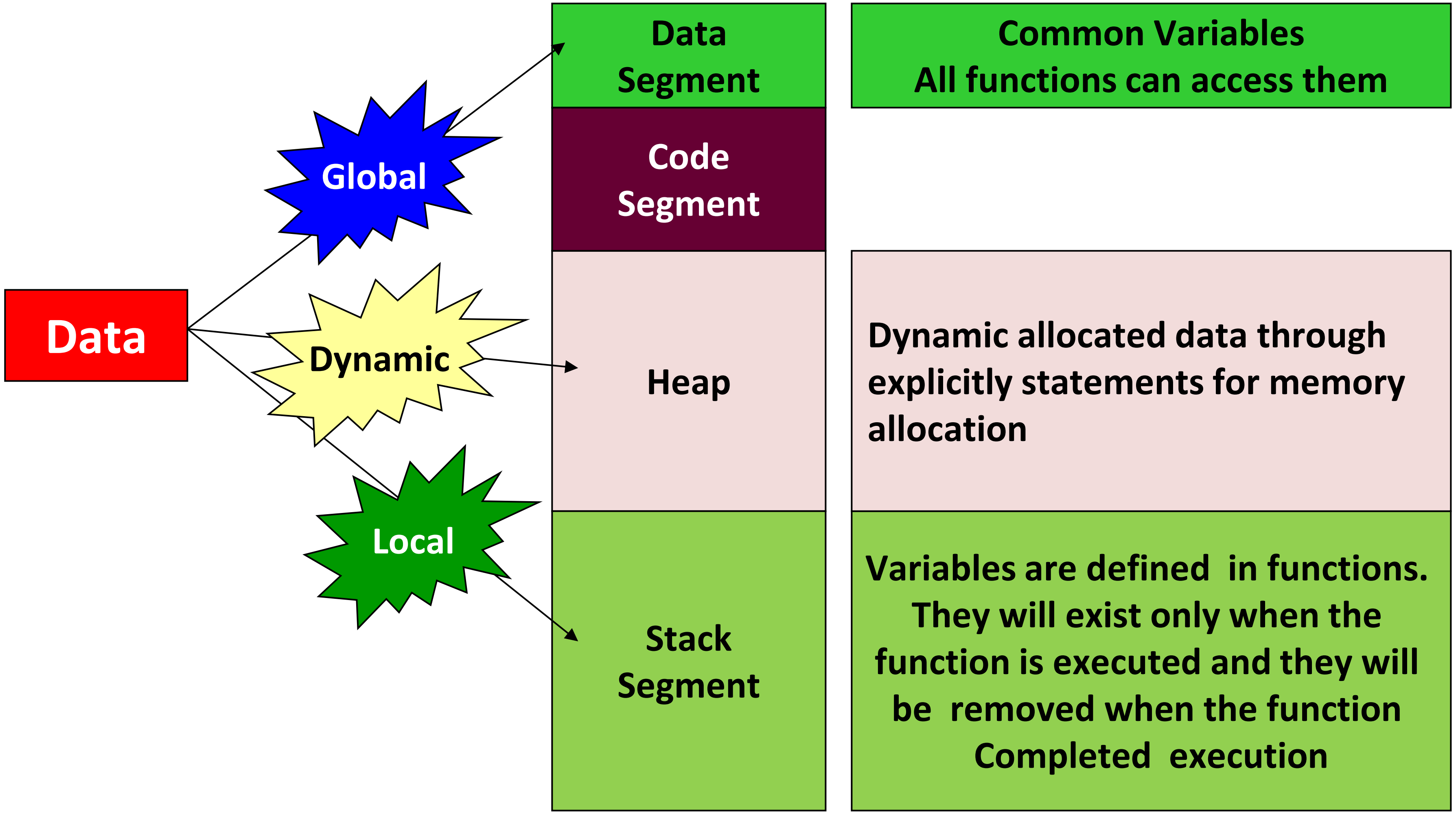
Address of a variable is a number. Can we assign this number to another variable then access data through the new variable?

Address of a function is a number. Can we assign this number to another variable then call this function through the new variable?

Yes. We can access a data through it's address and call a function through it's address also. **POINTER** is a way to satisfy these requirement. In this chapter, pointers of variables are concerned only.



2 - Where can we put program’s data?



3 - What is a Pointer?

- A pointer is a variable, which contains the address of a memory location of another variable.
- If one variable contains the address of another variable, the first variable is said to point to the second variable.
- A pointer provides an indirect method of accessing the value of a data item.
- Pointers can point to variables of other fundamental data types like int, char, or double or data aggregates like arrays or structures.

4 - Pointer variables

- A pointer declaration consists of a base type and a variable name preceded by an *
- **Syntax**:

```
dataType *pointerName;
```

- ***Note***: The created pointer will contain the address of the variable it points to, with the data type is dataType.
- **Example**:

```
int *pI;
double *pD;
char *pC;
```

5 - Why are Pointers used?

Some situations where pointers can be used are:

- To modify outside arguments of a function
- To return more than one value from a function
- To pass array and strings more conveniently from one function to another
- To manipulate arrays easily by moving pointers to them instead of moving the arrays itself
- To allocated memory and access it (direct memory allocation)

6 - Pointer Operators

How to	Operator	Example
Get address of a variable and assign it to a pointer	**&**	int n= 7; int *pn = &n; → assign address of **n** to pointer variable **pn**
Access indirectly value of a data through it's pointer	*	*pn =100;

```
int n= 7;
int *pn = &n;
```

pn = &n; → pn = 6684188

6684188 n = 7

6684176 pn

```
*pn = 100;
```

*pn = 100; → Value at [6684188] = 100

6684188 n = 7 → 100

6684176 pn = 10000

Pointer Operators: Example

```c
#include <stdio.h>

void printHeader(){
    printf("%-10s %-20s %-15s\n", "Variable", "Address", "Value");
    printf("----------------------------------------------\n");
}

int main(){
    int n = 7;
    // Declaration 2 pointers
    int *pn;
    int **ppn;
    pn = &n;     // Point the pointer pn to n
    ppn = &pn;   // Point the pointer ppn to pn

    printHeader();
    printf("%-10s %-15u %-5d\n", "n", &n, n);
    printf("%-10s %-15u %-5d\n", "pn", &pn, pn);
    printf("%-10s %-15u %-5d\n", "ppn", &ppn, ppn);

    printf("\n-----Using pointer p to update the n value-----\n");
    *pn = 10; // Access indirectly
    printHeader();
    printf("%-10s %-15u %-5d\n", "n", &n, n);

    printf("\n-----Using pointer ppn to update the n value-----\n");
    **ppn = 20; // Access indirectly
    printHeader();
    printf("%-10s %-15u %-5d\n", "n", &n, n);

    return 0;
}
```

n → int → **pn** stores address of **n** → **pn: int***
pn → int* → **ppn** stores address of **pn** → **ppn**: (int*)* → **ppn: int****

Result:

```
Variable   Address              Value
----------------------------------------------
n          6684188              7
pn         6684176              6684188
ppn        6684168              6684176

-----Using pointer p to update the n value-----
Variable   Address              Value
----------------------------------------------
n          6684188              10

----Using pointer ppn to update the n value----
Variable   Address              Value
----------------------------------------------
n          6684188              20
```

Pointer Operators: Walkthrough

address	memory-block
100	n=7 → 54
96	m=6 → -30
92	pn=100
88	pm= 96

```c
#include <stdio.h>
int main(){
    int n=7, m=6;

    int *pn = &n;
    int *pm = &m;

    *pn = 2*(*pm) + m*n;
    *pm += 3*m - (*pn);

    printf("m = %d, n = %d", m, n);
    return 0;
}
```

D:\MonHoc\PRF192\ThucHanh\pointer_walkthrough.exe

```
m = -30, n = 54
```

1

pn = 2(*pm) + m*n;
Value at 100 = 2*(value at 96) + m *n
Value at 100 = 2*6 + 6 *7
Value at 100 = 12 + 42 = 54

2

*pm += 3*m - (*pn);
Value at 96 += 3*6 – value at 100
Value at 96 += 3*6 – 54
Value at 96 += 18 – 54
Value at 96 += (-36)
Value at 96 = 6 + (-36) = -30

Exercises - Write code and Walkthrough

- **Exercise 1**:

```
int n = 7, m = 8;
int* p1= &n, *p2 = &m;
*p1 += 12 - m + (*p2);
*p2 = m + n - 2 * (*p1);
printf("%d", m+n);
```

→ **What is the output?**

- **Exercise 2**

```
int n = 7, m = 8;
int* p1= &n, *p2 = &m;
*p1 += 5 + 3 * (*p2) –n ;
*p2 = 5 * (*p1) – 4*m + 2*n;
```

→ **What is the output?**

Attention about Accessing Pointers

Accessing data through pointers will manipulate on basic-data size.

- Access int* → 4 bytes are affected.
- Access char* → only 1 byte is affected.
- Access double* → 8 bytes are affected.
- Assign pointers which belong to different types are not allowed. If needed, you must explicitly casting.

```
1 #include <stdio.h>
2 int main()
3 {   double x = 0.5;
4     double* pD = &x;
5     int * pI;
6     pI= pD;
7     getchar();
8     return 0;
9 }
```

Compiler | Resources | Compile Log | Debug | Find F

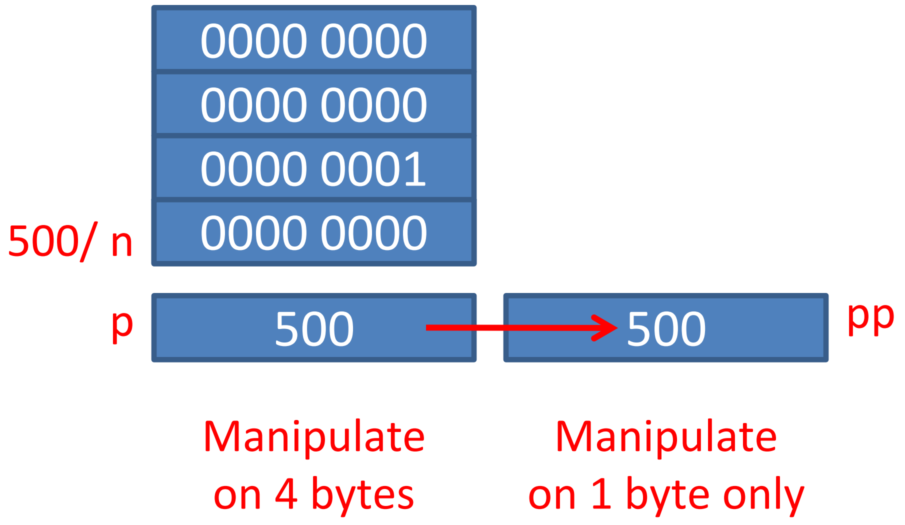
Line	File	Message
	K:\Gia...	In function `main':
6	K:\Gia...	[Warning] assignment from incompatible pointer type

Attention…Pointers: Explicit Casting

- Review: When a casting is performed, lowest byte is copied first then the higher bytes.

```c
#include <stdio.h>
#include <conio.h>
main()
{  int n=260, *p=&n;
   printf("n=%d\n",n);
   char *pp=(char*)p;
   *pp=0;
   printf("n=%d\n",n);
   getch();
}
```

```
n=260
n=256
```



Pointer Arithmetic Operators

++ptr_var or ptr_var++	points to next **integer** after var
—ptr_var or ptr_var—	points to **integer** previous to var
ptr_var + i	points to the ith integer after var
ptr_var - i	points to the ith integer before var
++*ptr_var or (*ptr_var)++	will increment **var** by 1
*ptr_var++	will fetch the value of the next integer after var

- Each time a pointer is incremented, it points to the memory location of the next element of its base type.
- Each time it is decremented it points to the location of the previous element.
- All other pointers will increase or decrease depending on the length of the data type they are pointing to.

Pointer Comparisons

- Two pointers can be compared in a relational expression provided both the pointers are pointing to variables of the same type.
- Consider that **ptr_a** and **ptr_b** are 2 pointer variables, which point to data elements **a** and **b**. In this case the following comparisons are possible:

ptr_a < ptr_b	Returns true provided **a** is stored before **b**
ptr_a > ptr_b	Returns true provided **a** is stored after **b**
ptr_a <= ptr_b	Returns true provided **a** is stored before **b** or ptr_a and ptr_b point to the same location
ptr_a >= ptr_b	Returns true provided **a** is stored after **b** or ptr_a and ptr_b point to the same location.
ptr_a == ptr_b	Returns true provided both pointers ptr_a and ptr_b points to the same data element.
ptr_a != ptr_b	Returns true provided both pointers ptr_a and ptr_b point to different data elements but of the same type.
ptr_a == NULL	Returns true if ptr_a is assigned NULL value (zero)

Pointer Arithmetic Operators: Example

```c
#include <stdio.h>

int main(){
    char    c = 'a';
    int     n = 1;
    double  d = 0.5;
    char    *pc = &c;
    int     *pn = &n;
    double  *pd = &d;

    printf("\n\npc, pn, pd with +0: %d, %d, %d", pc, pn, pd);
    printf("\n\npc, pn, pd with +1: %d, %d, %d", pc+1, pn+1, pd+1);
    printf("\n\npc, pn, pd with +2: %d, %d, %d", pc+2, pn+2, pd+2);
    return 0;
}
```

```
pc, pn, pd with +0: 6487559, 6487552, 6487544

pc, pn, pd with +1: 6487560, 6487556, 6487552

pc, pn, pd with +2: 6487561, 6487560, 6487560
```

Pointer Arithmetic Operators: Example

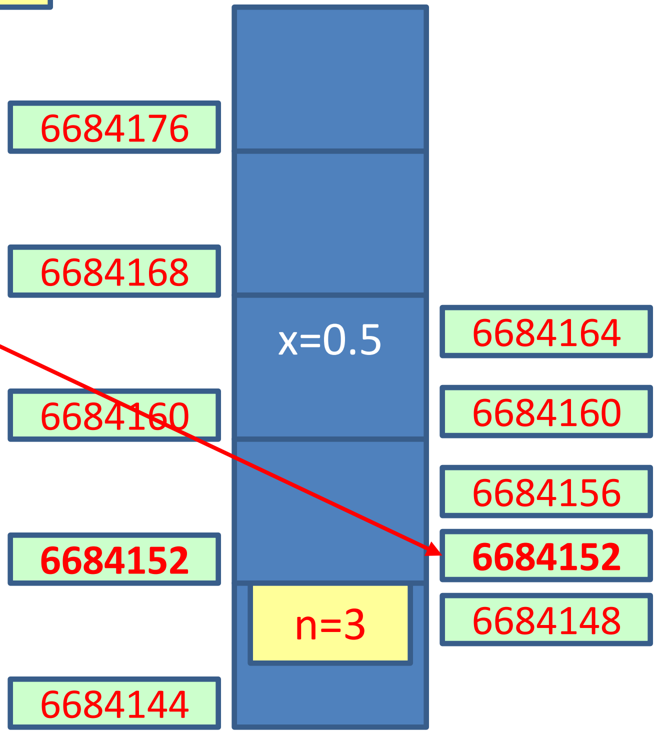
```c
#include <stdio.h>
int main(){
    double x = 0.5;
    double *pD = &x;

    int i;
    for(i=-2; i<=2; i++){
        printf("%u, ", pD+i);
    }
    printf("\n");

    int n = 3;
    int *pI = &n;
    for(i=-2; i<=2; i++){
        printf("%u, ", pI+i);
    }

    pI = pI - 1;
    *pI = 10;
    pD = pD - 1;
    *pD = 20.5;
    printf("\n\nAddress of pI: %u - Value: %d", pI, *pI);
    printf("\nAddress of pD: %u - Value: %lf\n", pD, *pD);

    pI++;
    printf("\n%u\n", pI);
    pI--;
    printf("%u\n", pI);

    printf("\n");
    return 0;
}
```

Pointer + i → Pointer + (i*sizeof(baseType))

If access data using **pI** (bytes) can cause harm to the variable

D:\MonHoc\PRF192\ThucHanh\pointer_arithmetic_operators.exe

```
6684144, 6684152, 6684160, 6684168, 6684176,
6684148, 6684152, 6684156, 6684160, 6684164,

Address of pI: 6684152 - Value: 0
Address of pD: 6684152 - Value: 20.500000

6684156
6684152
```



Exercise 3: Accessing the neighbor

- Rewrite, run the program and explain the result.

```
/* file pointer_demo.c */
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int n2= 10;
        int n1= 6;
        int n0= 5;
        printf("n2=%d, n1=%d, n0=%d\n", n2, n1, n0);
        int* p = &n1;
        *p=9;
        p++;
        *p=15;
        p--;
        p--;
        *p=-3;
        printf("n2=%d, n1=%d, n0=%d\n", n2, n1, n0);
        system("pause");
        return 0;
}
```

Exercises

- **Exercise 4**:

long*p;
Suppose that a long number occupies the memory block of 4 bytes
And p stores the value of 1000.
What are the result of the following expression? **p+8   p-3   p++**

- **Exercise 5**:

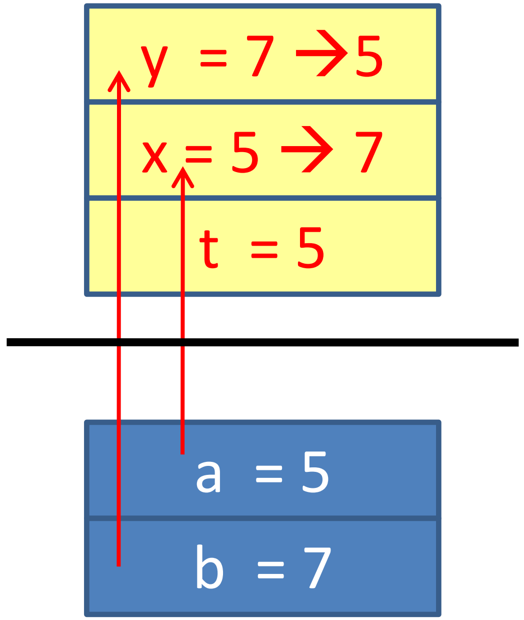
char*p;
Suppose that a character occupies the memory block of 1 byte
and p stores the value of 207000.
What are the result of the following expression? **p+8   p-3   p++**

7 - Pointers as Parameters of a Function

- Problem: C passes arguments to parameters by values only → C functions can not modify outside data.

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

/* swap 2 integers */
void swap1(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}

int main(){
    int a=5, b=7;

    printf("a=%d, b=%d\n", a, b);
    // Call swap1(...) functions
    swap1(a, b);
    printf("a=%d, b=%d\n", a, b);

    system("pause");
    return 0;
}
```

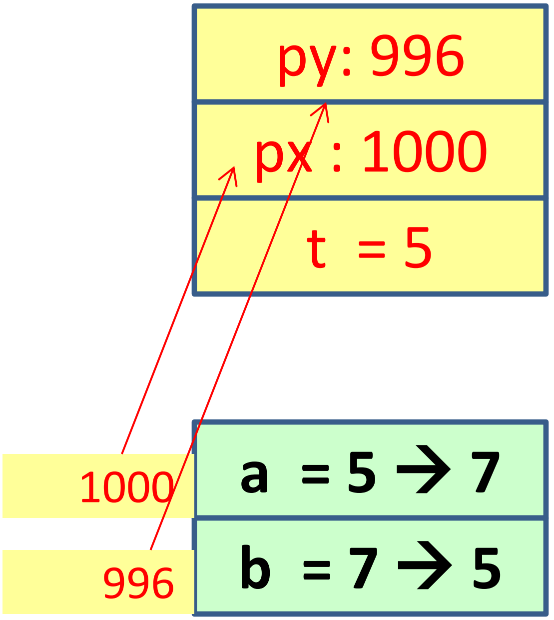
```
a=5, b=7
a=5, b=7
Press any key to continue . . .
```



7 - Pointers as Parameters of a Function (cont.)

- Solution: Use pointer arguments, we can modify outside values

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

/* swap 2 integers */
void swap2(int *px, int *py){
    int temp = *px; // t = value at px
    *px = *py;      // value at px = value at py
    *py = temp;     // value at py = temp
}

int main(){
    int a=5, b=7;

    printf("a=%d, b=%d\n", a, b);
    // Call swap2(...) functions
    swap2(&a, &b);
    printf("a=%d, b=%d\n", a, b);

    system("pause");
    return 0;
}
```




```
a=5, b=7
a=7, b=5
Press any key to continue . . .
```

8 - Dynamic Allocated Data

- In C allows you to allocate memory during runtime using functions provided in the standard library.
- It is particularly useful when the size of the data is not known at compile time.
- Dynamic memory is managed via pointers, and you can allocate and deallocate memory as needed.
- Standard library functions: **stdlib.h**

size_t: Another name of the **int** type. It is used in case of memory allocation managing.

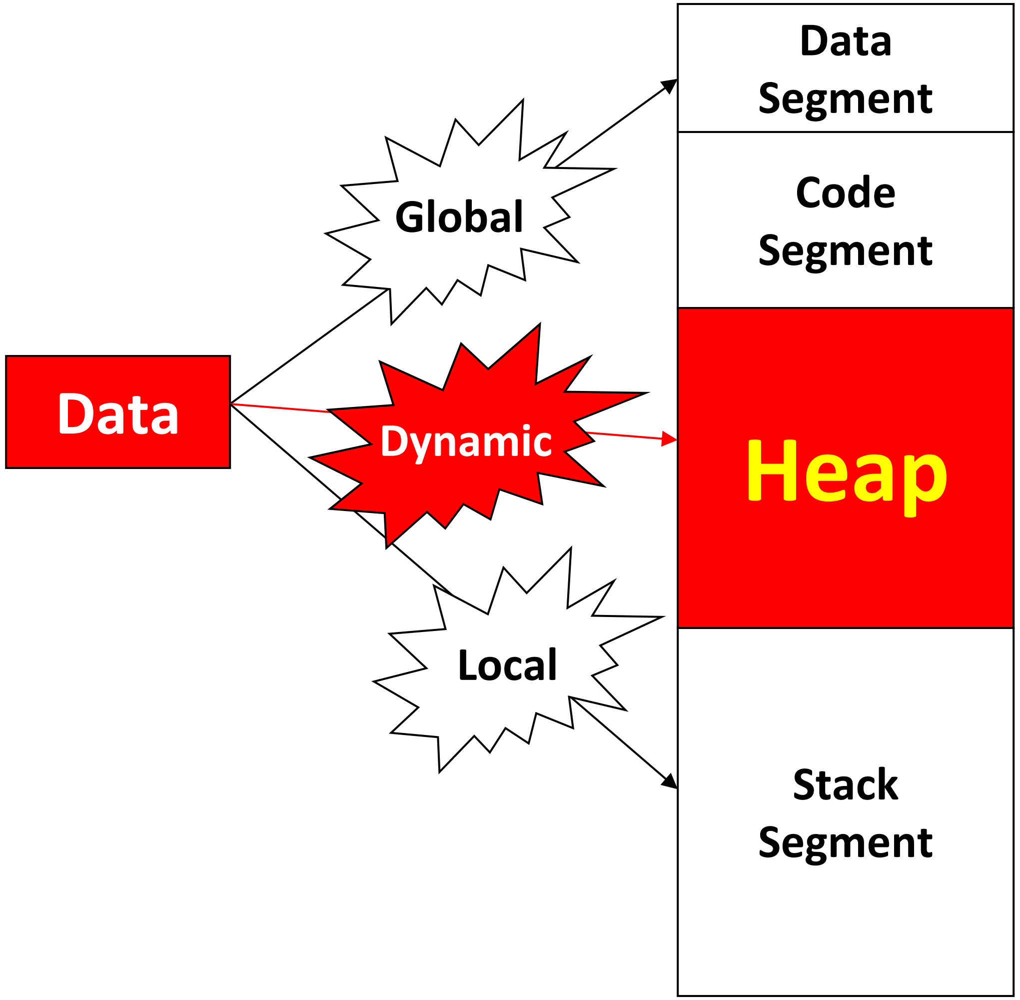
```
void*  calloc  (size_t numberOfItem, size_t  bytesPerItem)
void*  malloc (size_t  numBytes) ;
void* realloc (void* curPointer, size_t newNumBytes);
void  free(void* willBeDeletedPointer);
```

void is the general datatype which means that the data type is not determined yet. So, user must give an explicit casting when it is used.

Dynamic Allocated Data (cont.)

Example:

```
int* p = (int*)  malloc (sizeof (int)) ;
*p=2;
….
 free(p);
```



malloc (Memory Allocation)

- Allocates a specified number of bytes and returns a pointer to the allocated memory. The memory is uninitialized, meaning it may contain garbage values.
- **Syntax:** **void *malloc(size_t size);**
- Example:

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    // Allocates memory for an array of 5 integers
    int *arr = (int *)malloc((5*sizeof(int)));
    if(arr == NULL){
        printf("Memory allocation failed\n");
    }else{
        // Print memory infomation of 'arr'
        printf("Address of arr: %u\n", (void *)arr);
        printf("Size of memory of arr: %lu bytes\n", 5 * sizeof(int));
    }
    return 0;
}
```

```
Address of arr: 7345152
Size of memory of arr: 20 bytes
```

calloc (Contiguous Allocation)

- Allocates memory for an array of elements and initializes all bytes to zero.
- **Syntax**:

```
void *calloc(size_t   num, size_t   size);
```

- Example:

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    // Allocates and initializes memory for 5 integers
    int *arr = (int *)calloc(5, sizeof(int));
    if(arr == NULL){
        printf("Memory allocation failed\n");
    }else{
        // Print memory infomation of 'arr'
        printf("Address of arr: %u\n", (void *)arr);
        printf("Size of memory of arr: %lu bytes\n", 5 * sizeof(int));
    }
    return 0;
}
```

```
Address of arr: 7803904
Size of memory of arr: 20 bytes
```

realloc (Reallocation)

- Resizes an already allocated memory block. It can shrink or expand the memory block. If it expands the block, the new memory might be uninitialized.
- **Syntax**:

```
void  *realloc(void  *ptr, size_t   size);
```

- Example:

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

int main(){
    int *arr = (int *)malloc(5 * sizeof(int)); // Allocates memory for an array of 5 integers
    arr = (int *)realloc(arr, 10 * sizeof(int)); // Resize memory to hold 10 integers
    if (arr == NULL) {
        printf("Memory reallocation failed\n");
    }

    system("pause");
    return 0;
}
```

free (Deallocation)

- Frees memory previously allocated with: **malloc**, **calloc**, **realloc**
- **Syntax**:

```
void  free(void  *ptr);
```

- Example:

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

int main(){
    int *arr = (int *)malloc(5 * sizeof(int)); // Allocates memory for an array of 5 integers

    // ... other operations on array

    free(arr); // Releases allocated memory
    arr = NULL; // Avoids dangling pointer

    system("pause");
    return 0;
}
```

Dynamic Allocated Data: Demo 1

```
#include <stdio.h>
#include <stdlib.h>
const int MAXN =100;
int main()
{
  int n;   int *p1;   int *p2;   int *p3;
  printf("Address of MAXN: %u\n", &MAXN);
  printf("Main function ia allocated at: %u\n", &main);
  printf("Address of n : %u\n", &n);
  printf("Address of p1: %u\n", &p1);
  printf("Address of p2: %u\n", &p2);
  p1 = (int*)malloc(sizeof(int));
  p2 = (int*)malloc(sizeof(int));
  p3 = (int*)malloc(sizeof(int));
  printf("Dynamic allocation (p1) at: %u\n", p1);
  printf("Dynamic allocation (p2) at: %u\n", p2);
  printf("Dynamic allocation (p3) at: %u\n", p3);
  free(p1);
  free(p2);
  system("pause");
  return 0;
}
```

Requirement

(1) Copy, past, compile and run the program.
(2) Draw the memory map.
(3) Show that where is data segment, code segment, stack segment and heap of the program.
(4) Give comment about the direction of dynamic memory allocation.

Dynamic Allocated Data: Demo 2

- Use dynamic memory allocation. Develop a program that will accept two real numbers then sum of them, their difference, their product, and their quotient are printed out.
- Do yourself:

```
/* main() */
double *p1, *p2;
p1 = (double*) malloc ( sizeof(double));
p2 = (double*) malloc ( sizeof(double));
printf("p1, address: %u, value: %u\n", &p1, p1);
printf("p2, address: %u, value: %u\n", &p2, p2);
printf("Input 2 numbers:");
scanf( "%lf%lf", p1, p2);
printf("Sum: %lf\n", *p1 + *p2);
printf("Difference: %lf\n", *p1 - *p2);
printf("Product: %lf\n", *p1 * (*p2));
printf("Quotient: %lf\n", *p1 / *p2);
```

Requirement →

(1) Run this program
(2) Draw the memory map (stack, heap).

Dynamic Allocated Data: Demo 3

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

int main() {
    int n, i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Input values
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
```

```c
    // Display values
    printf("You entered:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(arr);

    return 0;
}
```

```
Enter the number of elements: 5
Enter 5 elements:
10
5
7
20
6
You entered:
10 5 7 20 6
```

Dynamic Allocated Data: Note

- Always check if the pointer returned by malloc, calloc, or realloc is NULL to avoid dereferencing a null pointer.
- Always use free to deallocate dynamically allocated memory when it's no longer needed.
- Avoid memory leaks by freeing all allocated memory.
- Avoid using free on pointers that were not dynamically allocated.

Exercise 6:

- Write a C program using dynamic allocating memory to allow user entering two characters then the program will print out characters between these in ascending order.
- Example:

  Input: DA

  Output:

A	65	81	41
B	66	82	42
C	67	83	43
D	68	84	44

- After the program executes, draw the memory map of the program.

Summary

- Review the memory structure of a program
- Where can we put program's data?
- Why are pointers?
- Pointer Declarations
- Where are pointers used?
- Pointer operators
  - Assign values to pointers
  - Access data through pointer
  - Explain pointer arithmetic
  - Explain pointer comparisons
- Pointers as parameters of a function
- Dynamic Allocated Data