Chapter 08
Function

Dr. Le Thanh Sach

Content

- What is function?
- Reason to use function
- The “main” function and library function
- Using user defined functions
  - Definition
  - Function call
  - Principles of execution when calling a function
- Function prototype, function signature, function overloading
- Pass by parameters
- Function and array, pointer
- Inline function
- Function pointer
- Recursive function
- Create a function library
  - Static links and dynamic links

What is function?

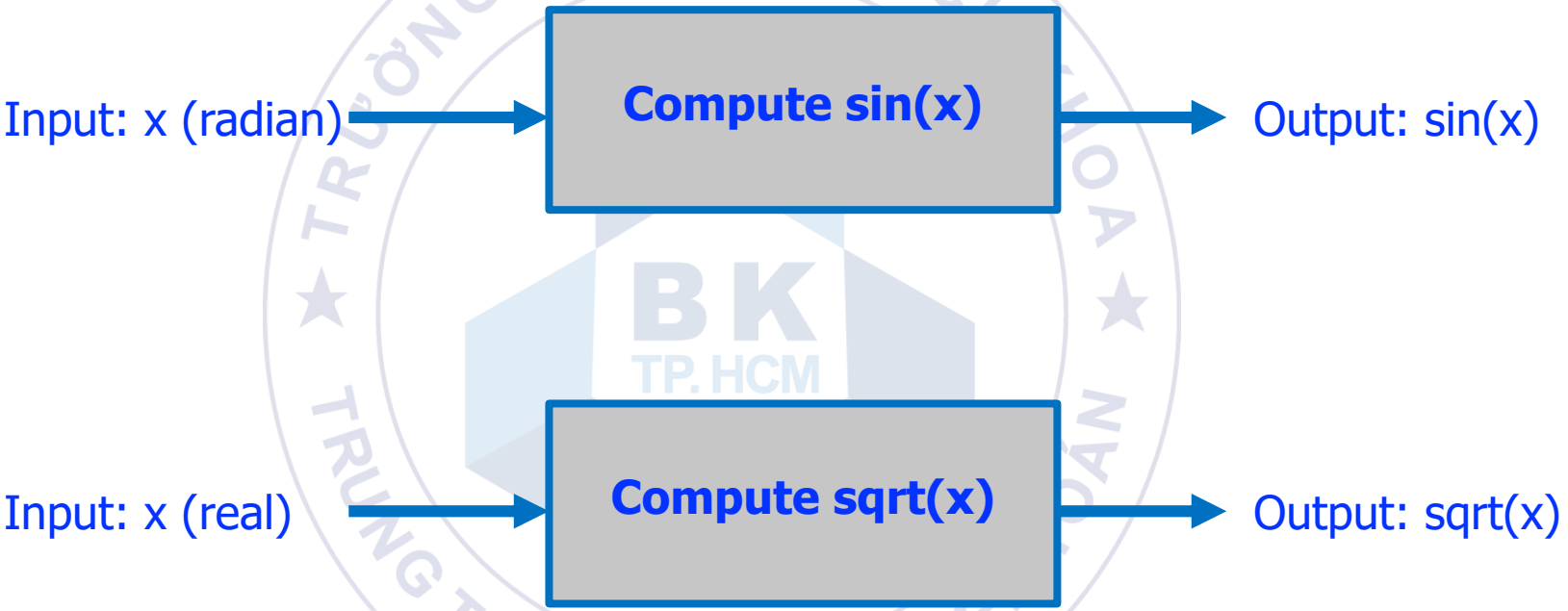
- Function is
  - A processing unit
  - A group of statements that are **related**, **executing together** to perform **a task**
  - Example: In library <math.h>
    - Function sin(x)
      - A group of statements that compute the sine of an angle x, The angle x has the unit of radians; function sin(x) returns a real number
    - Function sqrt(x)
      - A group of statements that compute the square root of x, x is real-valued (float or double); function sqrt(x) returns a real number

What is function?

- Function is
  - A processing unit
    - Get input value
    - Calculate
    - Return value
  - Illustration

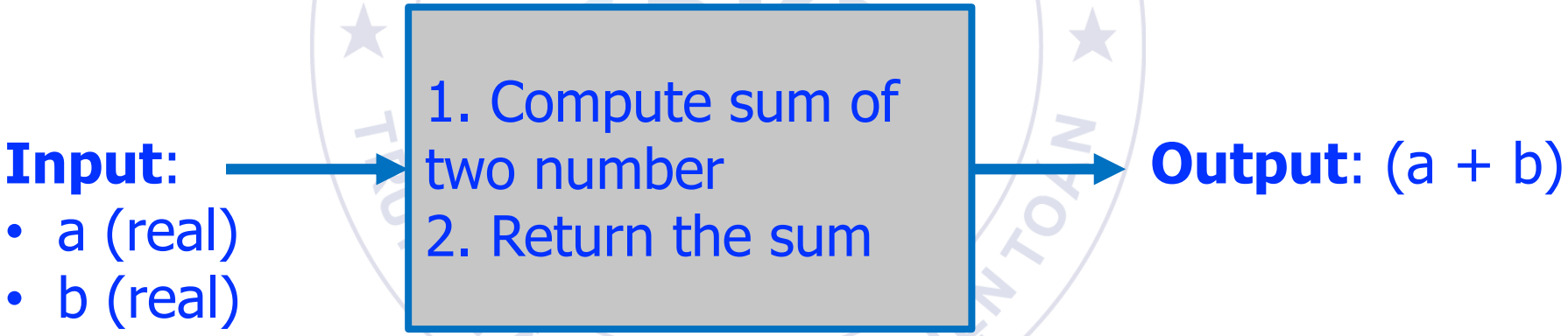
What is function?

- Illustration



What is function?

- Illustration for a function that adds two numbers
  - Input: Two real number a and b
  - Computation: Add two numbers
  - Output: Sum of two number



Reason to use function

- Avoid duplicating source code
  - ➔ Save development time
  - ➔ Change the source code in the function quickly and easily, just in one place
- Reuse a unit of calculation without having to rewrite it
  - Save development time
  - It is possible to share the unit of calculation not only for one project but for multiple projects
  - Example: Let's consider the case that every project is rewriting the mathematical function: sin(x), sqrt(x), v.v. ➔ Costly and wasteful
    - ➔ Use library <math.h>

Reason to use function

- Help developing algorithms, organizing programs easily
  - Algorithm:
    - A fundamental technique for problem-solving: decompose a large problem into smaller sub-problems
      - ➔ Each sub-problem can be a unit of calculation (as a function)
    - Example: Enter a sequence of numbers, calculate and print the mean and standard deviation of them. This problem can be decomposed into sub-problems
      - (1) Input a sequence of numbers
      - (2) Compute mean and standard deviation
      - (3) Print out the input numbers, mean and standard deviation

Reason to use function

- Help developing algorithms, organizing programs easily
  - Algorithm:
    - Example: Enter a sequence of numbers, calculate and print the mean and standard deviation of them. This problem can be decomposed into sub-problems
      - (1) Input a sequence of numbers
      - (2) Compute mean and standard deviation
      - (3) Print out the input numbers, mean and standard deviation
      - ➔ Each of the sub-problems above can be written as a function

Reason to use function

- Help developing algorithms, organizing programs easily
  - Organize a program:
    - Comparing a program (written in C ++) to a book (English)
      - Is there any book in reality where the author writes the whole book into successive sentences; do not separate it into chapters, sections, subsections, paragraphs?
      - Functions have similar meaning as chapters or sections

The "main" function and library function

Return value: Type of **int**

Function name: "main". A program must have and have only one main function

```
int main(){
    // statements of the main function
    return 0;
}
```

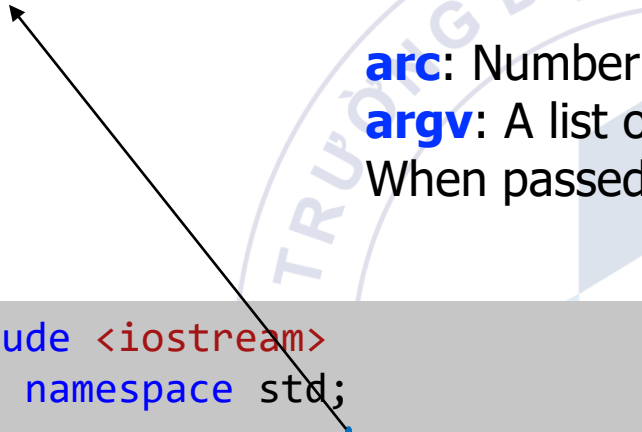
Returns the value to the caller of main

Return value of function "main":

- Must be **int**
- May be one of two constants
  - EXIT_SUCCESS (or 0): If the program finishes successfully
  - EXIT_FAILURE (or 1): If the program ends with some errors

The "main" function and library function

If you want to pass arguments in the command line



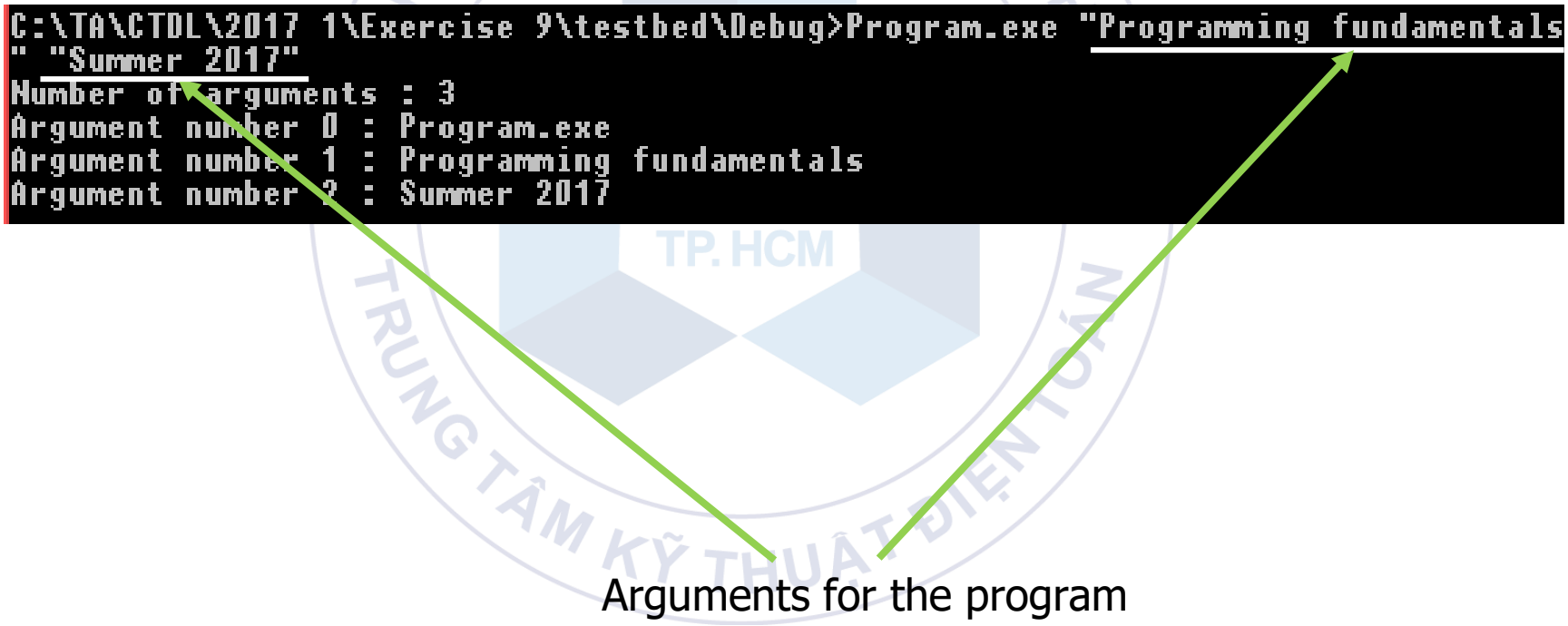
arc: Number of arguments , including program name
argv: A list of strings, each string is an argument.
When passed in, all the data is interpreted as strings

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[]){
        cout << "Number of arguments: " << argc << endl;
        for(int i=0; i < argc; i++)
                cout << "Argument number " << i << ":" << argv[i] << endl;

        return EXIT_SUCCESS;
}
```

The "main" function and library function

- After compiling the program successfully, the file "Program.exe" is created
- Run "Program.exe" in command line as follow:

```
C:\TA\CTDL\2017 1\Exercise 9\testbed\Debug>Program.exe "Programming fundamentals
" "Summer 2017"
Number of arguments : 3
Argument number 0 : Program.exe
Argument number 1 : Programming fundamentals
Argument number 2 : Summer 2017
```

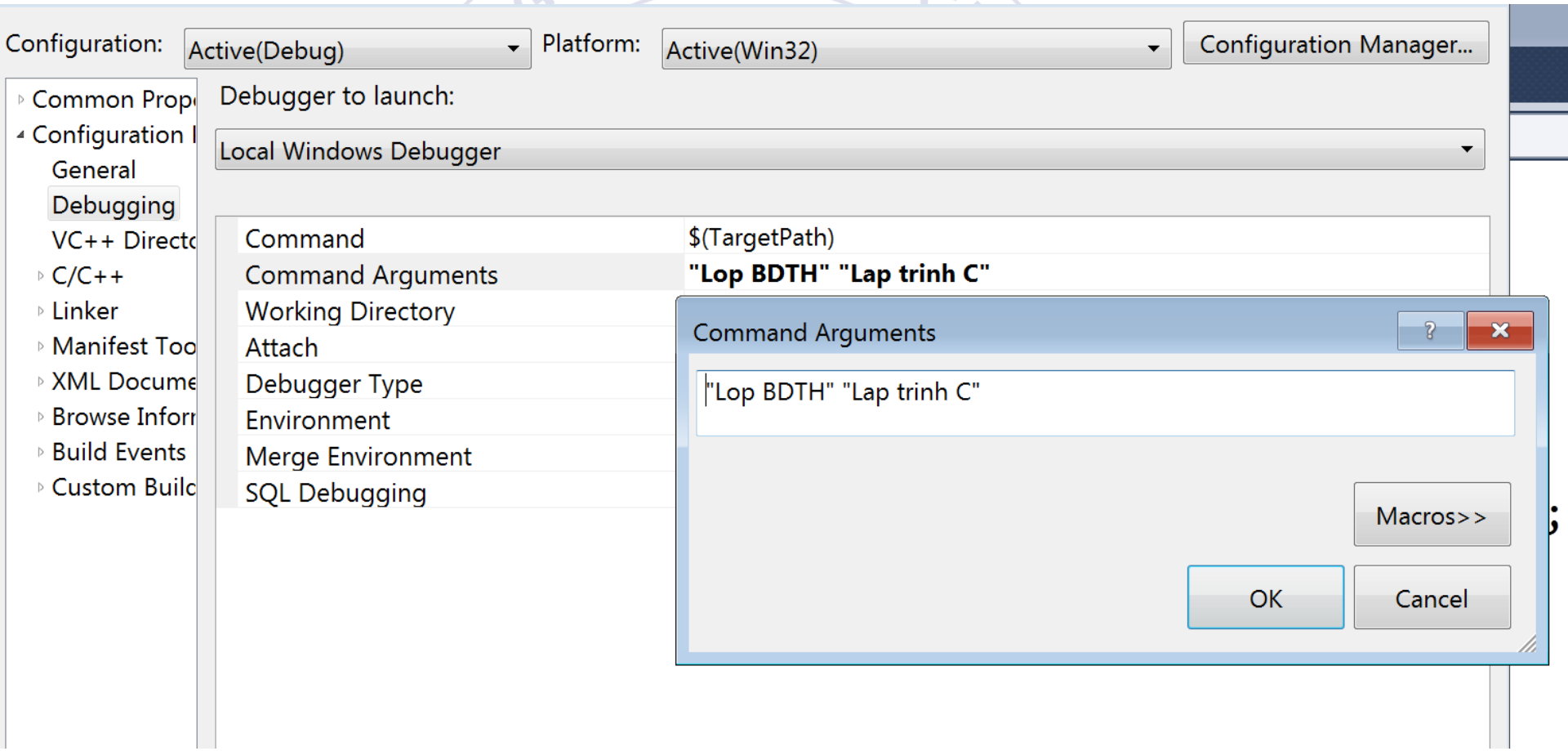


The “main” function and library function

- How to pass command line arguments in Visual Studio
  - (1) Right-click on the <project> in the "Solution Explorer"
  - (2) Choose “Debug” > “Command Arguments”
  - (3) Choose “Edit …” in the function list of “Command Arguments”
  - (4) **Type in the argument list: arguments are separated by spaces or comma (“,”)**

The "main" function and library function

- How to pass command line arguments in Visual Studio



The "main" function and library function

- Example: Functions in the library <math.h>
  - (1) Use directive #include <math.h> to tell the compiler to use the library <math.h>
  - (2) Call the necessary functions. When calling a function, we only need to know
    - Function name + use of a function
    - The values to be provided to the function
    - The return value of the function

The "main" function and library function

- Example: Functions in the library <math.h>
  - Example code:


```
#include <iostream>
#include <iomanip>
#include <math.h>
using namespace std;

int main(int argc, char* argv[]){
        cout << setw(10) << "sqrt(25.0)" << "=" << sqrt(25.0) << endl;
        cout << setw(10) << "sin(90.0)" << "=" <<
                sin(90.0f*(3.14159/180.0)) << endl;
        cout << setw(10) << "cos(0.0)" << "=" <<
                cos(90.0f*(3.14159/180.0)) << endl;

        cout << endl << endl;
        system("pause");
        return EXIT_SUCCESS;
}
```

Using user-defined functions

- Consist of two steps
  - (1) Create a function
    - Describe a function
    - Implement a function
  - (2) Call the function

Using user-defined functions

Function definition

```
#include <iostream>
using namespace std;

int add(int a, int b)
{
        int c;
        c = a + b;
        return c;
}
int main(){
        cout << "10 + 15 = " << add(10, 15) <<endl;

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

Function description, includes:

(1) Return type: in this example it's int

(2) Function name: in this example it's "add"

(3) Parameters: are the inputs of the function

This example has

- The first parameter: Named "a", type of int
- Second parameter: Named "b", type of int

- Parameter list: Begin with "(", end with ")"
- Parameters are separated by commas ","

Function names and parameter names follow C++ naming rules

Using user-defined functions

Function definition

```
#include <iostream>
using namespace std;

int add(int a, int b)
{
        int c;
        c = a + b;
        return c;
}
int main(){
        cout << "10 + 15 = " << add(10, 15) <<endl;

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

The body of the function:
Including statements that are to be executed together, sequentially. In this example: There are 3 statements in the body of function **add**.

Use statement **return** to terminate function execution and return the control to the function called this function → Execute the statement after function call statement.

The commands in the body of the function must be nested together with a pair of curly braces "{" and "}"

Using user-defined functions

Function call

```
#include <iostream>
using namespace std;

int add(int a, int b)
{
        int c;
        c = a + b;
        return c;
}
int main(){
        cout << "10 + 15 = " << add(10, 15) <<endl;

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

Function call :

Use the function name and pass values as parameters to the function:

- The passing order decides which value will be passed to which parameter.
- This example: 10 is passed to a; 15 is passed to b
- Must pass enough (No missing, no redundant) all parameters
- Invalid calls: add(), add(10), add(10, 20, 30)

The function defination must appear before the function call to compile without the "undefined identifer" error.

Using user-defined functions

Principles of execution when calling a function

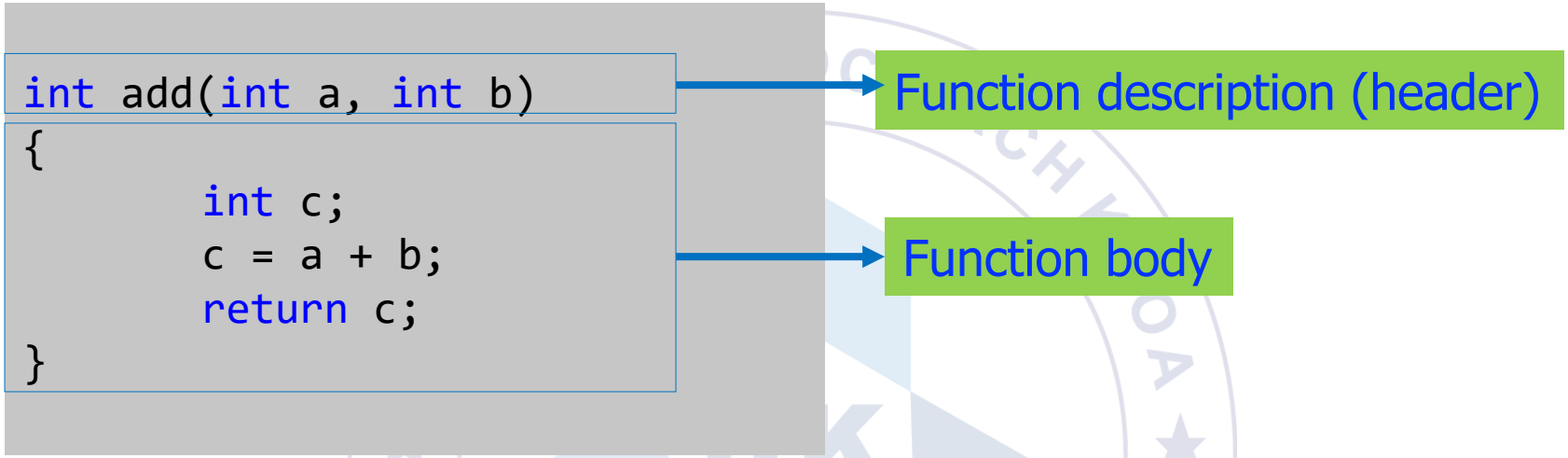
- When the function call is executed, the execution will do the following tasks
  - Trace: The next statement of the function call
  - Copy parameters for the function to be called
  - *Do other system jobs (C++ programmers should not care for now)*
  - Transfer execution control to the called function so that it executes its first statement
  - The called function execute the other statements
  - When the called function executes the statement return:
    - Release all of its local variables
    - Returns control to the statement following the function call statement in the caller
  - The caller function releases the passed parameters and **executes the next statement** after the function call statement

Using user-defined functions

Principles of execution when calling a function

- Display the actual sequence of execution using the "debug“ tool
  - Direct illustration
  - Note: Show Assembly statememts

Organizing source code



The description **should be separated** from the whole definition of a function.

Reason:

- No need to care about the order of functions in source code.
- Reusing functions in a project or multiple projects
- Develop library of functions → No need to deliver the actual implementation source code to 3rd-parties (Library buyers)

Organizing source code

```
#include <iostream>
using namespace std;

int add(int a, int b);

int main(){
        cout << "10 + 15 =" << add(10, 15);

        cout << endl;
        system("pause");
        return EXIT_SUCCESS;
}

int add(int a, int b)
{
        int c;
        c = a + b;
        return c;
}
```

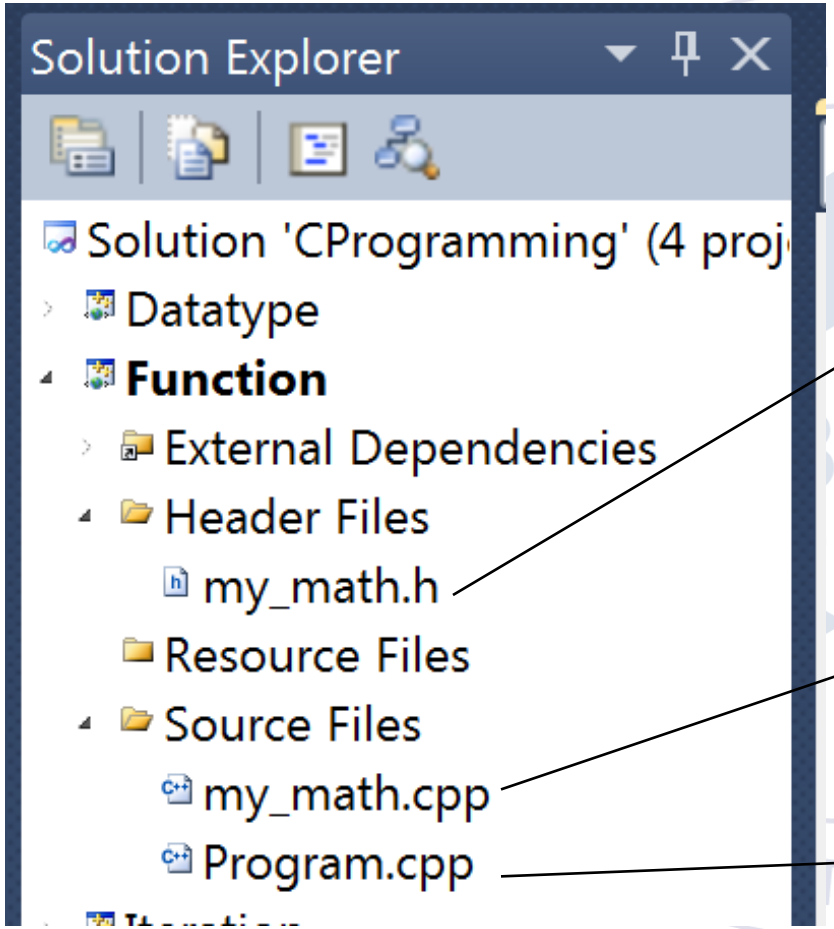
Separate the description of the "add" function and put it before "main" function (before usage).

➔ It is not necessary to place the entire definition of the "add" function before the "main" function

Organizing source code

- Put the descriptions into a separate file
  - Called description files (header): *.h
  - Can be reused in many other files in the project
    - Use the directive `#if !defined(.) … endif` to avoid the "repeat definition" error (`redefinition`)
- Put the implementation section into a separate file
  - Called implementation file: *.c; *.cpp
  - Can be reused in many other files in the project
  - Declaration should use the functions in * .h mentioned above

Organizing source code

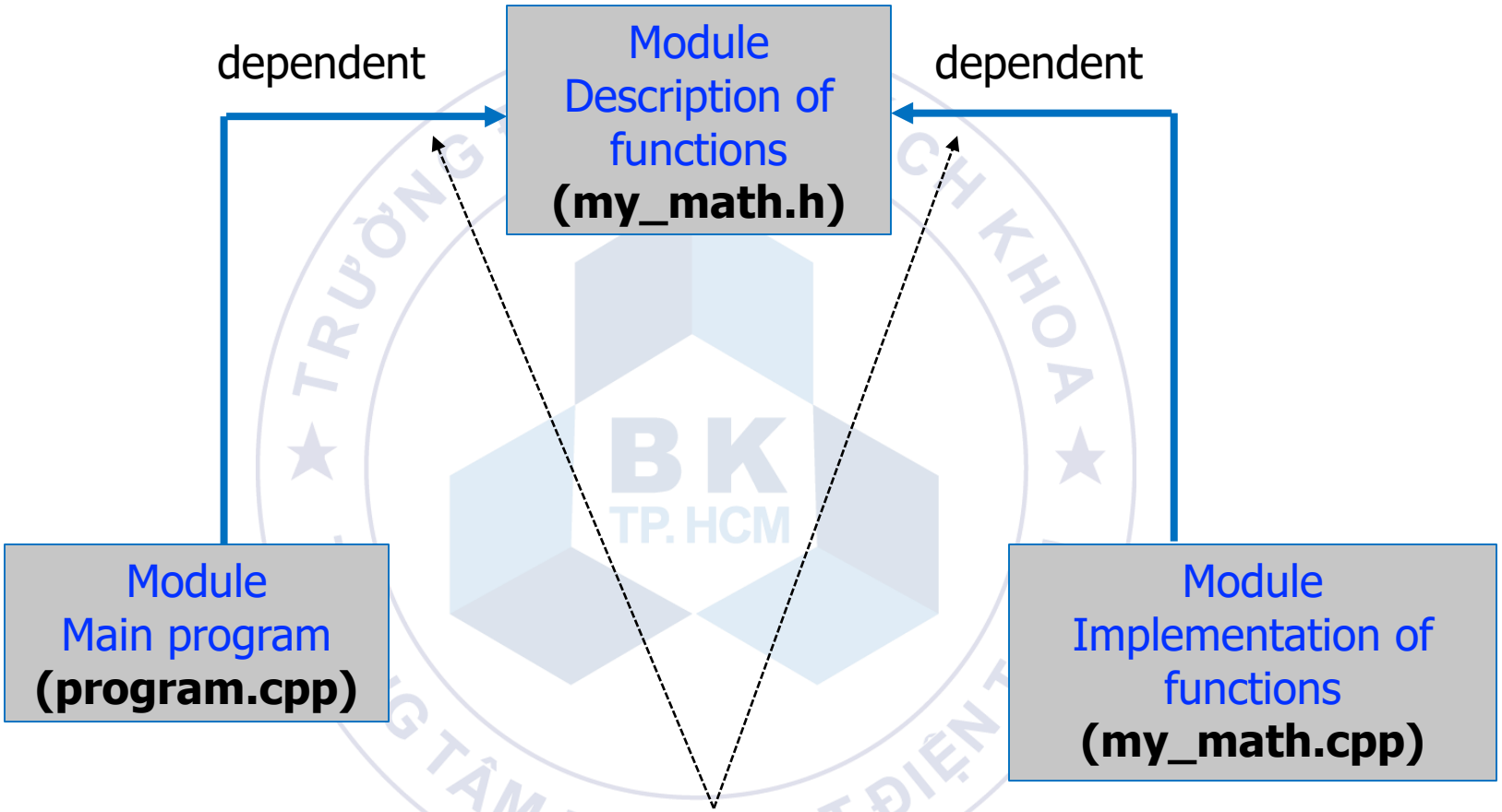


The file contains the description for the functions, data type, etc. Generally, descriptive sections

The file contains the implementation of "add" function, its declaration uses the description in * .h

The file contains the main function, which call the "add" function.

Organizing source code



Dependence is expressed by the directive
`#include "my_math.h"` in the source code

Organizing source code

File: ”my_math.h”

```
#if !defined(MY_MATH_HEADER)
#define MY_MATH_HEADER

int add(int a, int b);

#endif
```

MY_MATH_HEADER
Is a token (name)

Description of the add function

The meaning of the indicator #if:
If in the compilation process, up to the current point, the compiler has not yet seen token (MY_MATH_HEADER) then it will define a new name (MY_MATH_HEADER) And compile the source code in the corresponding block #if
Else, Does not define a new name and does not compile the corresponding block source code if

Organizing source code

File: ”my_math.h”

```
#if !defined(MY_MATH_HEADER)
#define MY_MATH_HEADER

int add(int a, int b);

#endif
```

MY_MATH_HEADER
Is a token (name)

Description of the add function

Thanks to the #if ... directive, description of the same function are not repeated many times when used in many different files, including the * .h file.

Organizing source code

File: ”my_math.cpp”

```
#include "my_math.h"
int add(int a, int b)
{
      int c;
      c = a + b;
      return c;
}
```

Declare the use of the description in * .h file ("my_math.h")

The implementation of a function (add function)

Organizing source code

File: ”program.cpp”

```
#include <iostream>
#include "my_math.h"
using namespace std;

int main(){
        cout << "10 + 15 =" << add(10, 15)
        << endl;
        system("pause");
        return EXIT_SUCCESS;
}
```

Declare the use the descriptions in * .h file ("my_math.h")

Function call (add function)

Organizing source code

- Problem: construct functions to calculate a complex number
- Analysis:
  - Need to provide data type for complex numbers: z = x + y*i
  - Provides functions with this new type
    - The function that compute the magnitude of the complex number

$$r = |z| = \sqrt{x^2 + y^2}$$

Organizing source code

- Problem: construct functions to calculate a complex number
- Analysis:
  - Need to provide data type for complex numbers: z = x + y*i
  - Provides functions with this new type
    - The function that compute the magnitude of the complex number
    - The function that computes the angle of the complex number

$$\varphi = \arg(z) = \begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0 \\ \arctan(\frac{y}{x}) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan(\frac{y}{x}) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{indeterminate} & \text{if } x = 0 \text{ and } y = 0. \end{cases}$$

Organizing source code

- The function that computes the angle of the complex number

$$\varphi = \arg(z) = \begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0 \\ \arctan(\frac{y}{x}) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan(\frac{y}{x}) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{indeterminate} & \text{if } x = 0 \text{ and } y = 0. \end{cases}$$

  - Need to define constants
    - Pi
    - Constant represents “indeterminate” value
      - Can be represented by 2 * PI or any value outside the range [-PI, PI].
    - It is necessary to define macros to support "==“ comparison with real numbers to avoid numerical representation errors

Organizing source code

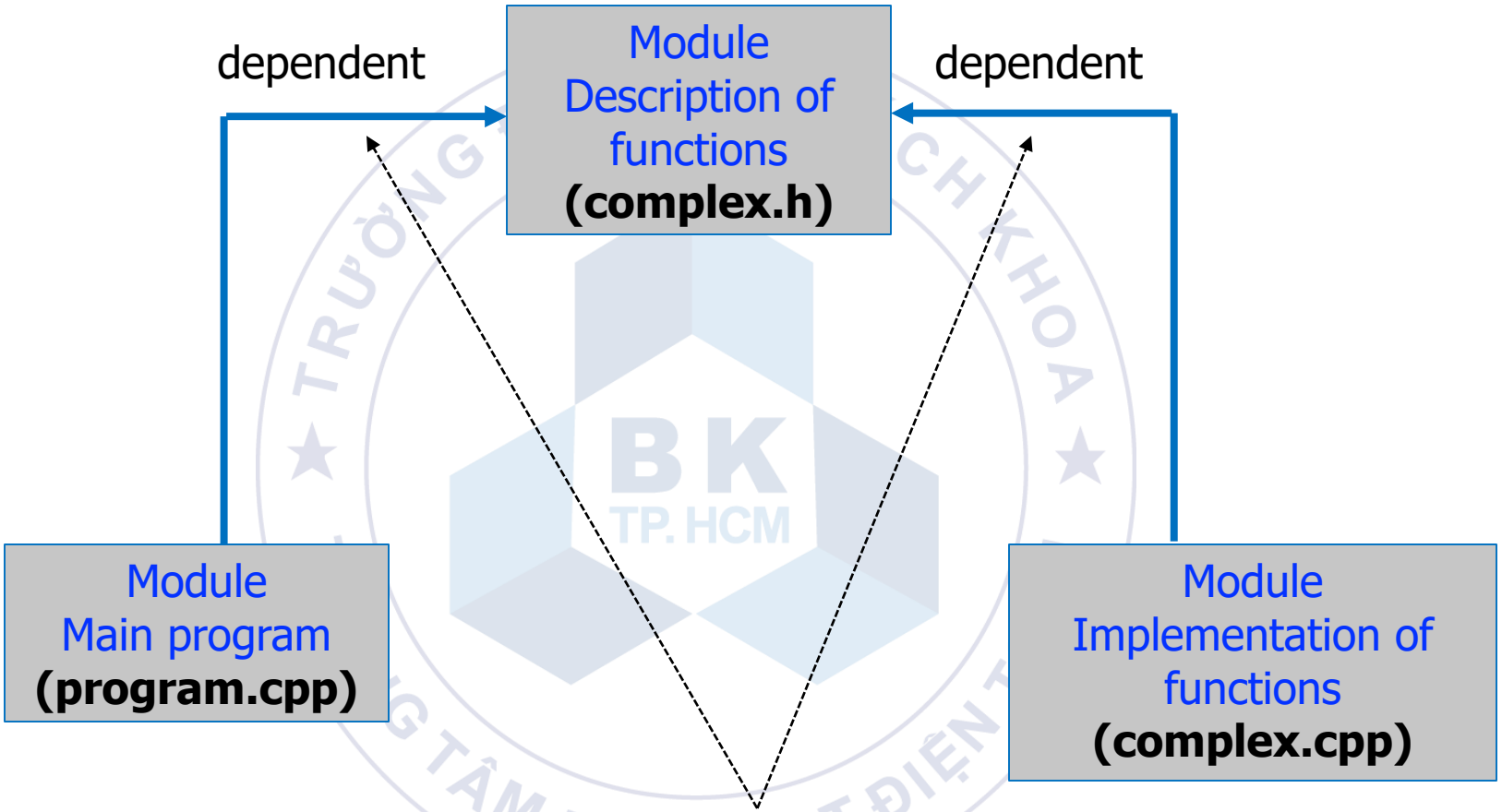
- The function that computes the angle of the complex number
  - Need to define constants
    - Pi
    - Constant represents “indeterminate” value
    - Can be represented by 2 * PI or any value outside the range [-PI, PI].
    - It is necessary to define macros to support "==“ comparison with real numbers to avoid numerical representation errors

```
#define PI 3.14159265
#define UN_DEF_ANGLE (2*PI)
#define EPSILON (1.0E-13)
#define equal(d1, d2) (abs((d1) - (d2)) < EPSILON)
#define RAD_2_DEG (180.0/PI)
#define DEG_2_RAD (PI/180.0)
```

Organizing source code

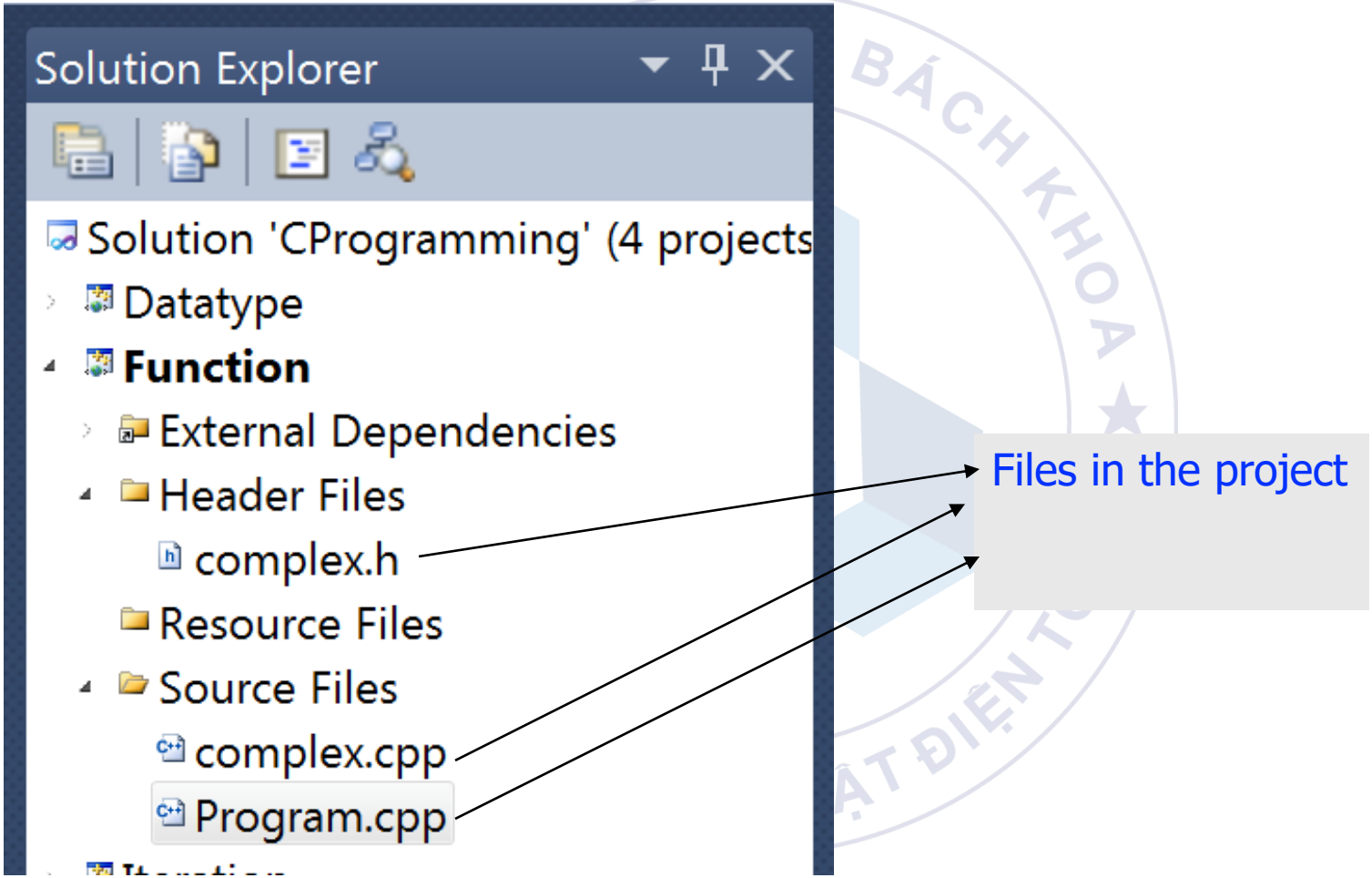
- Analysis:
  - Organize source code so that it has the following modules
    - The header file (complex.h) contains a new type description and describes functions with complex numbers
    - The source file (complex.cpp) contains the implementation of functions
    - The source file (program.cpp) contains the function “main” and uses the functions of the complex number

Organizing source code



Dependence is expressed by the directive
`#include "complex.h"` in the source code

Organizing source code



Organizing source code

File: **complex.h**

```
#if !defined(MY_MATH_HEADER)
#define MY_MATH_HEADER

#define PI 3.14159265
#define UN_DEF_ANGLE (2*PI)
#define EPSILON (1.0E-13)
#define equal(d1, d2) (abs((d1) - (d2)) < EPSILON)
#define RAD_2_DEG (180.0/PI)

typedef struct{
        double x, y;
} Complex;

double get_magnitude(Complex c);
double get_angle(Complex c);
void print_complex(Complex c);

#endif
```

Organizing source code

File: **complex.cpp**

```
#include "complex.h"
#include <math.h>
#include <iostream>
using namespace std;

double get_magnitude(Complex c){
        double mag;
        mag = sqrt(c.x*c.x + c.y*c.y);
        return mag;
}
```

Declare the use of the complex number description

Declare the use of the math functions in math.h

Declare the use cout function to print complex numbers to the screen

The definition of the function get_magnitude: Returns the magnitude value of the input (complex number c)

File: **complex.cpp**

```
double get_angle(Complex c){
        double angle;
        if(c.x > 0){
                angle = atan(c.y/c.x);
        }
        if((c.x < 0) && (c.y >= 0)){
                angle = atan(c.y/c.x) + PI;
        }
        if((c.x < 0) && (c.y < 0)){
                angle = atan(c.y/c.x) - PI;
        }
        if(equal(c.x, 0.0) && (c.y > 0)){
                angle = PI/2;
        }
        if(equal(c.x, 0.0) && (c.y < 0)){
                angle = -PI/2;
        }
        if(equal(c.x, 0.0) && equal(c.y, 0.0)){
                angle = UN_DEF_ANGLE;
        }
        angle *= RAD_2_DEG;
        return angle;
}
```

The function that computes the angle of a complex number

Organizing source code

File: **complex.cpp**

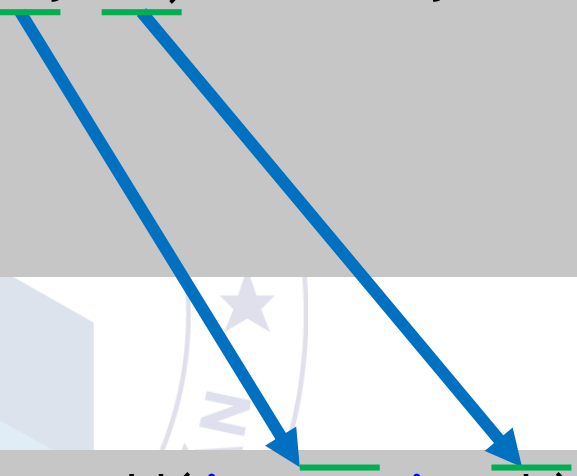
```
void print_complex(Complex c){
        cout << "[" << c.x << "," << c.y << "]";
}
```

The function that prints a complex number to the screen

Parameter passing methods

Parameters and arguments

```
int main(){
	cout << "10 + 15 = " << add(10, 15) << endl;

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



10: Is the **argument** of
the parameter a
15: Is the **argument** of
the parameter b

```
int add(int a, int b)
{
	int c;
	c = a + b;
	return c;
}
```

Parameter passing methods

- There are three methods to pass parameter to functions
  - Pass parameters by value or *pass by value*
  - Pass parameters by pointer or *pass by pointer*
  - Pass parameters by pointer or *pass by reference*

Parameter passing methods

- The reason why there are two types of passing
  - Pass by value
    - Used when we **DO NOT ALLOW** the called function to change the value of arguments
  - Pass by reference
    - Used when we want to **ALLOW** the called function to change the value of arguments
    - Or
      - When you do not want the program to spend too much time preparing the parameters of the called function in runtime (that is, the program **HAS TO COPY** the value of the arguments to the parameters). For example: copying a large array can take a lot of time -> pass by reference tackles this

Parameter passing methods

The difference in syntax

The asterisk (*) indicates which parameters will be passed by address

```
void swap(int a, int b){
}
```

a and **b** will be passed by value

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

a and **b** will be passed by pointer

Parameter passing methods

Function call: passed by value

```
#include <iostream>


void swap(int a, int b){
}
int main(){
        int x = 10, y = 100;

        swap(x, y);
        swap(10, 100);
        swap(x + 10, y*2);

        return EXIT_SUCCESS;
}
```

a and **b** will be passed by value

The argument could be: **Variable**

The argument could be: **Constant**

The argument could be: **expression** that has the same type as the parameter type

Parameter passing methods

Function call: passed by value

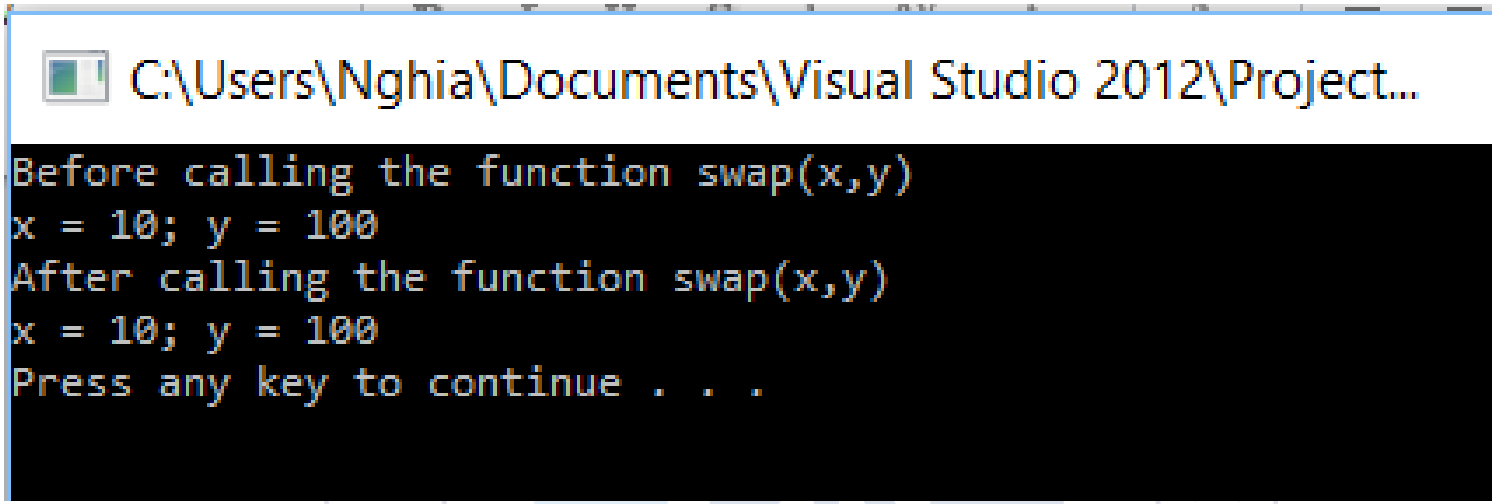
```
#include <iostream>
using namespace std;

void swap(int a, int b){
        int t = a;
        a = b;
        b = t;
}
int main(){
        int x = 10, y = 100;
        cout << "Before calling the function swap(x,y)\n";
        cout << "x = " << x << "; y = " << y << endl;
        swap(x, y);
        cout << "After calling the function swap(x,y)\n";
        cout << "x = " << x << "; y = " << y << endl;

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

The swap function swaps the values of two variables a and b through the temporary variable

Function call: x and y are the corresponding argument for a and b

Parameter passing methods

Function call: passed by value



The values of x and y stay the same after the swap function (x, y) is done.
Reason: the program has done the following while running

- **COPY** the values of arguments x and y to the memory cells of the parameters **a** and **b**, respectively
- Function **swap(int a, int b)** only swaps the values of **a** and **b** (not x and y) and ends it

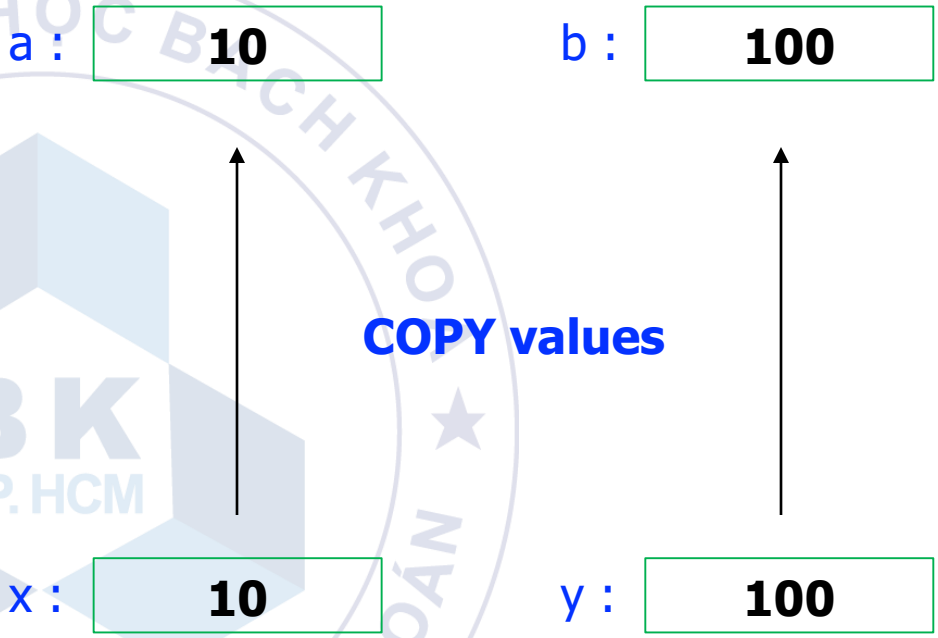
- => **The memory cells of x and y are unaffected**

Parameter passing methods

Function call: passed by value

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

```
int main(){
        int x = 10, y = 100;
        swap(x, y);
        return EXIT_SUCCESS;
}
```



The function call `swap(x, y)` in function "main" causes the variables a and b to hold the values of x and y respectively: 10 and 100

Parameter passing methods

Function call: passed by value

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

a : 10 b : 100

t : 10

```
int main(){
        int x = 10, y = 100;
        swap(x, y);
        return EXIT_SUCCESS;
}
```

x : 10 y : 100

`int t = a;` creates t and make it stores the value of a: 10

Parameter passing methods

Function call: passed by value

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

a : 100 b : 100

t : 10

```
int main(){
        int x = 10, y = 100;
        swap(x, y);
        return EXIT_SUCCESS;
}
```

x : 10 y : 100

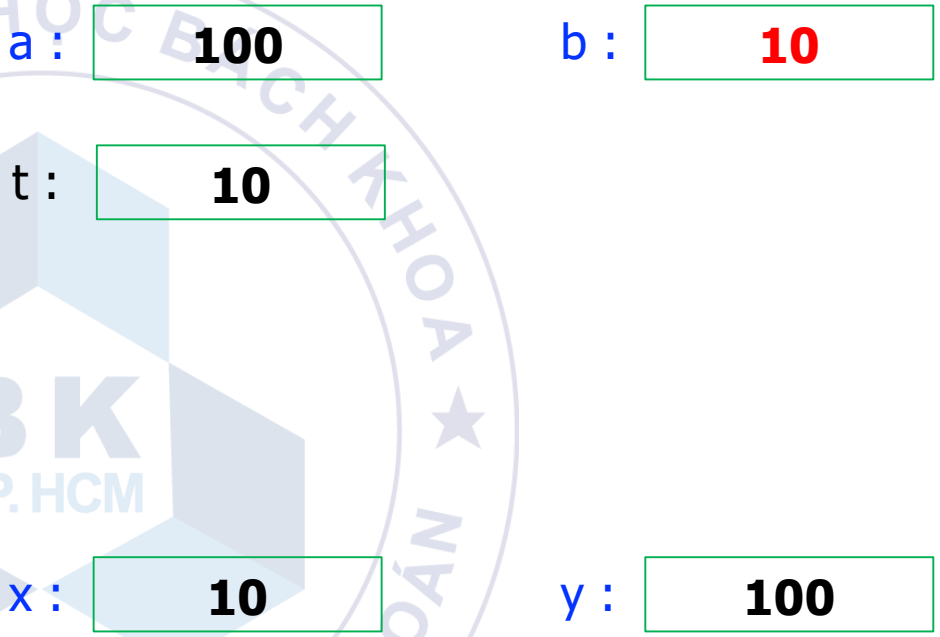
a = b; assigns the value of b to a: 100

Parameter passing methods

Function call: passed by value

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

```
int main(){
        int x = 10, y = 100;
        swap(x, y);
        return EXIT_SUCCESS;
}
```



b = t; assigns the value of t to b: 10

Parameter passing methods

Function call: passed by value

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

a : 100 b : 10

t : 10

```
int main(){
        int x = 10, y = 100;
        swap(x, y);
        return EXIT_SUCCESS;
}
```

x : 10 y : 100

swap(int a, int b) finishing swapping a and b, x and y in main are left untouched. Therefore, when **swap** ends, values of x and y stay the same

Parameter passing methods

Function call: passed by address

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

void swap(int *a, int *b){
}
int main(){
        int x = 10, y = 100;

        swap(&x, &y);
        swap(10, 100);
        swap(x + 10, y*2);

        return EXIT_SUCCESS;
}
```

a and **b** will be passed by address

Argument: **CAN ONLY BE VARIABLES**

Argument: **CAN'T BE CONSTANT**

Argument: **CAN'T BE EXPRESSIONS**

Parameter passing metho

Function call: passed by address

```
#include <iostream>
using namespace std;
void swap(int *a, int *b){
        int t = *a;
        *a = *b;
        *b = t;
}
int main(){
        int x = 10, y = 100;
        cout << "Before calling swap(x,y)\n";
        cout << "x = " << x << "; y = " << y << endl;
        swap(&x, &y);
        cout << "After calling swap(x,y)\n";
        cout << "x = " << x << "; y = " << y <<  endl;
        system("pause");
        return EXIT_SUCCESS;
}
```

The swap function swaps the values of the arguments passed into a and b

Asterisk operator (*):
a is address → (*a) is the value stored in that memory address

Function call: pass the addresses of x and y into the parameters a and b, respectively.

Ampersand operator (&): used to get the address of a variable

Parameter passing methods

Function call: passed by address

```
Before calling swap(x,y)
x = 10; y = 100
After calling swap(x,y)
x = 100; y = 10
```

The values of x and y are swapped after the function swap(&x, &y) finishes.
Reason: the following tasks were done

- The addressed of **x** and **y** were **COPIED** as the values of **a** and **b**
- The function **swap(int *a, int *b)** actually swaps the values of **x** and **y** through the addresses stored in **a** and **b**: using the * operator

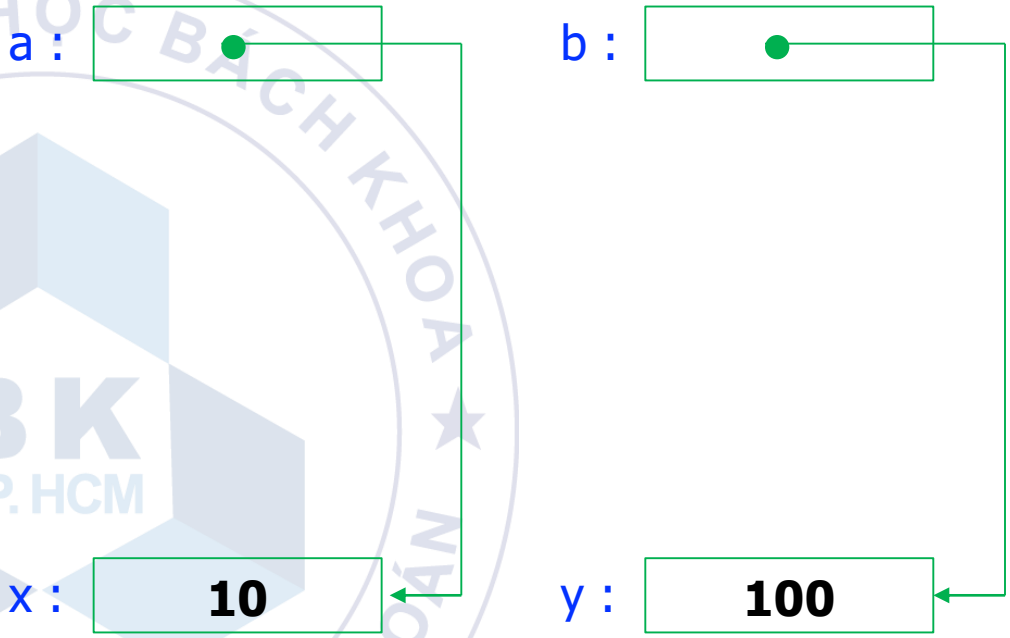
- **Note: when calling swap, we must use the & operator to get the address of a variable**

Parameter passing methods

Function call: passed by address

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

```
int main(){
        int x = 10, y = 100;
        swap(&x, &y);
        return EXIT_SUCCESS;
}
```



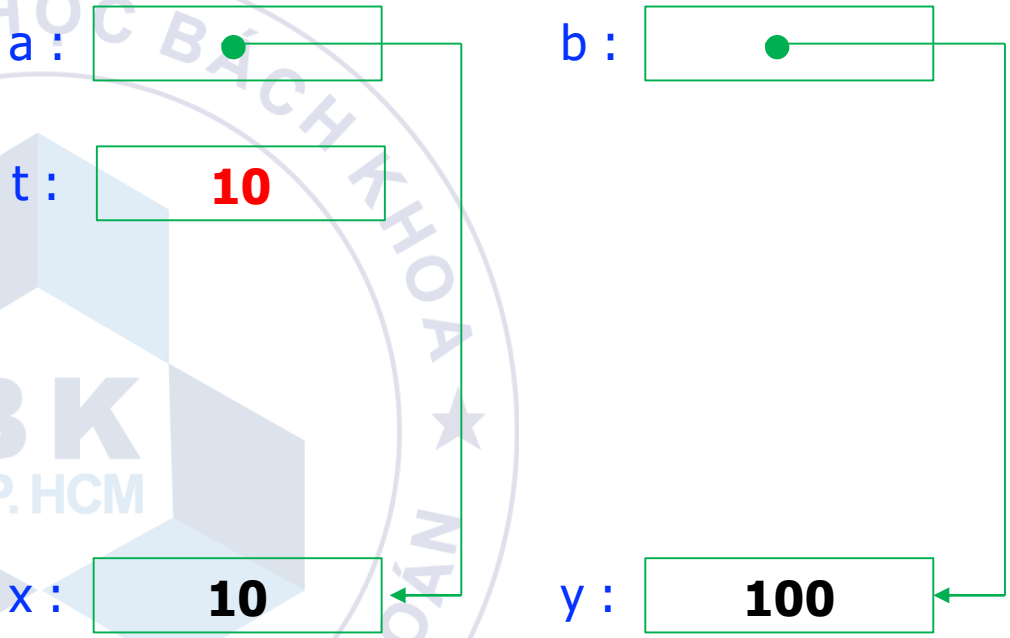
The function call `swap(&x, &y)` in `main` allows the variables `a` and `b` of `swap(int *a, int *b)` **to** store the memory addresses of `x` and `y`, respectively

Parameter passing methods

Function call: passed by address

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

```
int main(){
        int x = 10, y = 100;
        swap(&x, &y);
        return EXIT_SUCCESS;
}
```



`int t = *a;` creates t and let it take the value stored at the memory address a (i.e. x)

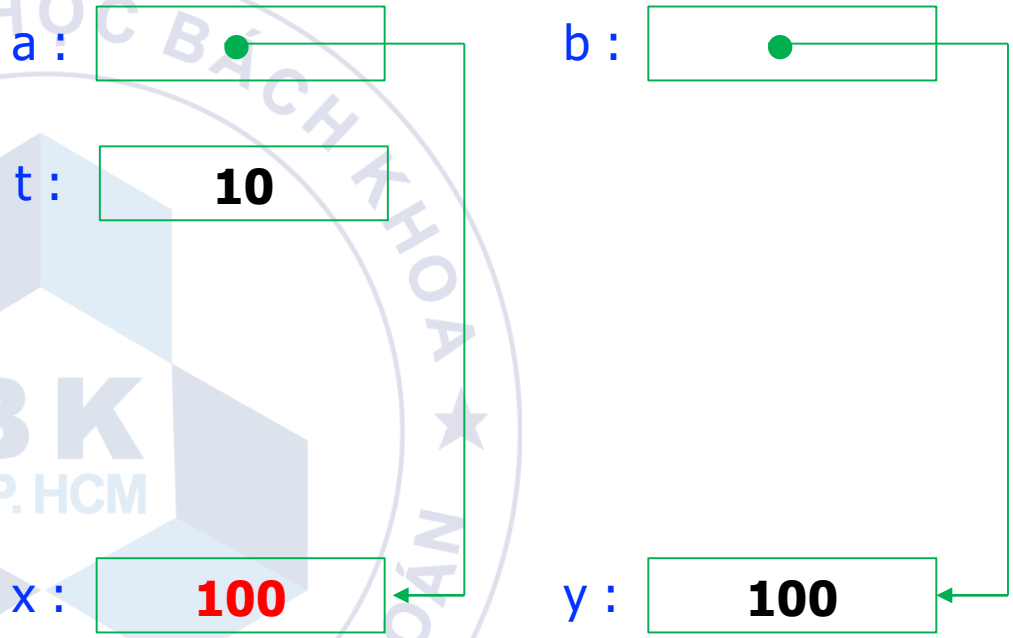
Note: **a is an address,** but ***a is an integer**

Parameter passing methods

Function call: passed by address

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

```
int main(){
        int x = 10, y = 100;
        swap(&x, &y);
        return EXIT_SUCCESS;
}
```



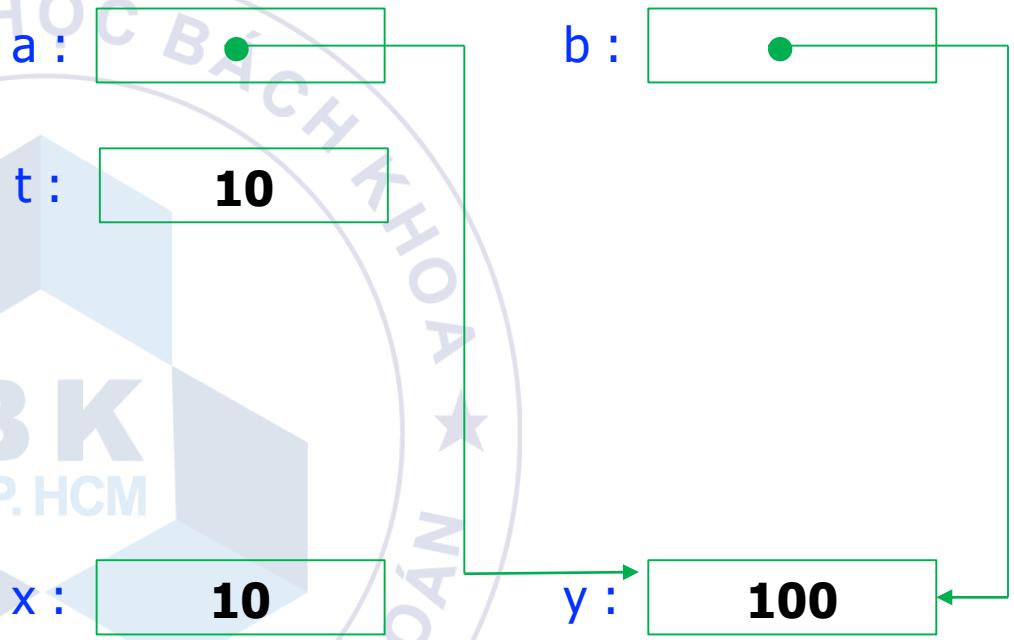
*a = *b; assigns the value stored at memory cell b (100) to the value at memory cell a: this is the same as assigning y to x.

Parameter passing methods

Function call: passed by address

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

```
int main(){
        int x = 10, y = 100;
        swap(&x, &y);
        return EXIT_SUCCESS;
}
```



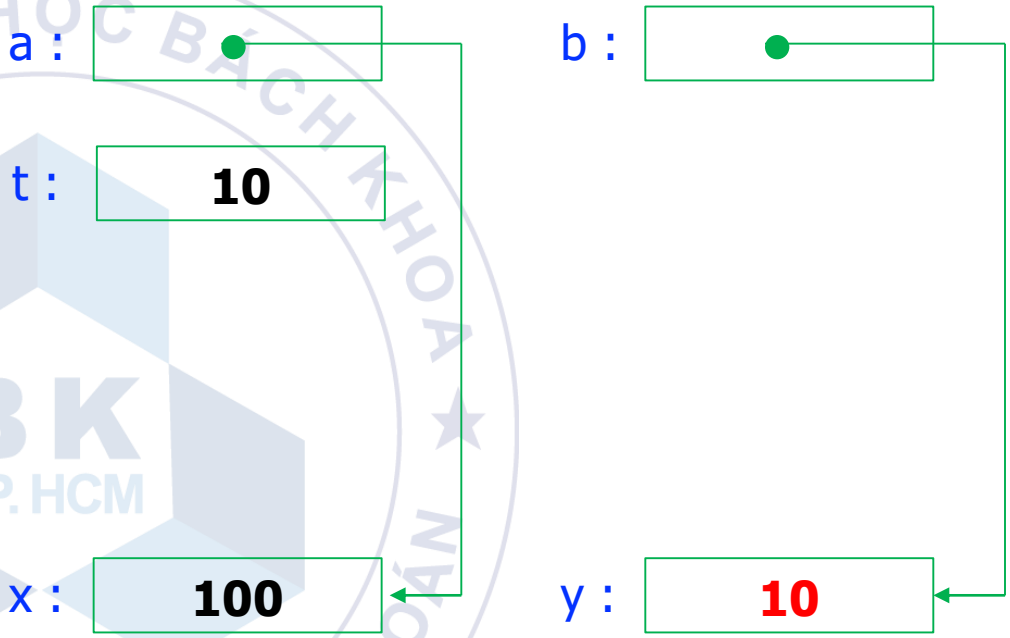
If we write a = b; this will make both a and b shares the address of y → it does not do what we want for this function: swapping values of x and y

Parameter passing methods

Function call: passed by address

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

```
int main(){
        int x = 10, y = 100;
        swap(&x, &y);
        return EXIT_SUCCESS;
}
```



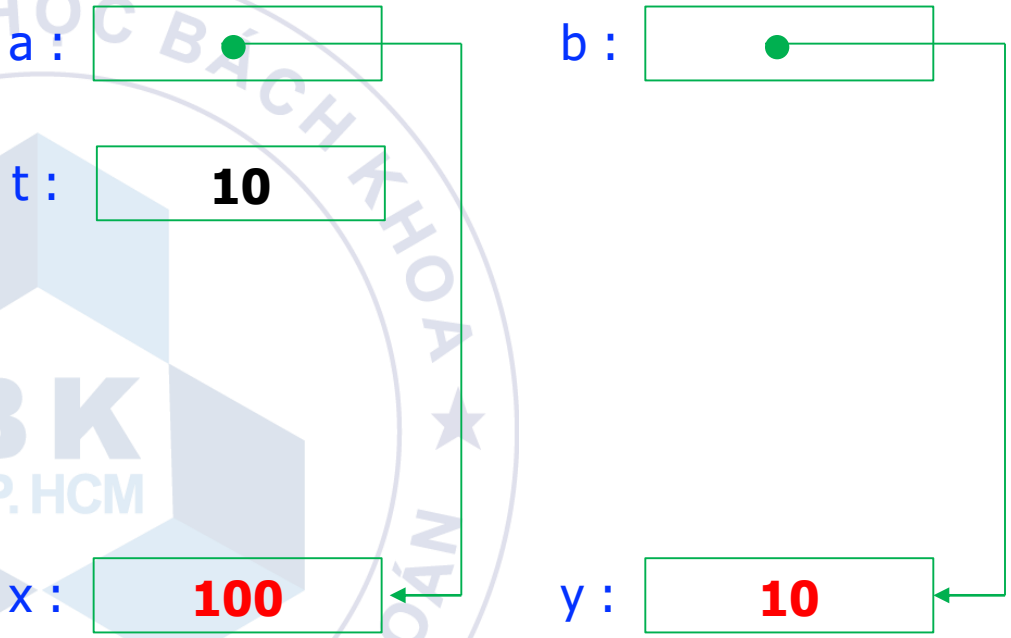
*b = t; makes the value of t COPIED into the address b; implicitly, the value of t is assigned to y

Parameter passing methods

Function call: passed by address

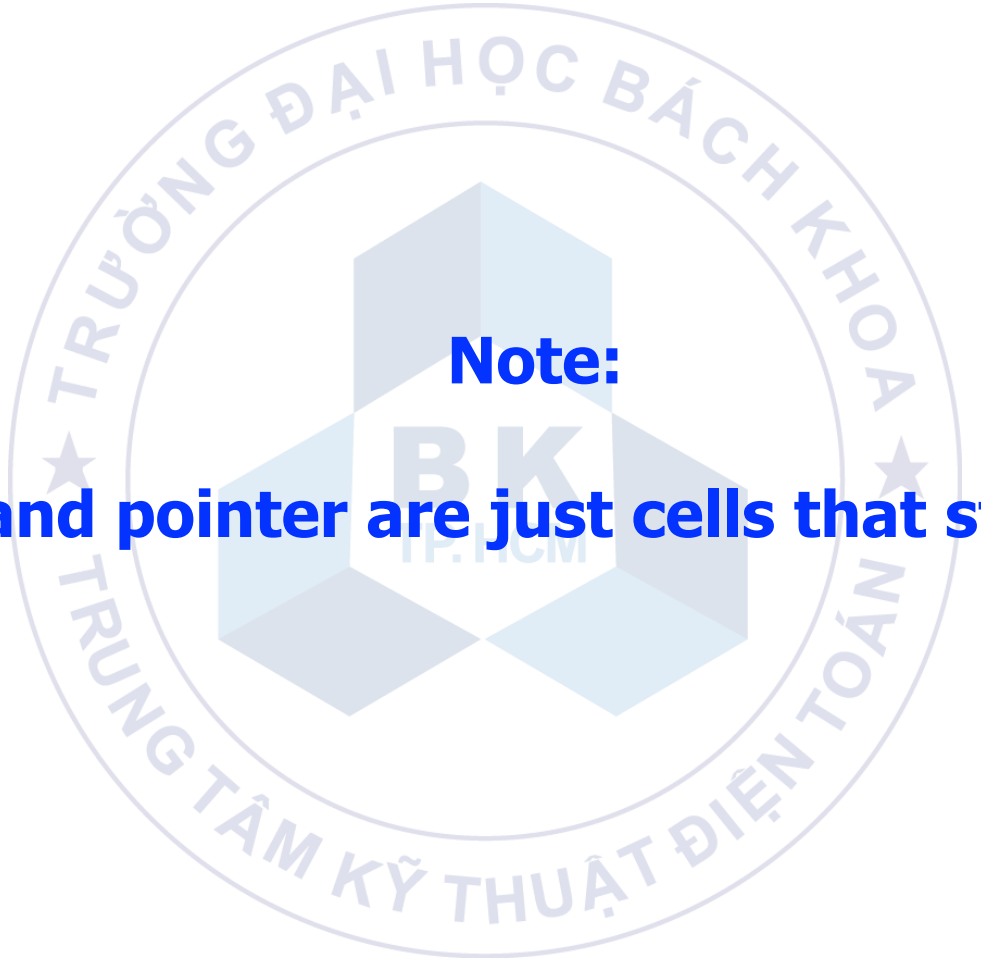
```
void swap(int *a, int *b){
        int t = *a;
        *a = *b;
        *b = t;
}
```

```
int main(){
        int x = 10, y = 100;
        swap(&x, &y);
        return EXIT_SUCCESS;
}
```



Therefore, when swap ends and the control is returned to main, the values of x and y are 100 and 10 respectively, this means that the swapping was successful.

Function and array, pointer



Note:

Both array and pointer are just cells that store addresses

Function and array, pointer

- To use function to process an array, we need to pass
  - (1) The array
  - In C++:
    To pass the array into a function:
    ⇔ Pass the address of the first element into the function
    ⇔ Pass the pointer to the first element into the function
      - C++ always pass arrays by addresses
  - (2) The number of elements in that array

Function and array, pointer

- Example:
  - Write a function to print every elements of an array
  - Analysis
    - (1) We need to pass the array to print it
    - (2) We need to pass the number of elements in that array

Function and array, pointer

- The syntax of an array parameter

```
void print_array1(int arr[MAX_SIZE], int size){
}
void print_array2(int arr[], int size){
}
void print_array3(int *arr, int size){
}
```

In all of the three functions above, the first parameters are actually just integer pointers.
The second parameter: size is the number of elements of arr.

Function and array, pointer

- The syntax of an array parameter

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE  100
void print_array1(int arr[MAX_SIZE], int size){}
void print_array2(int arr[], int size){}
void print_array3(int *ptr, int size){}

int main(){
        int size = 5;
        int a[MAX_SIZE];
        for(int i=0; i<size; i++) a[i] = i*i;

        print_array1(a, size);
        print_array2(a, size);
        print_array3(a, size);

        return EXIT_SUCCESS;
}
```

Function call:

- First argument: **a**

Pass the address of the first element of **a** into the function (passed by address)

- Second argument: **size**

(passed by value)

Function and array, pointer

- Similarly, if the elements are objects of a pre-defined struct

```
typedef struct{
      char code[10];
      char name[50];
      float gpa;
} Student;

typedef struct{
      double x,y,z;
} Point3D;
```

Function and array, pointer

- Then, the functions that print an array of such type (Student or Point3D) can have the following syntaxes:

```
void print_array1(Student arr[MAX_SIZE], int size);
void print_array2(Student arr[], int size);
void print_array3(Student *arr, int size);

void print_array1(Point3D arr[MAX_SIZE], int size);
void print_array2(Point3D arr[], int size);
void print_array3(Point3D *arr, int size);
```

Function and array, pointer

Prevent a function from modifying an array

- Use the `const` keyword before the type keyword

```
void print_array1(const Student arr[MAX_SIZE], int size);
void print_array2(const Student arr[], int size);
void print_array3(const Student *arr, int size);

void print_array1(const Point3D arr[MAX_SIZE], int size);
void print_array2(const Point3D arr[], int size);
void print_array3(const Point3D *arr, int size);
```

Function and array, pointer

Prevent a function from modifying an array

- Use the `const` keyword before the type keyword

```
void print_array3(const Point3D *arr, int size){
    arr[0].x = 100;
}
```

const Point3D *arr

Error: expression must be a modifiable lvalue

If const was used and we still try to modify any element of the array, the compiler will raise an error

Error: "**expression must be a modifiable lvalue**"
Meaning: arr must not be used in the left-hand side of the assignment expression

Inline function

- What is an inline function?
  - Is a function with the "`inline`" keyword written before the return type keyword:

```
inline void print_array1(const Point3D arr[MAX_SIZE], int size);
inline void print_array2(const Point3D arr[], int size);
inline void print_array3(const Point3D *arr, int size);
```

`Inline` keyword

Return type

Inline function

- The usefulness of inline functions
  - Non-inline functions:
    - Each function call needs to perform the following procedures:
      - Copy parameters
      - Storing registers
      - Loading registers
      - Release parameters
      - V.v

Inline function

- The usefulness of inline function
  - Inline function
    - Instead of performing the calling and returning procedures in the case of a normal function, the codes inside the body of an inline function is directly inserted into the caller function
    - => Save calling and returning costs
    - => But this will increase the size of the executable file (.exe) if the inline function is called multiple times
    - => We should only use inline function when we need to optimize the program for speed

Function pointer

- What is function pointer
- Application of function pointer
- Function pointer declaration
- Calling a function through a pointer
- Define function pointer data type
- Passing function pointer
- Function pointer array

Function pointer

What is function pointer

- The pointer to function fx is the address of the first statement to be executed in fx

- How to get the address of a function?
  - **The name of a function is also its address**

Function pointer

Application of function pointer

- If there is no function pointer
  - The function is called through its name in the source code
  - ➔ We need to know the name of the function at compiling time
- If function pointer is used
  - We can call a function through the pointer
  - ➔ There is no need to know the name of the function at compiling time
  - ➔ We only need to know the address of the function at runtime and call it
  - ➔ Our program can be more flexible

Function pointer

Application of function pointer

- Example: drawing a graph
  - The drawing function only needs to know: when it pass x as an argument, it will receive the value y from a function. It does not need to know the name of that function and how y is computed at compiling time.
  - The program creates a function table (an array storing addresses of functions). Function calls can be done through these addresses to calculate y from x.
  - This function table can be inserted with more functions or accessed in runtime
    - When the program is running, user will choose a library consisting the evaluating function (evaluate y from x), let the program know the name of the function (in string)
    - => The program can draw the graph of functions in that calculations are only known in runtime

Function pointer

Application of function pointer

- Example: Creating a library in which a function that does some tasks can only be known in the future, depending on each project.
  - Even-handling functions in libraries used for developing applications in which graphics user interface (GUI) is needed:
    - Callback function

- Example: implement the function table in C++

Function pointer

Function pointer declaration

- We need to know
  - The return data type
  - The list of data types of all parameters
  - The meaning of the set of functions satisfy the 2 constraints above
    - Decide the appropriate name for the pointer

Function pointer

Function pointer declaration – example

```
struct Student{
        char code[5];
        char name[20];
        float gpa;
};
```

Student:
Data type storing information of a student

```
void print_one_row(Student student);
```

print_one_row:
A function with void return type, input is a Student

```
void (*print_ptr1)(Student);
void (*print_ptr2)(Student) = NULL;
void (*print_ptr3)(Student) = print_one_row;
```

print_ptr1, print_ptr2, print_ptr3:

- These are function pointer variables (storing addresses of functions).

We can assign them with a function that (the name of this function is not important):

- Has void return type, return nothing
- AND has a parameter of type Student

Function pointer

Calling a function through a pointer

Source code

```
#include <stdio.h>

struct Student{
        char code[5];
        char name[20];
        float gpa;
};

void print_one_row(Student student);
void print_one_row(Student student){
        cout << student.code << " "
                << student.name << " "
                << student.gpa << "\n";
}
```

Student printing function in a single line

Function pointer

Calling a function through a pointer

Main function

```
int main(){
        void (*print_ptr3)(Student) = print_one_row;

        Student s = {"001", "Nguyen Thanh An", 9.8f};

        print_ptr3(s);

        (*print_ptr3)(s);

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

Calling a function through the pointer: the function address of a common function name in a common function call

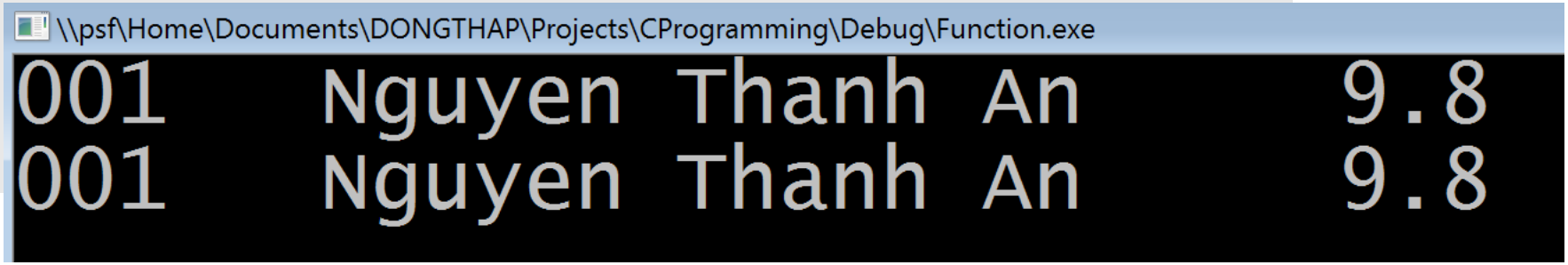
Calling a function through the pointer: using * operator

Function pointer

Calling a function through a pointer - result

Main function

```
int main(){
        void (*print_ptr3)(Student) = print_one_row;

        Student s = {"001", "Nguyen Thanh An", 9.8f};

        print_ptr3(s);

        (*print_ptr3)(s);

}
```



Function pointer

Define function pointer data type

```
struct Student{
        char code[5];
        char name[20];
        float gpa;
};

typedef void (*PrintStudentPtr)(Student);
```

The **typedef** keyword helps shortening variable declaration

PrintStudentPtr: can be used as a new data type now
It is the set of all functions with void return type, accepting a parameter with type Student

Function pointer

Define function pointer data type

```
void (*print_ptr3)(Student) = print_one_row;

PrintStudentPtr print_ptr = print_one_row;
```

`print_ptr3`, `print_ptr` :
These are variable names. They are initialized with the address of the function `print_one_row`

Thanks to `PrintStudentPtr`
Declaring `print_ptr` is the same as declaring any other data type, shorter and easier to read

Function pointer

Define function pointer data type

Complete example

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

struct Student{
        char code[5];
        char name[20];
        float gpa;
};
typedef void (*PrintStudentPtr)(Student);

void print_one_row(Student student);
void print_one_row(Student student){
        cout << student.code << " "
                << student.name << " "
                << student.gpa << "\n";
}
```

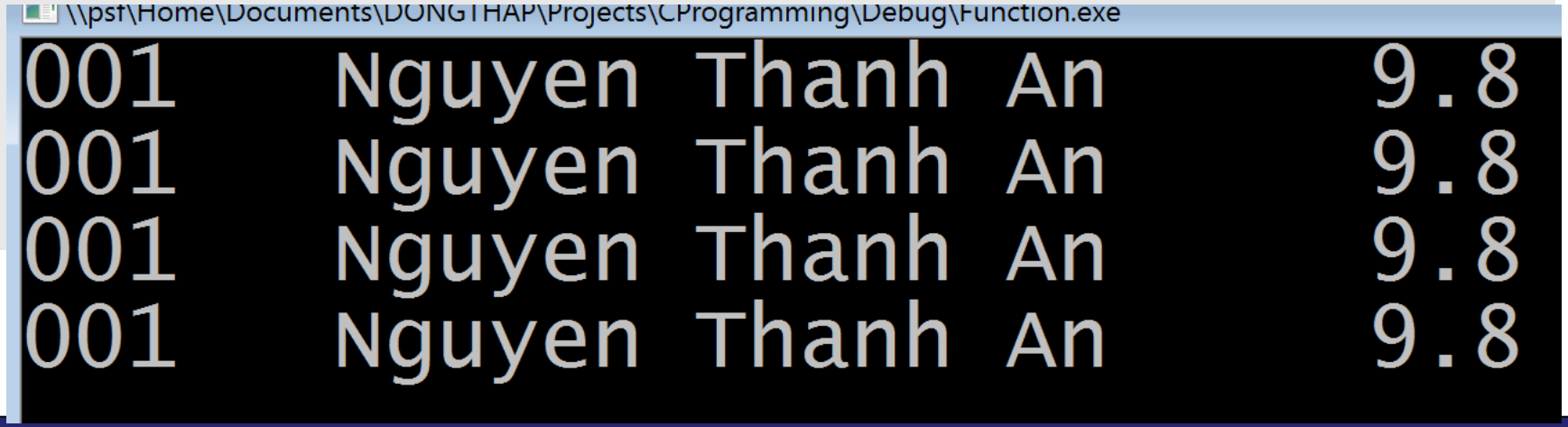
Function pointer

Define function pointer data type

Complete example

```
int main(){
        void (*print_ptr3)(Student) = print_one_row;

        PrintStudentPtr print_ptr = print_one_row;

        Student s = {"001", "Nguyen Thanh An", 9.8f};
        print_ptr3(s);
        (*print_ptr3)(s);
        print_ptr(s);
        (*print_ptr)(s);
}
```

Print 4 rows



Function pointer

Passing function pointer as a parameter – parameter declaration

```
void print_list1(Student *list, int size,
      PrintStudentPtr print_ptr);

void print_list2(Student *list, int size,
      void (*print_ptr)(Student));
```

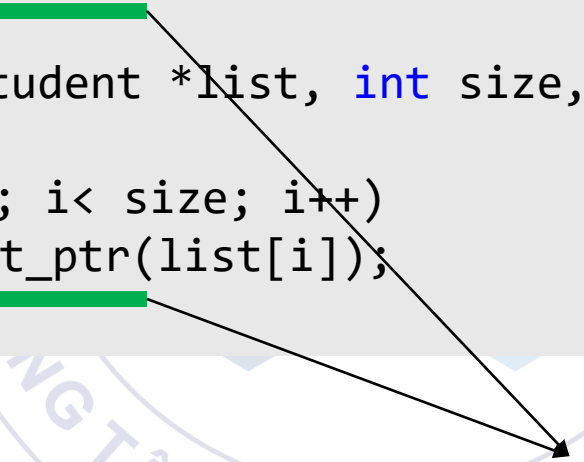
Without using function pointer data type

Using function pointer data type with typedef

Function pointer

Passing function pointer as a parameter – using parameter

```
void print_list1(Student *list, int size, PrintStudentPtr print_ptr)
{
        for(int i=0; i< size; i++)
                print_ptr(list[i]);
}
void print_list2(Student *list, int size, void (*print_ptr)(Student))
{
        for(int i=0; i< size; i++)
                print_ptr(list[i]);
}
```



Both of these are just function calls through pointers

Function pointer

Passing function pointer as a parameter – using parameter

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

struct Student{
        char code[5];
        char name[20];
        float gpa;
};
typedef void (*PrintStudentPtr)(Student);

void print_one_row(Student student);
void print_list1(Student *list, int size,
                PrintStudentPtr print_ptr);
void print_list2(Student *list, int size,
                void (*print_ptr)(Student));
```

Function pointer

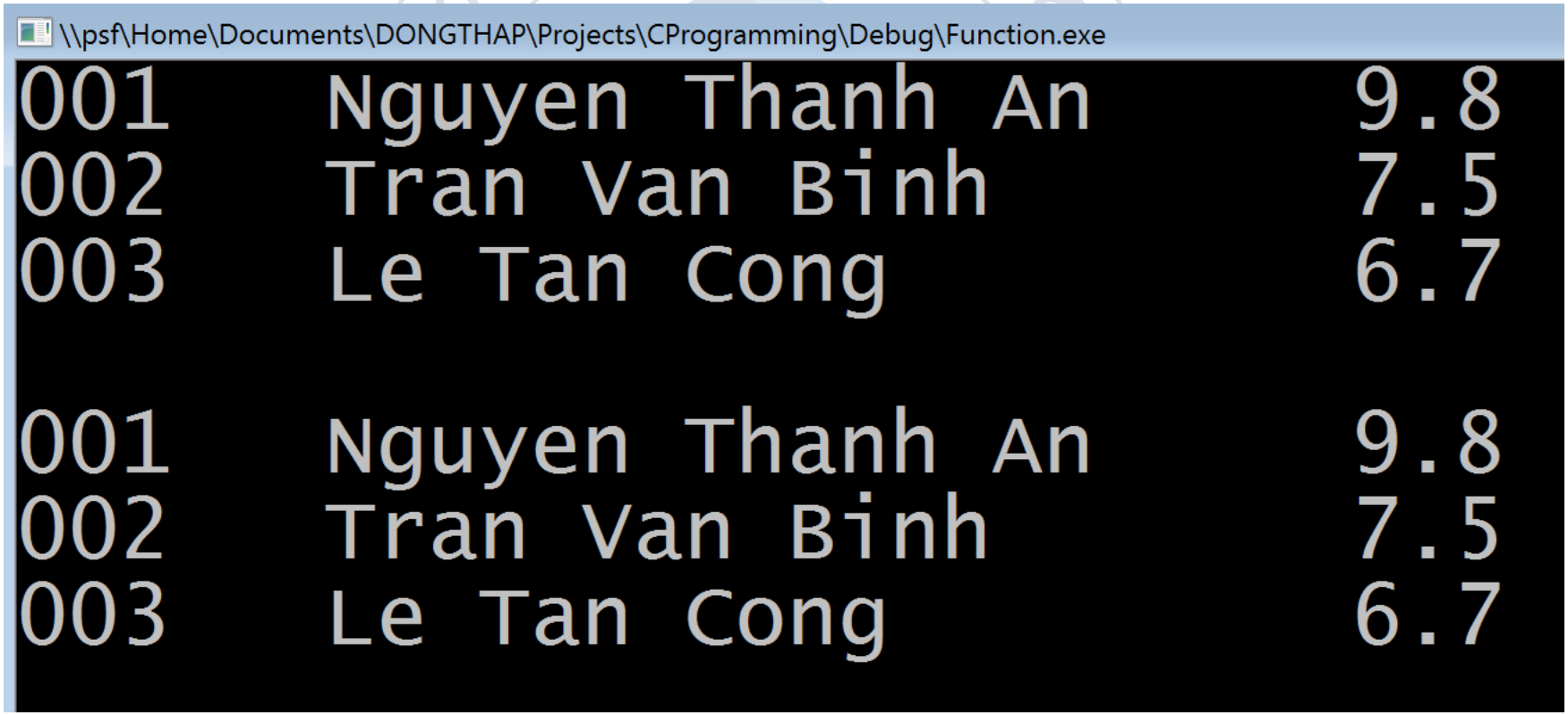
Passing function pointer as a parameter – using parameter

```
int main(){
	Student aList[] = {
		{"001", "Nguyen Thanh An", 9.8f},
		{"002", "Tran Van Binh", 7.5f},
		{"003", "Le Tan Cong", 6.7f},
	};
	PrintStudentPtr func_ptr = print_one_row;
	print_list1(aList, 3, func_ptr);
	cout << "\n";
	print_list2(aList, 3, func_ptr);

	cout << "\n\n";
	system("pause");
	return EXIT_SUCCESS;
}
```

Function pointer

Passing function pointer as a parameter – using parameter

Printing result



Function pointer

Function pointer array

```
PrintStudentPtr func_arr_ptr[10];
```

```
void (*print_ptr[10])(Student);
```

Using function pointer data type `PrintStudentPtr`

Not using function pointer data type `PrintStudentPtr`

`func_arr_ptr` and **`print_ptr`** are arrays, each has 10 function pointers.
They can be use the same way as arrays of other types

Recursive function

- A recursive function is a function that calls itself
  - Directly:
    - foo() calls foo() directly inside the body of foo() (itself)
  - Indirectly:
    - foo() calls bar() and bar() calls foo(), there can be more intermediate calls but basically some functions will be called again in the call chain

Recursive function

- Example
  - Calculate the sum 1+2+3+ .. + N
  - Assume that the name of the function is sum: sum(N) will call sum(N-1), sum(N-1) will call sum(N-2), etc.

```
int sum(int N){
      int result;
      if(N <= 0) result = 0;
      else result = N + sum(N-1);

      return result;
}
```

Recursive function

- Example
  - Calculating the factorial: 1x2x3x …x N
  - fact(N) will call fact(N-1)

```
long long fact(int N){
      int result;
      if(N <= 1) result = 1;
      else result = N*fact(N-1);

      return result;
}
```

Recursive function

- A requirement of a recursive function:
  - It must have a stopping condition
  - Example:

```
int sum(int N){
      ...
      if(N <= 0) result = 0;
      ...
}
```

This will stop the recursion

Recursive function

- A requirement of a recursive function:
  - It must have a stopping condition
  - Example:

```
long long fact(int N){
        ...
        if(N <=1) result = 1;
        ...
}
```

Recursive function

- Requirements of a recursive function
  - It must have a stopping condition
  - It must have a function call to itself
    - Function sum(N) calls sum(N-1)
    - Function fact(N) calls fact(N-1)

Recursive function

- Solving a problem using recursion
  - The solution of the problem of size N is created from the solutions of problems of smaller sizes.
  - Example:
    - To solve the problem of summing N elements:
      - We must know the solution to the summing N-1 elements problem
      - Then, how is the solution to the summing N elements problem related to the above result?

Recursive function

- Solving a problem using recursion
  - The solution of the problem of size N is created from the solutions of problems of smaller sizes.
  - Example:
    - Solution to the factorial of N
      - Assume that we know the factorial of N-1
      - How can we produce the solution to the factorial of N knowing the factorial of N-1?

Recursive function

- Solving a problem using recursion
  - The solution of the problem of size N is created from the solutions of problems of smaller sizes.
  - Example:
    - Finding the N-th Fibonacci number
      - F(1) = 1
      - F(2) = 1
      - N >2: F(N) = F(N-1) + F(N-2)
    - Solution
      - Assume that F(N-1) and F(N-2) are known
      - How can the solution of F(N) be made from the above assumption?

Recursive function

- Solving a problem using recursion
  - Example:
    - Hanoi tower
      - Moving a stack of disks from the first rod to the last
      - The disk above MUST ALWAYS BE SMALLER than the one below