University of Glasgow

电子科技大学 University of Electronic Science and Technology of China

UESTC1005 - Introductory Programming

Pointers & Structures

Lecture 12

Dr Ahmed Zoha

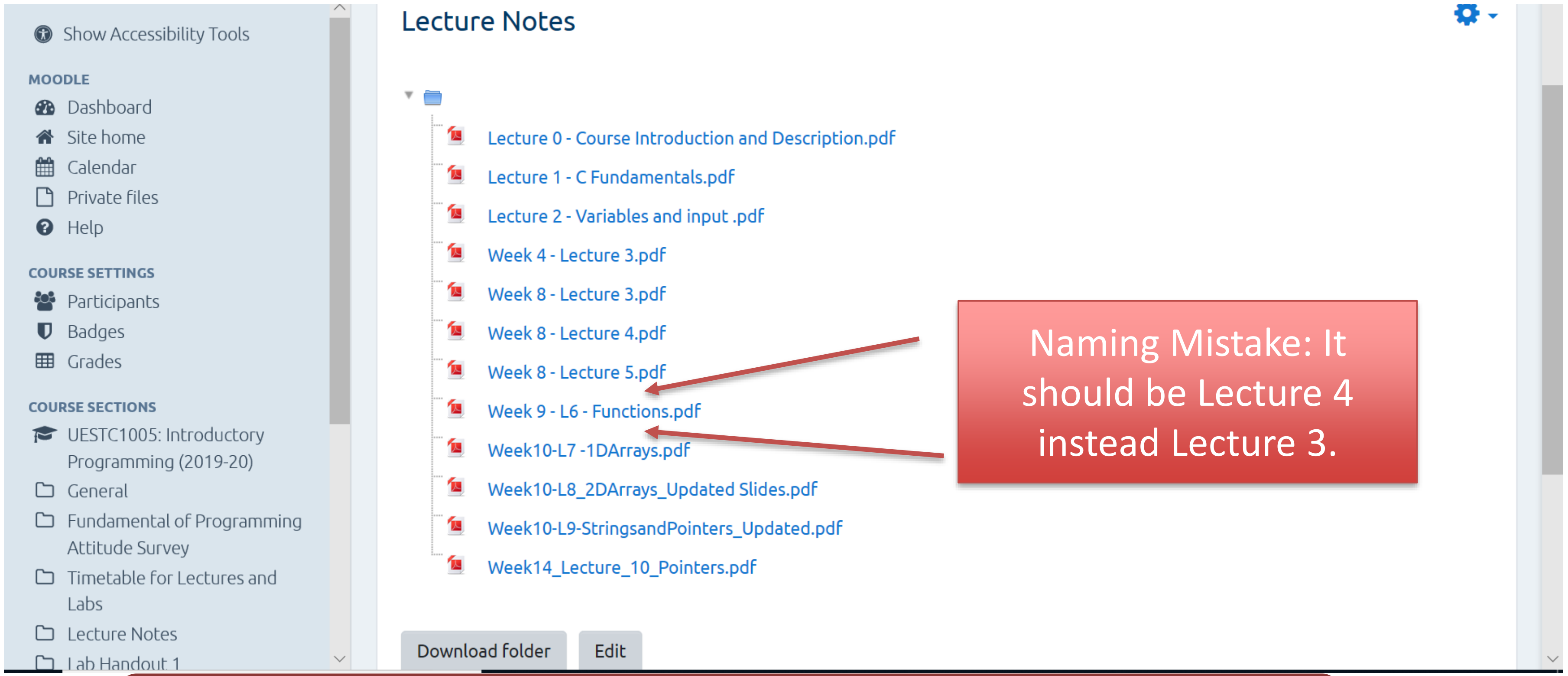
Lecturer in Communication Systems

School of Engineering

University of Glasgow

Email: ahmed.zoha@Glasgow.ac.uk

Error in the Sequence of Lecture Sequence on Moodle



Show Accessibility Tools

MOODLE
- Dashboard
- Site home
- Calendar
- Private files
- Help

COURSE SETTINGS
- Participants
- Badges
- Grades

COURSE SECTIONS
- UESTC1005: Introductory Programming (2019-20)
- General
- Fundamental of Programming Attitude Survey
- Timetable for Lectures and Labs
- Lecture Notes
- Lab Handout 1

Lecture Notes

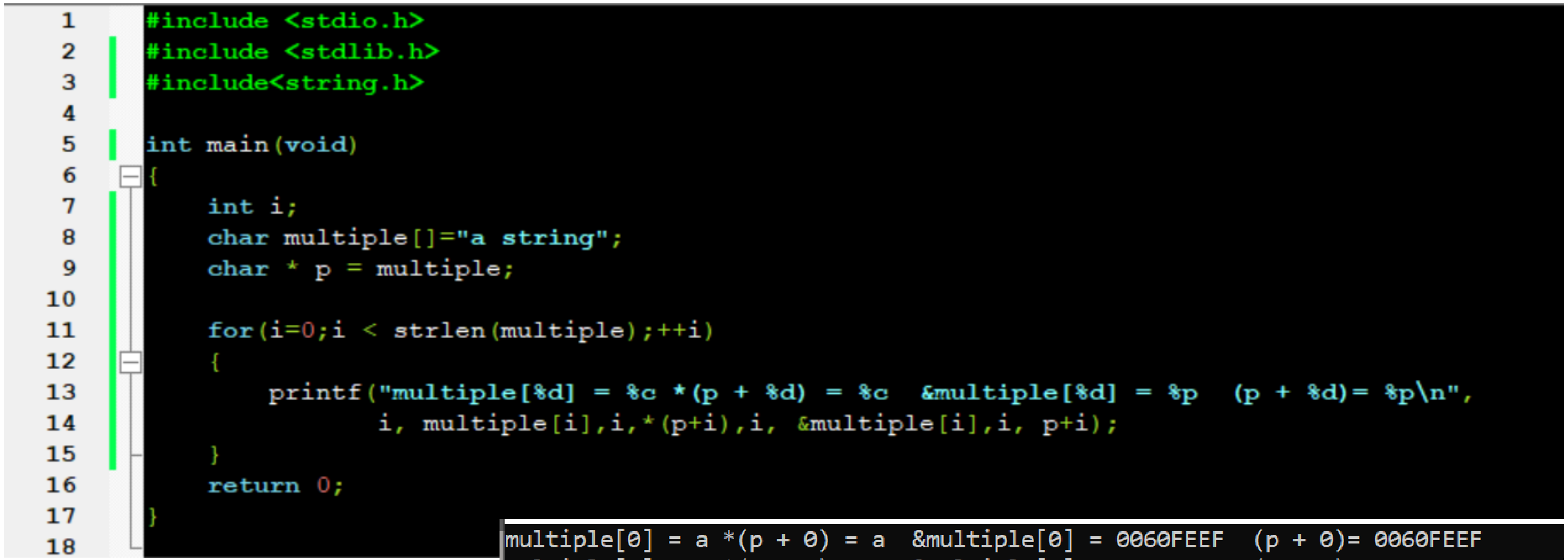
- Lecture 0 - Course Introduction and Description.pdf
- Lecture 1 - C Fundamentals.pdf
- Lecture 2 - Variables and input .pdf
- Week 4 - Lecture 3.pdf
- Week 8 - Lecture 3.pdf
- Week 8 - Lecture 4.pdf
- Week 8 - Lecture 5.pdf
- Week 9 - L6 - Functions.pdf
- Week10-L7 -1DArrays.pdf
- Week10-L8_2DArrays_Updated Slides.pdf
- Week10-L9-StringsandPointers_Updated.pdf
- Week14_Lecture_10_Pointers.pdf

Download folder | Edit

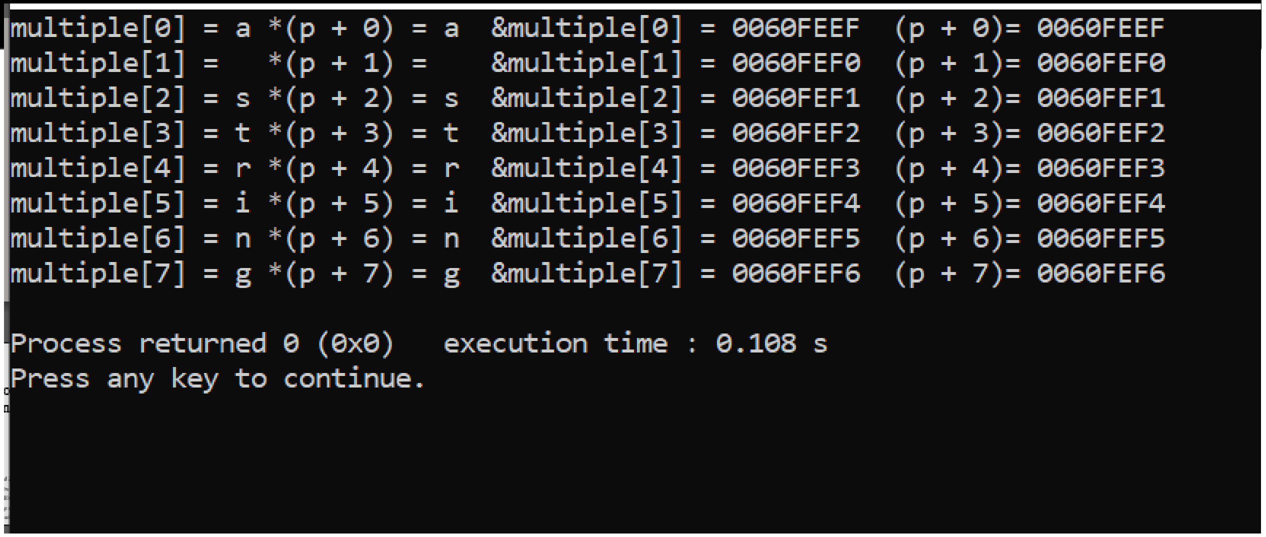
Naming Mistake: It should be Lecture 4 instead Lecture 3.

So today's lecture is Lecture 12 not 11. I will update the moodle with right sequence numbers for all the lectures to avoid any confusion

Pointers Cont’d (Example of Pointer Arithmetic's)

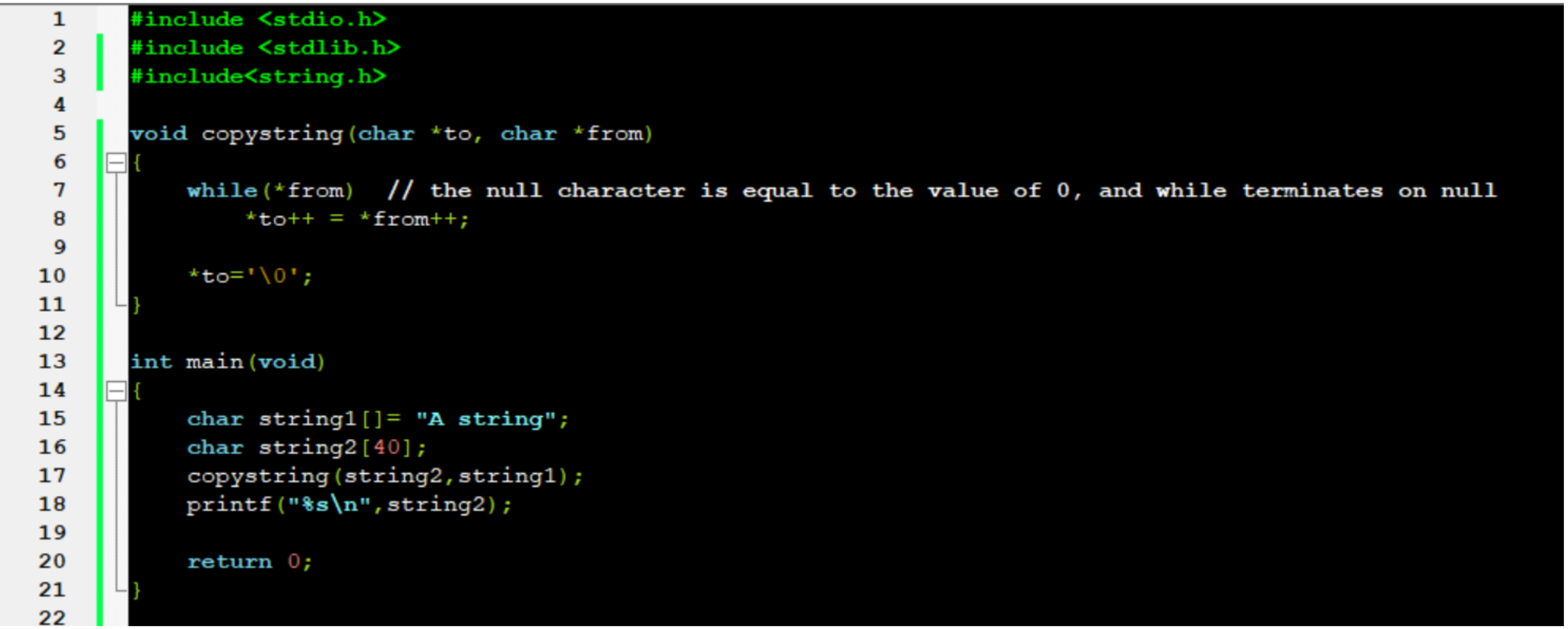


```c
#include <stdio.h>
#include <stdlib.h>
#include<string.h>

int main(void)
{
    int i;
    char multiple[]="a string";
    char * p = multiple;

    for(i=0;i < strlen(multiple);++i)
    {
        printf("multiple[%d] = %c *(p + %d) = %c  &multiple[%d] = %p  (p + %d)= %p\n",
               i, multiple[i],i,*(p+i),i, &multiple[i],i, p+i);
    }
    return 0;
}
```

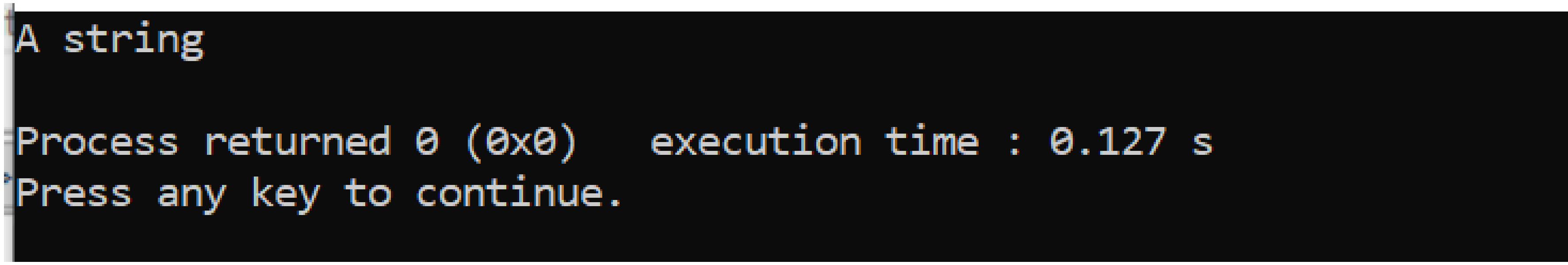


```
multiple[0] = a *(p + 0) = a  &multiple[0] = 0060FEEF  (p + 0)= 0060FEEF
multiple[1] =   *(p + 1) =    &multiple[1] = 0060FEF0  (p + 1)= 0060FEF0
multiple[2] = s *(p + 2) = s  &multiple[2] = 0060FEF1  (p + 2)= 0060FEF1
multiple[3] = t *(p + 3) = t  &multiple[3] = 0060FEF2  (p + 3)= 0060FEF2
multiple[4] = r *(p + 4) = r  &multiple[4] = 0060FEF3  (p + 4)= 0060FEF3
multiple[5] = i *(p + 5) = i  &multiple[5] = 0060FEF4  (p + 5)= 0060FEF4
multiple[6] = n *(p + 6) = n  &multiple[6] = 0060FEF5  (p + 6)= 0060FEF5
multiple[7] = g *(p + 7) = g  &multiple[7] = 0060FEF6  (p + 7)= 0060FEF6

Process returned 0 (0x0)   execution time : 0.108 s
Press any key to continue.
```

Pointers and String



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

void copystring(char *to, char *from)
{
    while(*from)   // the null character is equal to the value of 0, and while terminates on null
        *to++ = *from++;

    *to='\0';
}

int main(void)
{
    char string1[]= "A string";
    char string2[40];
    copystring(string2,string1);
    printf("%s\n",string2);

    return 0;
}
```

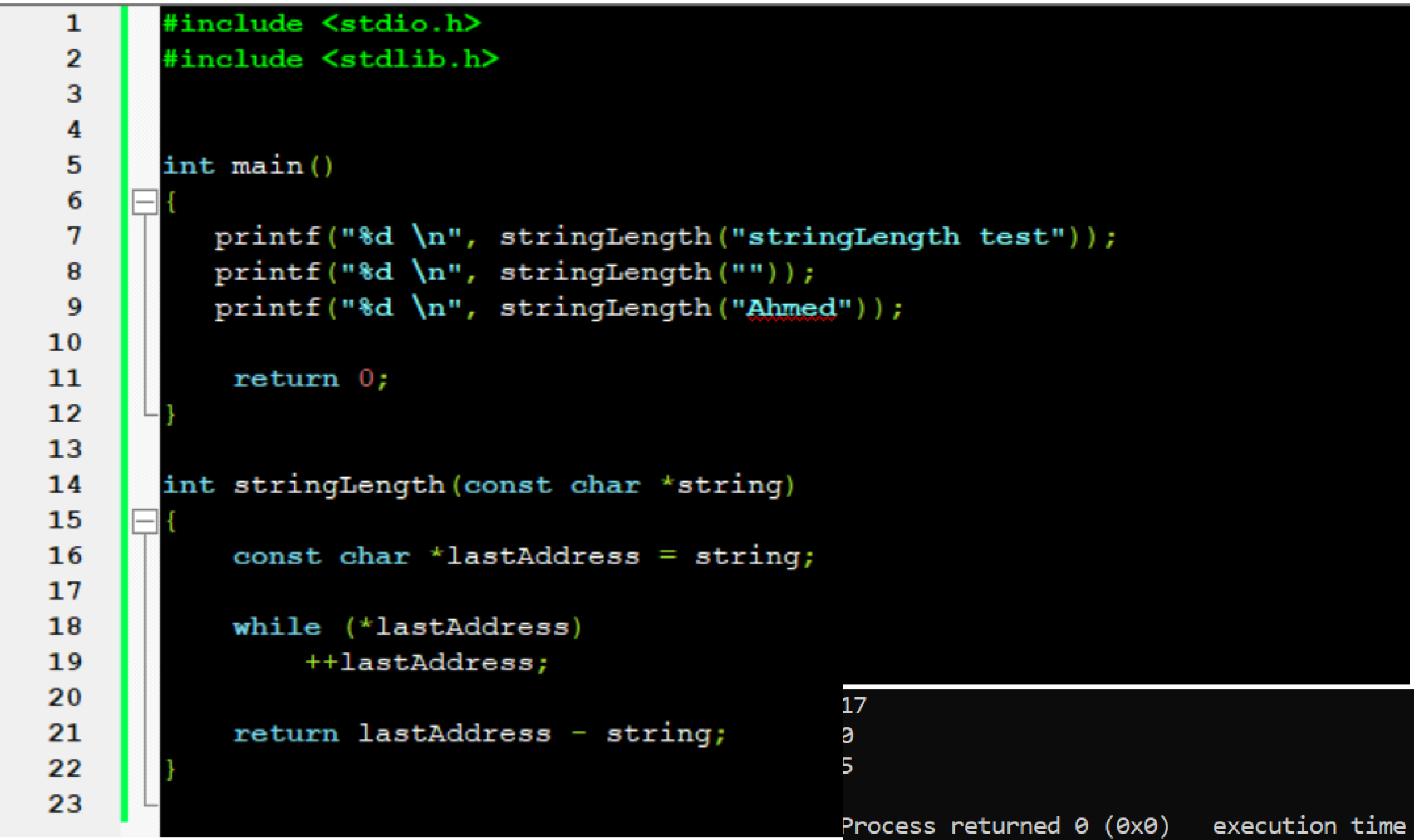


```
A string

Process returned 0 (0x0)   execution time : 0.127 s
Press any key to continue.
```

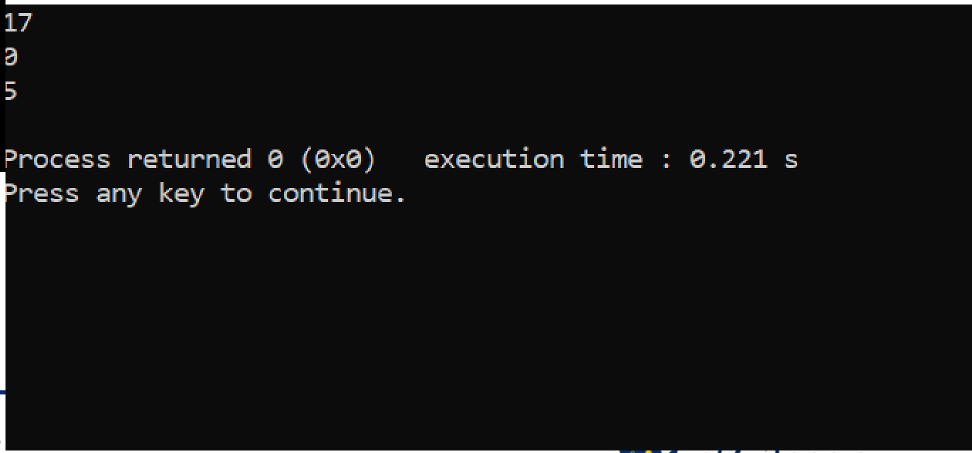
Counting Length of the String using Pointers

- Write a function that calculates the length of the string
  - The function should take as a parameter a const char pointer
  - The function can only determine the length of the string using pointer arithmetic
  - You are required to use only the while loop using the value of pointer to exit
  - The function should subtract two pointers
  - The function should return an int that is the length of the string passed to the function

Counting Length of the String using Pointers



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


int main()
{
   printf("%d \n", stringLength("stringLength test"));
   printf("%d \n", stringLength(""));
   printf("%d \n", stringLength("Ahmed"));

    return 0;
}

int stringLength(const char *string)
{
    const char *lastAddress = string;

    while (*lastAddress)
        ++lastAddress;

    return lastAddress - string;
}
```



```
17
0
5

Process returned 0 (0x0)   execution time : 0.221 s
Press any key to continue.
```

Passing Arguments to Functions

- There are two ways you can pass data to function
- Pass by value
- Pass by reference
- **Pass by Value:** Copies actual value of the argument into formal parameter
  - Changes made to parameter inside the function has no impact on the argument (local scope)
- **Pass by Reference:** When calling a function with arguments that should be modified, the addresses of the arguments are passed.

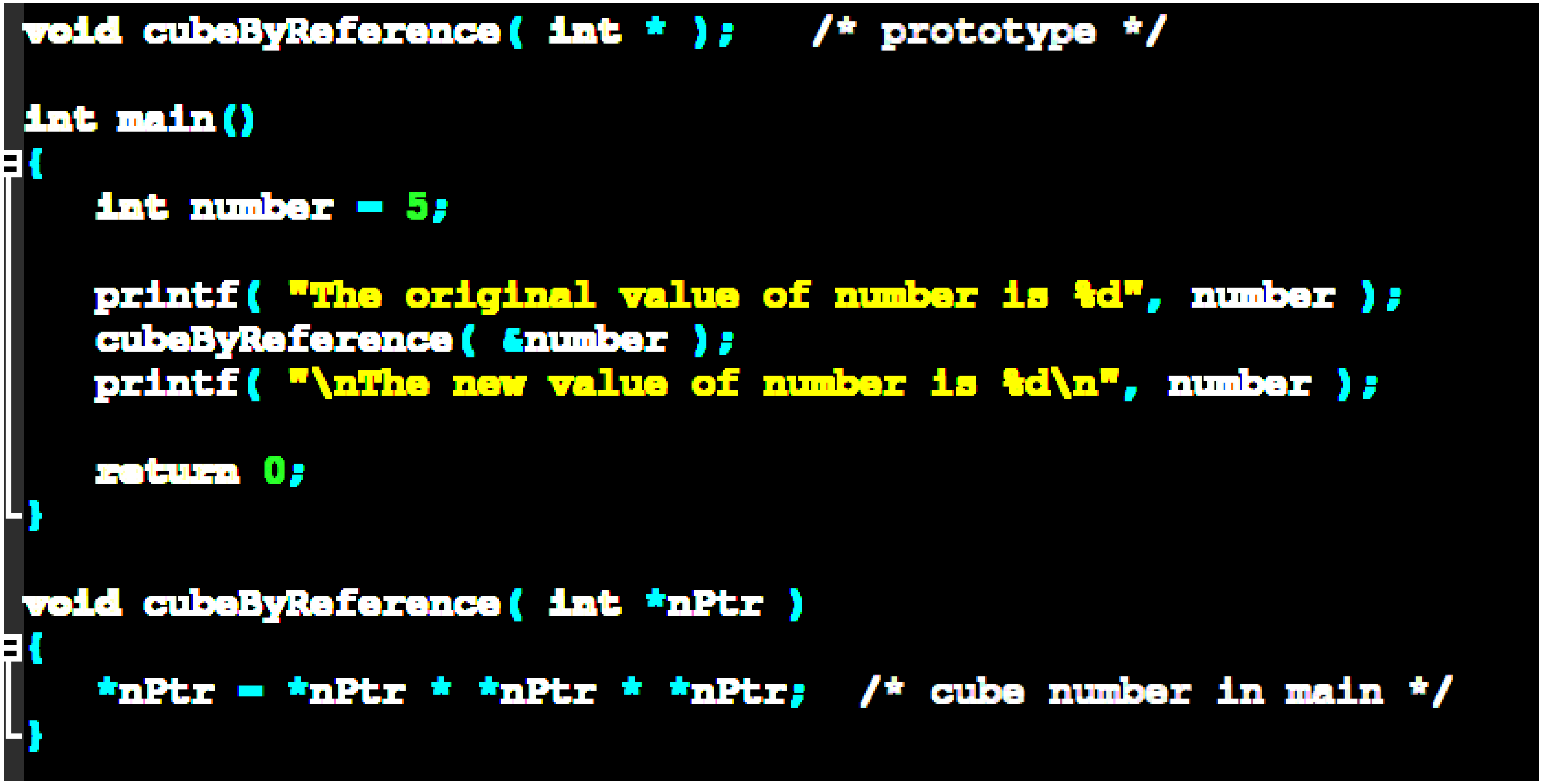
Call by Value- Example

```
int cubeByValue( int );    /* prototype */

int main()
{
    int number = 5;

    printf( "The original value of number is %d", number );
    number = cubeByValue( number );
    printf( "\nThe new value of number is %d\n", number );

    return 0;
}

int cubeByValue( int n )
{
    return n * n * n;    /* cube local variable n */
}
```

Passing data using copies of Pointers

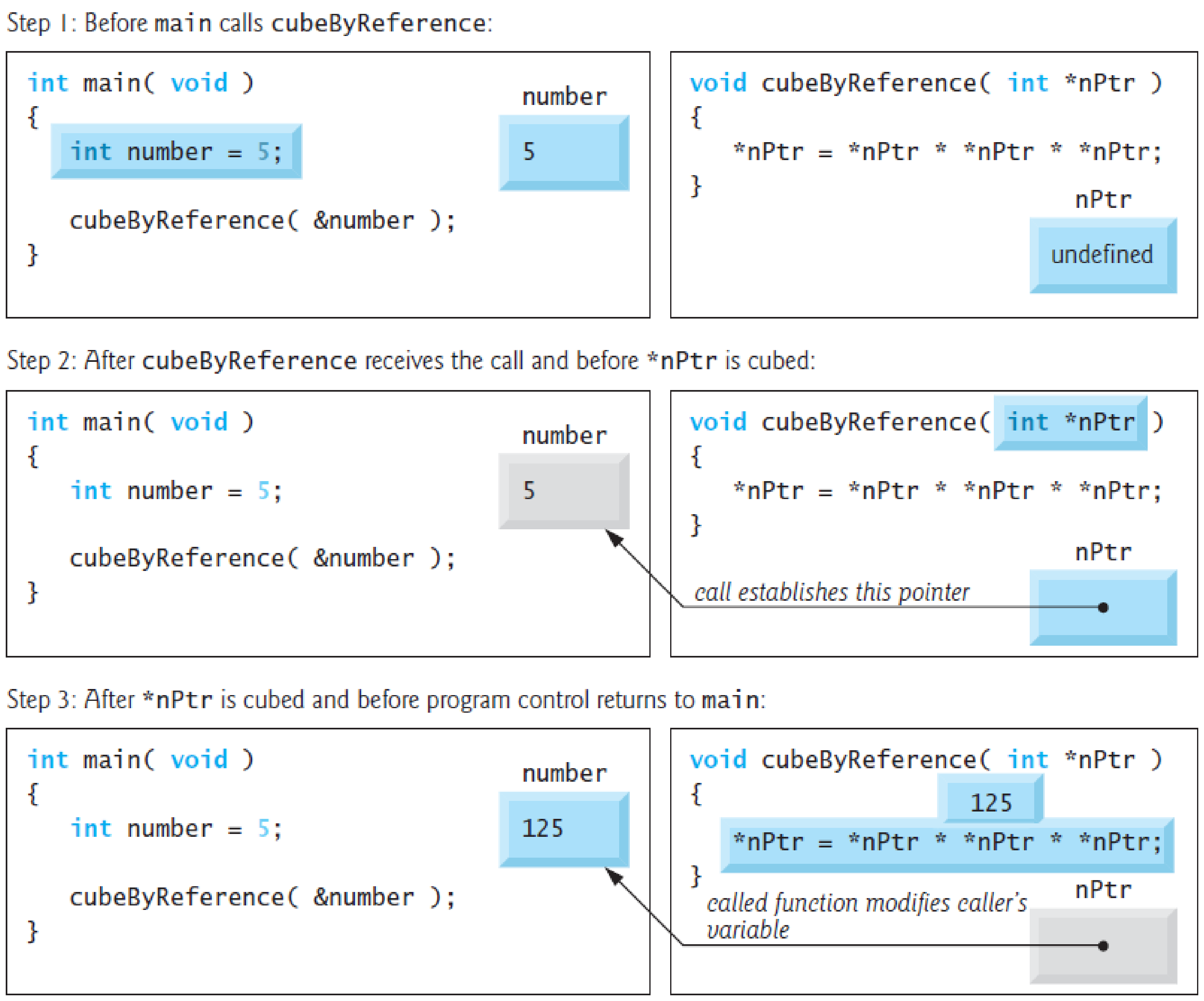
- Pointers and functions get along well with each other
  - You can pass a pointer as an argument to a function, and you can also have a function return a pointer
- Pass by reference copies an address of an argument into formal parameter (previous example)
  - The address is used to access the actual argument value
  - The changes and operations made on the argument effects it value outside the function
  - You need to declare the function parameters as pointer types *

Call by Reference- Example



```
void cubeByReference( int * );    /* prototype */

int main()
{
    int number = 5;

    printf( "The original value of number is %d", number );
    cubeByReference( &number );
    printf( "\nThe new value of number is %d\n", number );

    return 0;
}

void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
}
```

Call by Reference (Example)



Step 1: Before main calls cubeByReference:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Step 2: After cubeByReference receives the call and before *nPtr is cubed:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Step 3: After *nPtr is cubed and before program control returns to main:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Dynamic Memory Allocation

- Compiler allocates a memory whenever you define a variable in C, according to the specified data type.
- Think about a scenario when you need to dynamically allocate storage while a program is running
  - You are suppose to read the data from a file into an array, now you have following choices
    - Define the array to contain the maximum number of possible elements at compile time
    - Allocate the array dynamically using one of C’s memory allocation routine
  - What is problem with the first approach?

Dynamic Memory Allocation and Pointers

- Dynamic Memory Allocation allocates the memory as the program is executing
  - It relies on the concept of pointers
  - Creates pointers on the runtime that are just large enough to hold the amount of data you require for the task.
- Dynamic memory allocation reserves space in a memory area called heap
- Stack is another place where memory is allocated
  - Function variables and arguments are stored here
  - When the function execution ends, the space allocated to store arguments and variables is freed
- In case of heap, you need to track the memory and need to free the space when not needed.

malloc

- Common way to allocate memory is using standard c library <stdlib.h> and calling malloc ()
  - You specify the number of bytes of memory that you want to allocate in the argument
  - Returns the address of the first byte of memory that it allocated
  - Because you get an address returned, you need to cast it to a pointer

```
int * pNumber = (int*) malloc(100)
```

Problem

- You need to know specific size of memory block you want to allocate
- Assumption is that my computer requires 4 bytes to store int variable and therefore I require 100 bytes to store 25 int values.

malloc () Cont'd

- Remove the assumption and instead use sizeof()
  - int *pNumber = (int*) malloc (25*sizeof(int));
  - This makes the code much more portable
- malloc() itself returns the pointer of type void, and you need to cast it to specific type
- If the memory that you requested cannot be allocated for any reason malloc() returns NULL.

Releasing memory

- When you allocate memory dynamically, you should always release the memory when it is no longer needed
- Memory that you allocate on the heap will be automatically released when your program ends. But you should adopt a practice of releasing memory in the code
- You need to have access to the address that references the allocated block of memory in order to release it
  - You pass the pointer to free function to release the memory
  - Ex : free(pNumber); pNumber = NULL;
  - free function has formal parameter of type void*, so you can pass pointer of any type.

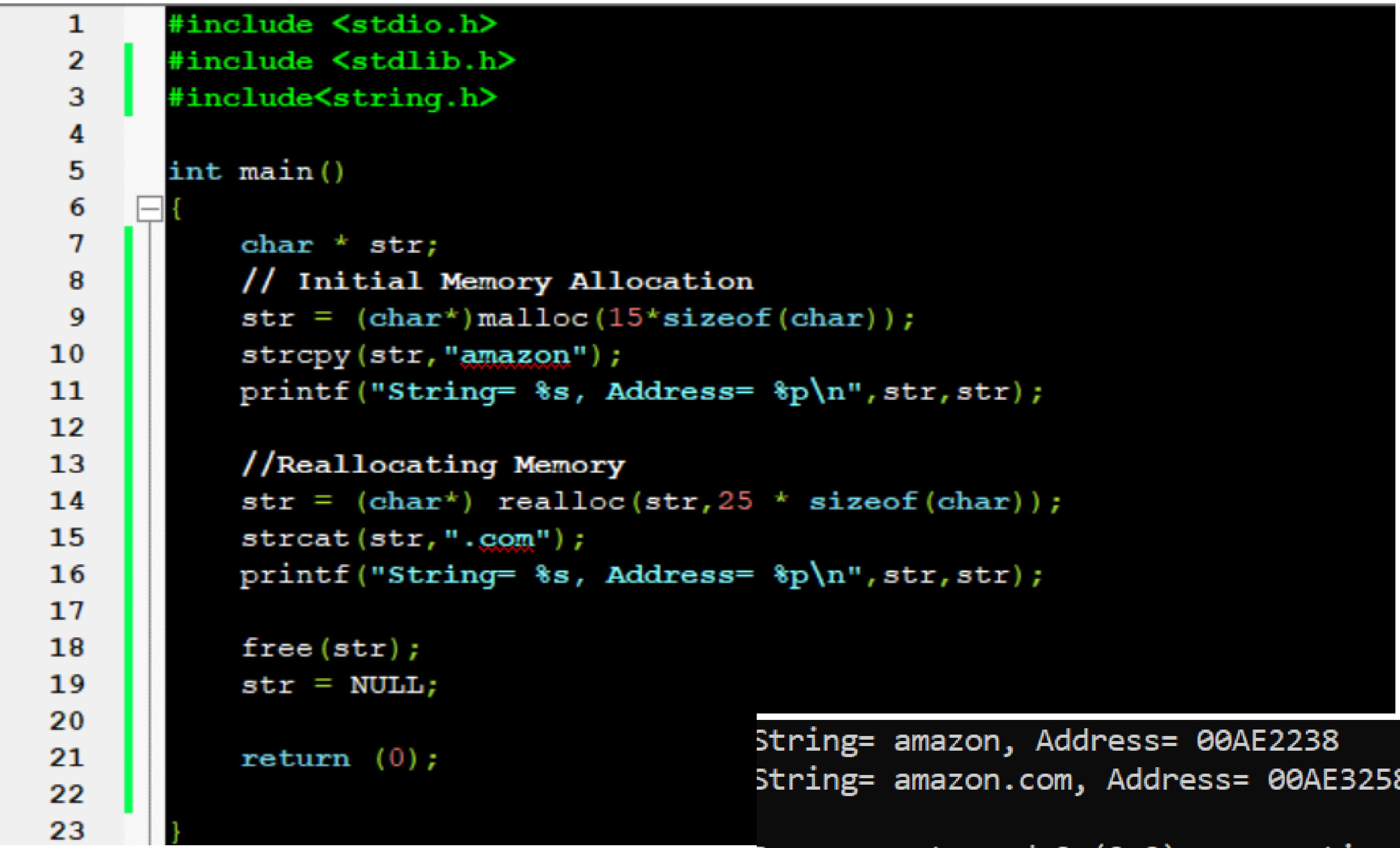
calloc()

- The calloc() function offer couples of advantages
  - It allocate memory as a number of elements of given size
  - It initializes the memory that is allocated so that all bytes are zero
- calloc() function requires two argument values
  - Number of data items for which space is required
  - Size of each data item
- Also it is declared in the <stdlib.h>
- int *pNumber = (int*) calloc(75,sizeof(int));
- In case it is not possible to allocate memory, a NULL value is returned.

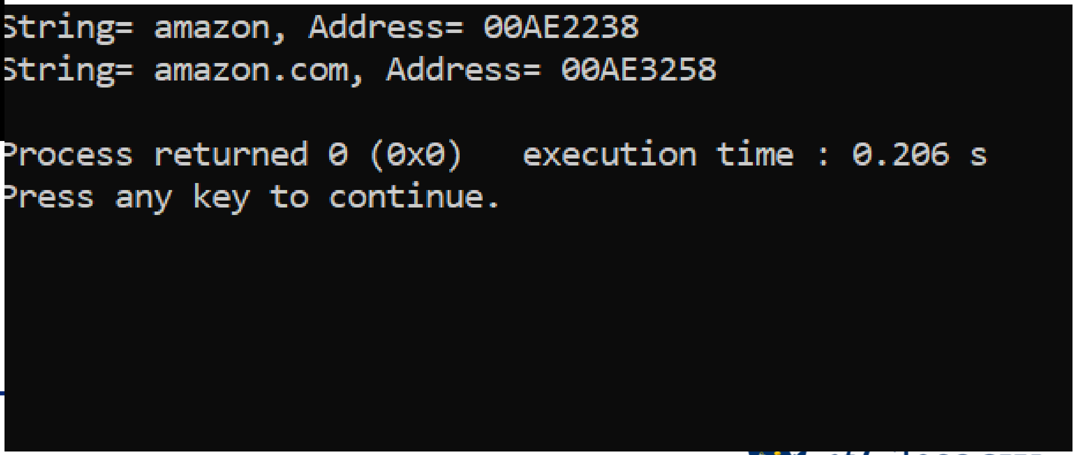
realloc()

- **realloc()** enables you to reuse or extend memory that you previously allocated using **malloc()** or **calloc()**
- It takes two arguments
  - A pointer containing an address that was previously returned by a call to malloc() or calloc()
  - The size in bytes of the new memory that you want to allocate
- Memory allocation is done as specified by the user
  - Transfer the contents of the previously allocated memory referenced by the pointer that you supply as the first argument
  - Returns a void* pointer to the new memory or NULL if the operation fails for some reason.
- **It preserves the content of the original memory area**

Example



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

int main()
{
    char * str;
    // Initial Memory Allocation
    str = (char*)malloc(15*sizeof(char));
    strcpy(str,"amazon");
    printf("String= %s, Address= %p\n",str,str);

    //Reallocating Memory
    str = (char*) realloc(str,25 * sizeof(char));
    strcat(str,".com");
    printf("String= %s, Address= %p\n",str,str);

    free(str);
    str = NULL;

    return (0);

}
```



```
String= amazon, Address= 00AE2238
String= amazon.com, Address= 00AE3258

Process returned 0 (0x0)   execution time : 0.206 s
Press any key to continue.
```

struct
– for a structure

how to build more complex and useful data types in C

Structure in C

- Structure in C: A tool to group elements together
- It would be nice to pack a whole pile of different pieces of related data in a bunch, and sometimes deal with the bunch all together
- Suppose you want to store a date inside the program
- int month = 9, day = 25, year = 2015
- If you are required to store the date of purchase of items
  - You must keep track of these the above three separate variables
  - These variables are logically related, **and structures help you group them together under one name**

Creating a structure

- Structure Declaration:
  - What are the elements inside the structure
- struct keyword enables you to define a collection of variables of various types called a structure that can be treated as single unit.

```
struct tag_name {
   data type member1;
   data type member2;
   …
};
```

Can you think of a structure for date?

```
struct lib_books {
   char title[20];
   char author[15];
   int pages;
   float price;
};
```

Structure Declaration (1/3)

- There is no memory allocation for the structure declaration unless there is a pointer.
- How structures are different from array?
  - They don't have exact same data type
- struct **lib_books** declares a structure to hold the details of four fields namely title, author, pages and price. These are members of the structure. Each member may belong to different or same data type. The structure is not a variable by itself but a template.
- Afterwards, we can declare structure variables using

  struct lib_books book1, book2;

Structure Declaration (2/3)

- `book1` and `book2` each have the four elements of the structure lib_books.
- Structures do not occupy any memory until associated with an actual structure variable such as `book1,` because they are templates.
- Structures are passed by value (so the arrays in `book1` or `book2` would be passed by value, and thus completely copied).
- *Structure names are not pointers.*

Structure Declaration (3/3)

```
struct lib_books {
  char title[20];
  char author[15];
  int pages;
  float price;
}
book1,book2,books[100],*b
kPtr;
```

is valid,

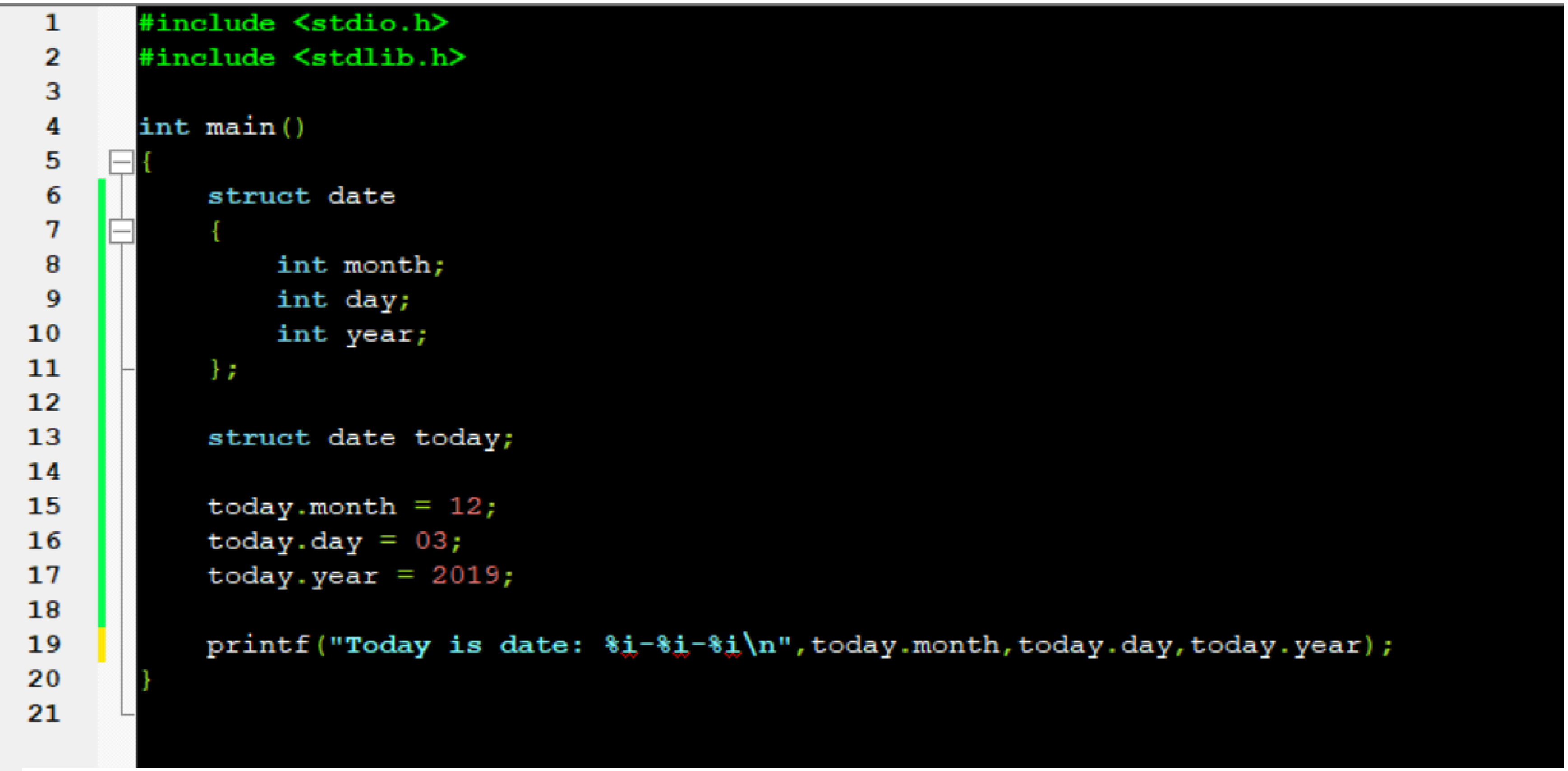
=

```
struct lib_books {
  char title[20];
  char author[15];
  int pages;
  float price;
};

struct lib_books
book1,book2,books[100],*b
kPtr;
```

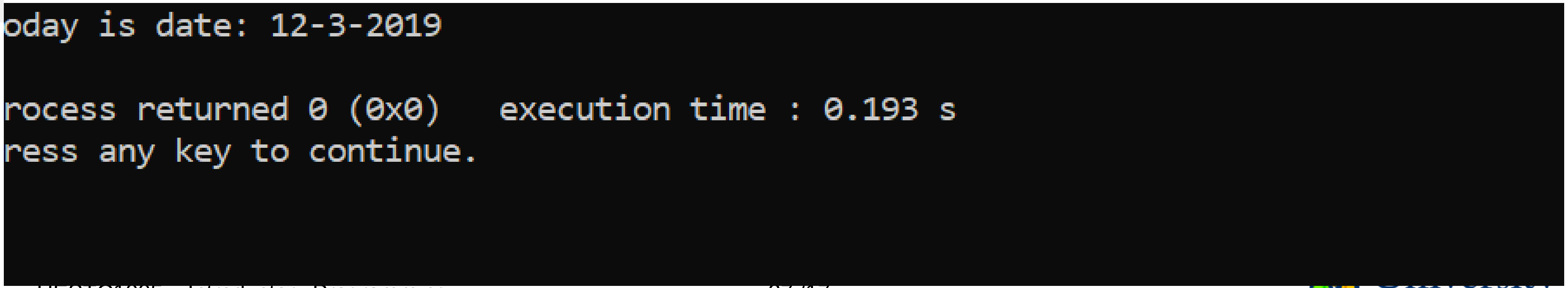
Accessing members in Structure

- A structure variable name is not a pointer
  - You need special syntax to access the members
- You access the member of the structure by **writing the variable name followed by a period**, followed by the member variable name
  - The period is called the member selection operator (aka dot operator)
  - There are no spaces permitted between the variable name, the period, and the member name.
  - To set the value of title in book 1, you write
    - book1.title = “Spring”;
    - book1.author = “ahmed”;

Display Date Struct Example



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

int main()
{
    struct date
    {
        int month;
        int day;
        int year;
    };

    struct date today;

    today.month = 12;
    today.day = 03;
    today.year = 2019;

    printf("Today is date: %i-%i-%i\n",today.month,today.day,today.year);
}
```



```
oday is date: 12-3-2019

rocess returned 0 (0x0)   execution time : 0.193 s
ress any key to continue.
```

Initializing Structures

- Initializing structures is similar to initializing arrays
  - The elements are listed inside a pair of braces, with each element separated by comma
  - E.g.; struct date today ={7,2,2019};
- Just like an array initialization, fewer values might be initialized than are contained in the structure
  - Example: struct date date1 = {12, 10}; which sets the date1.month to 12, date.date to 2, and no value to date.year .
- Another method is to use .member = value method
  - Example struct date dates1 = {.month =12, .date=10};